The Open Group Standard

FACE™ Technical Standard, Edition 3.1

The Open Group Standard

FACE™ Technical Standard, Edition 3.1

ISBN: 1-947754-61-4

Document Number: C207

Published by The Open Group, July 2020.

Comments relating to the material contained in this document may be submitted to:

The Open Group, Apex Plaza, Forbury Road, Reading, Berkshire, RG1 1AX, United Kingdom

or by electronic mail to:

ogface-admin@opengroup.us

Contents

Table of Figures

Table of Tables

Preface

Introduction

This document is the Future Airborne Capability Environment™ Technical Standard, Edition 3.1, known as the FACE™ Technical Standard. This Standard is developed and maintained by The Open Group FACE Consortium.

Background

Today's military aviation airborne systems typically entail a unique set of requirements and a single vendor. This form of development has served the military aviation community well; however, this stovepipe development process has had some undesired side-effects including long lead times, cumbersome improvement processes, lack of hardware and software reuse between various aircraft platforms resulting in platform-unique designs.

The advent of complex mission equipment and electronics systems has caused an increase in the cost and schedule to integrate new hardware and software into aircraft systems. This – combined with the extensive testing and airworthiness qualification requirements – has begun to affect the ability of the military aviation community to deploy new capabilities across the military aviation fleet.

The current military aviation community procurement system does not promote the processes of hardware and software reuse across different programs. In addition, the current aviation development community has not created sufficient standards to facilitate the reuse of software components across the military aviation fleet. Part of the reason for this is the small military aviation market. Another part is the difficulty in developing qualified software for aviation. An additional problem is the inability to adopt current commercial software Common Operating Environment (COE) standards because they do not adhere to the stringent safety requirements developed to reduce risk and likelihood of loss of aircraft, reduced mission capability, and ultimately loss of life.

To counter these trends, the Naval Aviation Air Combat Electronics program office (PMA-209), Army Program Executive Office (PEO) Aviation (AVN), the Army's Aviation and Missile Research, Development, and Engineering Center (AMRDEC), and Air Force Research Laboratory (AFRL), enabled by the expertise and experience of the military aviation community's industrial base, are adopting a new approach.

Approach

The FACE approach addresses the affordability initiatives of today's military aviation domain. The FACE approach is to develop a Technical Standard for a software COE designed to promote portability and create software product lines across the military aviation domain. Several components comprise the FACE approach to software portability and reuse:

- Business processes to adjust procurement and incentivize industry
- Technical practices to promote development of reusable software components

- A software standard to promote the development of portable components between differing avionics architectures

The FACE approach allows software-based "capabilities" to be developed as software components that are exposed to other software components through defined interfaces. It also provides for the reuse of software across different hardware configurations.

Ultimately, the FACE key objectives are to reduce development costs, integration costs, and time-to-field for avionics capabilities.

FACE Artifacts

The following documents provide definition and support of the FACE technical and business practices:

- FACE Business Guide
- FACE Technical Standard
- FACE Conformance Policy
- FACE Reference Implementation Guide
- FACE Library Policy
- FACE Contract Guide
- FACE Problem Report (PR)/Change Request (CR) Process

Additional information can be found at www.opengroup.org/face/information.

The Open Group

The Open Group is a global consortium that enables the achievement of business objectives through technology standards. Our diverse membership of more than 750 organizations includes customers, systems and solutions suppliers, tools vendors, integrators, academics, and consultants across multiple industries.

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
- Developing and operating the industry's premier certification service and encouraging procurement of certified products

Further information on The Open Group is available at www.opengroup.org.

The Open Group publishes a wide range of technical documentation, most of which is focused on development of Standards and Guides, but which also includes white papers, technical studies, certification and testing documentation, and business titles. Full details and a catalog are available at www.opengroup.org/library.

Trademarks

Acknowledgements

The Open Group gratefully acknowledges the contribution of the following people in the development of this document:

Principal Authors

- Don Akers, Boeing
- William Antypas, NAVAIR
- Kirk Avery, Lockheed Martin, Technical Working Group (TWG) Chair
- David Bowes, NAVAIR
- Edward Burke, MITRE Corporation
- Stephanie Burns, Rockwell Collins
- Spencer Crosswy, NAVAIR
- James “Bubba” Davis, U.S. Army AMRDEC
- Joe Dusio, Rockwell Collins
- Matthew Eby, NAVAIR
- Christopher J. Edwards, U.S. Army AMRDEC
- Stuart Frerking, NAVAIR
- Stephen Fulmer, U.S. Army AMRDEC
- Jeff Hegedus, Raytheon
- Daniel Herring, CoreAVI
- Patrick Huyck, Green Hills Software
- Paul Jennings, Presagis USA, Inc.
- William Kimmel, NAVAIR, TWG Vice-Chair
- Bill Kinahan, Sikorsky Aircraft
- Marc Moody, Boeing
- Sean Mulholland, TES-SAVi
- Joseph Neal, Harris Corporation
- Marcell Padilla, NAVAIR
- Allan Reynolds, NAVAIR

- Joel Sherrill, U.S. Army AMRDEC
- Robert Sweeney, NAVAIR
- Levi Van Oort, Rockwell Collins
- Scott Wigginton, U.S. Army AMRDEC

Additional Contributors

- Joshua Anderson, NAVAIR
- Scott Bessler, CMC Electronics
- David Bockenfeld, CMC Electronics
- Mathias Boddicker, NAVAIR
- Kevin Buesing, Objective Interface Systems
- Rafael J. Cajigas, NAVAIR
- H. Glenn Carter, U.S. Army AMRDEC
- Judy Cerenzia, The Open Group
- Paul Chen, Wind River
- Robert Daniels, NAVAIR
- Gary Gilliland, DDC-I
- Charles Stephen Kuehl, Raytheon
- Leanne May, Rockwell Collins
- Steve Mills, GoAhead
- Marbert Moore, III, NAVAIR
- Pramod Patel, Honeywell Aerospace
- Bruce Pulliam, Raytheon
- Charlie Rush, Objective Interface Systems
- Joe Schlesselman, Real Time Innovations
- Stephen Smith, NAVAIR
- Jonathan Strootman, TES-SAVi
- John Tencate, GE Aviation
- Terence Thomason, General Dynamics Mission Systems
- Scott Tompkins, U.S. Army AMRDEC
- Jason York, U.S. Army AMRDEC

With special thanks to W. Mark Vanfleet, NSA.

This Edition of the FACE Technical Standard is dedicated to Don Akers for his technical contributions, cheerful countenance, and dedication in the face of personal adversity. He will always be remembered by FACE Technical Working Group members.

Funding for the FACE Consortium and its work products comes from the following organizations, which at the time of publication include:

Sponsors: Air Force Research Laboratory, Boeing, Lockheed Martin, Rockwell Collins, U.S. Army PEO Aviation

Principals: AeroVironment Inc., BAE Systems, BELCAN, Booz Allen Hamilton, DRS Training & Control Systems, Elbit Systems of America, GE Aviation Systems, General Dynamics, Green Hills Software, Harris Corp., Honeywell Aerospace, IBM, Northrop Grumman, Raytheon, Sierra Nevada Corporation, Sikorsky Aircraft, Textron Systems, U.S. Army AMRDEC, Wind River

Associates: Abaco Systems, AdaCore, Arizona State Univ., ARTEMIS Inc., Astronautics, Avalex Technologies, Brockwell Technologies, Carnegie Mellon University-SEI, CERTON, CMC Electronics, Cobham Aerospace Communications, CoreAVI, CS Communication & Systems Inc., Crossfield Technology, CTSi, Curtiss-Wright Defense Solutions, DDC-I, DornerWorks, Draper Lab, Elma Electronic Inc., Enea Software & Services Inc., ENSCO Avionics, Esterel Technologies, Esterline AVISTA, EuroAvionics USA, Garmin International Inc, GECO Inc., General Atomics ASI, IEE, Infinite Dimensions Integration, Inter-Coastal Electronics, Johns Hopkins University Applied Physics Lab, Joint Tactical Networking Center, Kaman Precision Products, KEYW Corporation, KIHOMAC, L-3 Communications, LDRA Technology, Leidos, Lynx Software, Mercury Systems, North Atlantic Industries, OAR Corporation, Performance Software, Physical Optics Corp., Presagis USA Inc., PrismTech, Pyrrhus Software, Rapid Imaging Software, Real-Time Innovations, Riverside Research, Rogerson Kratos, SAIC, Selex Galileo Inc., SimVentions, Skayl, Southwest Research Institute, StackFrame, Technology Service Corporation, Terma North America, TES-SAVI, Thales USA Inc., Thomas Production Company, Trideum, TTTech N.A., Univ. of Dayton Research Institute, Vector Software Inc., Verocel, VTS Inc., Zodiac Data Systems

And **Naval Air Systems Command (NAVAIR) under NAVAIR Cooperative Agreement No. N00421-12-2-0004.** The U.S. notwithstanding any copyright notation thereon. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Naval Air Warfare Center Aircraft Division or the U.S. Government.

Referenced Documents

Normative References

Normative references for the FACE Technical Standard are defined in Section 1.4.

Informative References

The following documents are referenced in the FACE Technical Standard:

- Department of Defense Instruction 8510.01, Risk Management Framework (RMF) for DoD Information Technology (IT) (DIARMF), March 12, 2014
- Department of Defense Reference Architecture Description, prepared by the Office of the DoD CIO, June 2010
- Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, ARP 4761, 1996
- Guidelines for Development of Civil Aircraft and Systems, ARP 4754A, 2010
- Guidelines for the Use of the C Language in Critical Systems, Motor Industry Software Reliability Association (MISRA), October 2004
- Guidelines for the Use of the C++ language in Critical Systems, Motor Industry Software Reliability Association (MISRA), June 2008
- ISO/IEC 12207:2008: Systems and Software Engineering – Software Lifecycle Processes
- ISO/IEC 24765:2010: Systems and Software Engineering Vocabulary
- Object Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A, DO-332, December 2011
- OMG CORBA, Version 3.1, January 2008 (www.omg.org/spec/CORBA)
- OMG: Data Distribution Service (DDS) for Real-Time Systems Specification, Version 1.2, January 2007 (www.omg.org/spec/DDS/1.2)
- Open Systems Management Plan (OSMP) Data Item Description (DID) DI-MGMT-82099, U.S. Army AMC Aviation & Missile Command, January 11, 2017
- Software Considerations in Airborne Systems and Equipment Certification, ED-12B/DO-178B, December 1992
- Software Considerations in Airborne Systems and Equipment Certification, ED-12C/DO-178C, January 2012
- Software Integrity Assurance Considerations for Communication, Navigation, Surveillance, and Air Traffic Management (CNS/ATM) Systems, DO-278A, December 2011

1 Introduction

1.1 Objective

The FACE Technical Standard defines a Reference Architecture intended for the development of portable software components targeted for general purpose, safety, and/or security purposes. A more detailed explanation of the FACE Reference Architecture is found in Chapter 3.

The objectives are to:

- Define the FACE Reference Architecture for developing and verifying software components
- Define the FACE Reference Architecture for defining interfaces allowing communication between software components
- Enable affordability, interoperability, and time-to-field across military systems based upon fundamental software engineering principles and practical experience

1.2 Overview

This document embodies a set of requirements and descriptions referred to as the FACE Technical Standard. The FACE Technical Standard uses industry standards for distributed communications, programming languages, graphics, operating systems, and other areas as appropriate.

The FACE Technical Standard contains an architectural overview in Chapter 3 introducing the Reference Architecture, followed by a detailed description of each architectural segment and interface. This document then outlines requirements of each software component of the Reference Architecture. Finally, the appendices list the specific Application Programming Interfaces (APIs) required by the FACE Technical Standard, the FACE Data Architecture required by the FACE Technical Standard, as well as other applicable standards. Specific FACE terms are defined in Chapter 2.

1.2.1 Background

The aviation development community has a unique subset of constraints pertaining to the development of military systems, often adhering to varying degrees of flight-safety and security requirements. Many of these systems have advanced the state-of-the-art in software design and implementation practices. Despite the technical advances, many software systems today result in the tightly-coupled integration of software components without regards to portability. This lack of focus on portability results in software components that require a specific software architecture in order to function correctly and an inability to reuse a software capability from one platform to another without significant software modification. A side-effect of this is vendor-lock as only the original implementer can efficiently make software modifications.

1.2.2 Technical Approach

The FACE technical approach tackles barriers to modularity, portability, and interoperability by defining a Reference Architecture and employing design principles to enhance software portability. To meet the objectives of the FACE Technical Standard, several software engineering practices have been employed focusing on the following principals:

- Use published industry standards to provide normative references, allowing the use of existing software libraries and tools whenever possible
- Use profiles to define subsets of those standards when support of the entire standard would lead to safety or security certification issues, or when supporting only a defined subset would lead to a more cost effective solution; a profile can also reference a specific version of a standard in its entirety
- Use a standardized architecture describing a conceptual breakdown of functionality and the FACE Reference Architecture to promote the reuse of software components to share common functionality across military systems
- Define standardized interfaces to allow software components to be moved between systems developed by different vendors
- Use a data architecture to ensure the data communicated between the software components is fully described to facilitate the integration on new systems
- Require that hardware abstraction be used to decouple software components from specific hardware implementations, and device driver normalization be used to allow interfaces to external devices to be developed independently of the computing platform device drivers
- Use a display window management strategy to incorporate common avionics user interface standards to aid in the integration of components needing to share display areas and input devices

This technical approach enables software components to be redeployed on other platforms to achieve greater portability and interoperability when standardized FACE Interfaces are used.

1.3 Conformance

The FACE Consortium has established a FACE Conformance Program and defined an associated Conformance Policy for the FACE Technical Standard. The FACE Conformance Program provides the associated conformance criteria and processes necessary to assure Units of Conformance (UoCs) are developed according to the FACE Technical Standard. The FACE Conformance Program consists of Verification, Certification, and Registration.

FACE Verification is the process of determining the conformance of a UoC implementation to the applicable FACE Technical Standard requirements. Verification is performed using the Conformance Verification Matrix and running the Conformance Test Suite. Verification is handled through a Verification Authority (VA).

FACE Certification is the process of applying for FACE Conformance Certification once verification has been completed. Certification is processed through the FACE Certification Authority (CA).

FACE Registration is the process of listing FACE Certified UoCs in a public listing of FACE Certified UoCs known as the FACE Registry. The FACE Registry is accessed from the FACE Landing Page.

Successful completion of the FACE Conformance Program leads to a FACE Conformance Certificate and the right to use the FACE Conformance Certification Trademark.

1.4 Normative References

The following standards contain provisions that, through references in this FACE Technical Standard, also constitute provisions of the FACE Technical Standard Profiles. At the time of publication, the editions indicated were valid.

- ANSI/ISO/IEC 8652:1995: Ada 95 Reference Manual, Language and Standard Libraries
- ANSI/TIA-232-F: Interface Between Data Terminal Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange, October 2002
- ANSI/TIA-422-B: Electrical Characteristics of Balanced Voltage Digital Interface Circuits, September 2005
- ARG Ravenscar Profile for High-Integrity Systems, Technical Report, ISO/IEC/JTC1/SC22/WG9, AI-00249, 2003
- ARINC Characteristic 739-1: Multi-purpose Control and Display Unit (MCDU), June 1990
- ARINC Characteristic 739A-1: Multi-purpose Control and Display Unit (MCDU), December 1998
- ARINC Report 661-5: Cockpit Display System Interfaces to User System, July 2013
- ARINC Report 664: Aircraft Data Network, September 2009
- ARINC Specification 429: Mark 33 Digital Information Transfer System (DITS), May 2004
- ARINC Specification 653P1-3: Avionics Application Software Standard Interface, Part 1: Required Services, November 2010
- ARINC Specification 653P1-4: Avionics Application Software Standard Interface, Part 1: Required Services, August 2015
- ARINC Specification 653P2-3: Avionics Application Software Standard Interface, Part 2: Extended Services, August 2015
- ARINC Specification 825-4: General Standardization of CAN (Controller Area Network) Bus, September 2018
- Department of Defense: IPv6 Standard Profiles for IPv6-Capable Products, Version 5.0, July 2010
- Department of Defense: Joint Technical Architecture, Volume I, Version 6, October 2003

- EIA/TIA-485-A: Electrical Characteristics of Generators and Receivers for Use in Balanced Digital Multipoint Systems, March 2003
- Extensible Markup Language (XML), Version 1.0 (Fifth Edition), November 26, 2008, W3C (www.w3.org/XML)
- Extensible Markup Language (XML) Schema Definition (XSD), Version 1.0, 2004, W3C (www.w3.org/XML)
- Future Airborne Capability Environment™ Shared Data Model (SDM) Governance Plan, Edition 2.1, October 2015
- IEEE Std 754-2008: IEEE Standard for Floating-Point Arithmetic, August 2008
- IEEE Std 1003.1-2008: IEEE Standard for Information Technology – Portable Operating System Interface (POSIX®) – Base Specifications, Issue 7, December 1, 2008
- IEEE Std 1003.13-2003: IEEE Standard for Information Technology – Standardized Application Environment Profile (AEP) – POSIX® Realtime and Embedded Application Support, Issue 10, September 2004
- IETF RFC 0768: User Datagram Protocol, August 1980 (http://tools.ietf.org/html/rfc768)
- IETF RFC 0791: Internet Protocol (IP), September 1981 (http://tools.ietf.org/html/rfc791)
- IETF RFC 0793: Transmission Control Protocol, September 1981 (http://tools.ietf.org/html/rfc793)
- IETF RFC 1112: Host Extensions for IP Multicasting, August 1989 (http://tools.ietf.org/html/rfc1112)
- IETF RFC 2460: IPv6 Specification, December 1998 (http://tools.ietf.org/html/rfc2460)
- IETF RFC 3390: Increasing TCP's Initial Window, October 2002 (http://tools.ietf.org/html/rfc3390)
- IETF RFC 4213: Basic Transition Mechanisms for IPv6 Hosts and Routers, October 2005 (http://tools.ietf.org/html/rfc4213)
- IETF RFC 5424: The Syslog Protocol, March 2009 (http://tools.ietf.org/html/rfc5424)
- ISO/IEC 8652:2012(E) (with Corrigendum 1): Ada 2012 – Reference Manual, Language, and Standard Libraries, March 2016
- ISO/IEC 9899:1999: Programming Languages – C
- ISO/IEC 14882:2003: Programming Languages – C++
- ISO/IEC TR 15942:2000: Information Technology – Programming Languages – Guide for the Use of the Ada Programming Language in High Integrity Systems
- Java Platform, Enterprise Edition (Java EE) Specification, v8 (Java EE 8), July 2017
- Java Platform, Standard Edition 8 (Java SE 8), March 2014
- Khronos Native Platform Graphics Interface, EGL Version 1.4, December 2013 (www.khronos.org/registry/EGL/specs/eglspec.1.4.pdf)

- MIL-STD-1553B: Aircraft Internal Time Division Command/Response Multiplex Data Bus, United States Department of Defense, September 1978
- MIL-STD-1553B Notice 2: Digital Time Division Command/Response Multiplex Data Bus, United States Department of Defense, September 1986
- OMG: Ada Language Mapping, Version 1.3, June 2010 (www.omg.org/spec/ADA/1.3)
- OMG: C Language Mapping Specification, Version 1.0, June 1999 (www.omg.org/spec/C/1.0)
- OMG: C++ Language Mapping for, Version 1.3, July 2012 (www.omg.org/spec/CPP/1.3)
- OMG: C++11 Language Mapping, Version 1.2, August 2015 (www.omg.org/spec/CPP11/1.2/PDF)
- OMG: IDL to Java Language Mapping, Version 1.3, January 2008 (www.omg.org/spec/I2JAV/1.3)
- OMG: Interface Definition Language, Version 4.1, May 2017 (www.omg.org/spec/IDL/4.1)
- OMG (n.d.) Meta-Object Facility, Version 2.0, January 2006 (www.omg.org/spec/MOF/2.0)
- OMG (n.d.) Object Constraint Language, Version 2.4, February 2014 (www.omg.org/spec/OCL/2.4)
- Open Universal Domain Description Language (Open UDDL), Edition 1.0 (C198), July 2019, published by The Open Group; refer to: www.opengroup.org/library/c198
- OpenGL ES Common Profile Specification, Version 2.0.25, November 2010
- OpenGL SC Safety-Critical Profile Specification, Version 1.0.1, March 2009
- OpenGL SC Safety-Critical Profile Specification, Version 2.0, April 2016
- OSGi (Open Services Gateway initiative) Service Platform, Release 7, April 2018 (http://www.osgi.org/release7)

1.5 Terminology

For the purposes of the FACE Technical Standard, the following terminology definitions apply:

Can	Describes a feature or behavior that is optional. The presence of this feature should not be relied upon.
May	Describes a feature or behavior that is optional. The presence of this feature should not be relied upon.
Must	Describes a feature or behavior that is strongly recommended for an implementation that conforms to this document. A software component cannot rely on the existence of the feature or behavior.

Shall	Describes a feature or behavior that is mandatory for an implementation that conforms to this document. A software component relies on the existence of the feature or behavior.
Should	Describes a feature or behavior that is strongly recommended for an implementation that conforms to this document. A software component cannot rely on the existence of the feature or behavior.

1.6 Future Directions

The FACE Technical Standard has been developed as a cooperative effort by the diverse members of the FACE Consortium. The FACE Consortium is a Voluntary Consensus Standards Body comprised of U.S. Government, Industry, and Academia representatives. Potential improvements, corrections, and additions may be suggested via the FACE PR/CR Process (https://ticketing.facesoftware.org). Additionally, an organization may join the FACE Consortium and actively participate in the development of future editions of the FACE Technical Standard. For more information about how to participate in the FACE Consortium, visit the FACE Consortium website (www.opengroup.org/face).

2 Definitions

The list of terms and definitions provided in this chapter is not intended to be exhaustive, but includes concise definitions for principal terms commonly used throughout this document, as well as for referenced FACE terms and concepts that are more thoroughly discussed in other FACE documentation.

2.1 Component State Persistence

A Portable Components Segment, Platform-Specific Services Segment, or Transport Services Segment Unit of Conformance internal state saved to a Data Store.

2.2 Data Architecture

A set of related models, specifications, and governance policies with the primary purpose of providing an unambiguous description of exchanged data and an interoperable means of data exchange.

2.3 Data Model Language

A language specified as an EMOF metamodel and OCL constraints used to capture data element syntax and semantics.

2.4 Domain-Specific Data Model

A Data Model designed to the FACE Data Architecture Requirements. It captures domain-specific semantics.

2.5 FACE Architectural Segments

There are five (5) logical segments in the FACE Reference Architecture: one, Portable Components Segment (PCS); two, Transport Services Segment (TSS); three, Platform-Specific Services Segment (PSSS); four, Operating System Segment (OSS); and five, Input/Output Services Segment (IOSS). FACE Architectural Segments are connected by three (3) FACE Standardized Interfaces (*aka* as "KEY" Interfaces per a Modular Open Systems Approach (MOSA)).

2.6 FACE Computing Environment

A generic concept and is instantiated for a particular system under development. It includes all elements of the FACE Reference Architecture necessary to deploy FACE conformant

components. The FACE Computing Environment is composed of the software infrastructure (Transport Services, Operating System, and I/O Services Segments) and the Platform-Specific Services Segment required by the FACE components.

2.7 FACE Infrastructure

An implementation of a FACE Operating System Segment, I/O Services Segment, and Transport Services Segment that is capable of hosting PCS and PSSS software components that are aligned to the FACE Technical Standard.

Note: PCS and PSSS components are not required to have a FACE conformant "stamp" in order to be integrated.

2.8 FACE Interfaces

Standardized interfaces providing connections between software components of the FACE Segments.

2.9 I/O Service

A collection of software components that provides a unified view of an I/O Services Interface to all Platform-Specific Services Segment software components using that interface.

2.10 I/O Services Segment

Segment where normalization of vendor-supplied interface hardware device drivers occurs.

2.11 Operating System Segment

Segment where foundational system services used by all other segments and vendor-supplied code reside.

2.12 Platform-Specific Common Services

Sub-segment comprised of higher-level services including Logging Services, Centralized Configuration Services, DPM Services, Streaming Media, and System-Level HMFM.

2.13 Platform-Specific Device Services

Sub-segment where management of data and translation between platform-unique ICDs and the FACE Data Model occurs.

2.14 Platform-Specific Graphics Services

Sub-segment that abstracts the interface specifics of a graphics device driver from software components within the FACE Reference Architecture.

2.15 Platform-Specific Services Segment

Segment comprised of sub-segments including Platform-Specific Common Services, Platform-Specific Device Services, and Platform-Specific Graphics Services.

2.16 Portable Components Segment

Segment where software components providing capabilities and/or business logic reside.

2.17 Security Transformation

Transformation of data to meet system-level security controls.

2.18 Security Transformation Boundary

A boundary between a software component's function and the Security Transformation algorithm. This boundary may exist at a FACE Interface, or internal to a Unit of Conformance that contains its own Security Transformation.

2.19 Shared Data Model

An instance of a Data Model whose purpose is to define commonly used items and to serve as a basis for all other data models. Alignment with the required elements in the Shared Data Model is necessary for conformance of any other data model. The Shared Data Model is governed by a Configuration Control Board.

2.20 Transport Protocol Module

TPM enables interoperability between Transport Services Domains providing connectivity to physical transports (e.g., TCP, UDP, RTPS, IIOP, Queues, IPC, IP, shared memory) not natively supported by a TSS.

2.21 TS Domains

All of the Units of Conformance that constitute a system utilizing a single implementation of Transport Services Segment UoCs and integrated by the same integrator.

2.22 Transport Services Segment

Segment which abstracts transport mechanisms and data access from software components facilitating integration into disparate architectures and platforms using different transports.

2.23 TS to TS Interoperability

The ability to exchange and use information between Transport Services Domains without re-engineering the TSS UoCs in either TSS Domain. Configuration and/or access to additional TSS capabilities may be required.

2.24 Unit of Conformance

A software component or domain-specific data model designed to meet the applicable requirements defined in the FACE Technical Standard. It is referenced as a UoC at any point in its development, and becomes a FACE Certified UoC upon completion of the FACE Conformance Process.

2.25 Unit of Conformance Package

A collection of Units of Conformance combined to create a singular software logical entity which may be placed in the Registry. The Units of Conformance that make up a Unit of Conformance Package may be from different FACE Segments.

2.26 Unit of Portability

A UoC that resides within a PCS or PSSS.

2.27 UoP Supplied Model

A data model provided by a software supplier that documents the data exchanged by a Unit of Conformance via the Transport Services Interface.

3 Architectural Overview

3.1 FACE Architectural Segments

The FACE Reference Architecture is comprised of logical segments where variance occurs. The structure created by connecting these segments together is the foundation of the FACE Reference Architecture.

The five (5) segments of the FACE Reference Architecture are listed below and introduced in the following subsections:

- Operating System Segment (OSS)
- Input/Output Services Segment (IOSS)
- Platform-Specific Services Segment (PSSS)
- Transport Services Segment (TSS)
- Portable Components Segment (PCS)

Figure 1 depicts a representation of the FACE Reference Architecture illustrating the OSS as the foundation upon which the other FACE segments are built. The segments introduced in Figure 1 are defined and described in Section 3.1.1 through Section 3.1.5. This architecture is constrained to promote separation of concerns and establish a product line approach for reusable software capabilities.

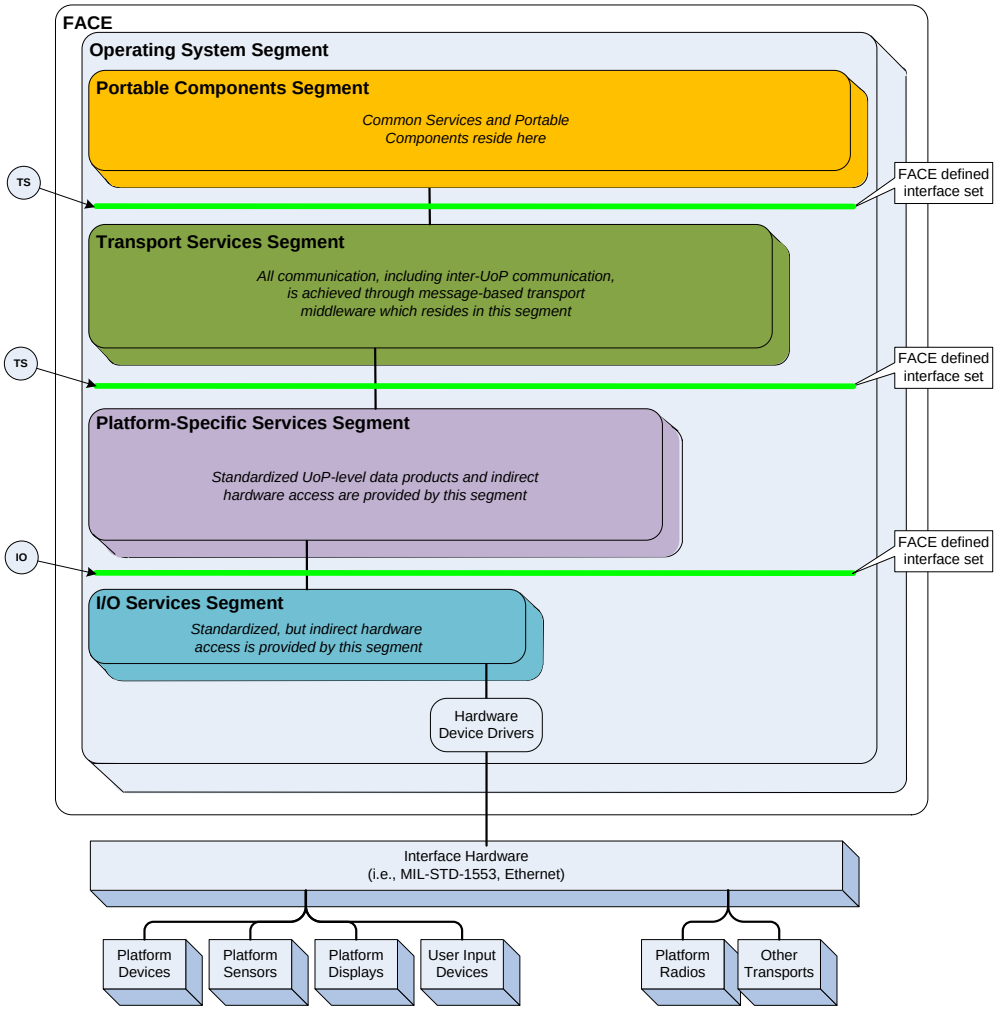

Figure 1: FACE Architectural Segments

3.1.1 Operating System Segment

The OSS is where foundational system services and vendor-supplied software reside. An OSS UoC provides and controls access to the computing platform. An OSS UoC supports the execution of all FACE UoCs and hosts various operating system, integration, and low-level health monitoring interfaces. An OSS UoC can also optionally provide external networking capabilities, Programming Language Run-Times, Component Framework, Life Cycle Management, and Configuration Services capabilities.

3.1.2 Input/Output Services Segment

The IOSS is where normalization of vendor-supplied interface hardware device drivers occurs. IOSS UoCs provide the abstraction of the interface hardware and drivers from the PSSS UoCs. This allows the PSSS UoCs to focus on the interface data and not the hardware and driver specifics.

3.1.3 Platform-Specific Services Segment

The PSSS is comprised of sub-segments including Platform-Specific Device Services, Platform-Specific Common Services, and Platform-Specific Graphics Services.

3.1.3.1 Platform-Specific Device Services

Platform-Specific Device Services (PSDS) are where management of data and translation between platform-unique Interface Control Documents (ICDs) and FACE UoP Supplied Models (USMs) occurs.

3.1.3.2 Platform-Specific Common Services

Platform-Specific Common Services (PSCS) are comprised of higher-level services including Logging Services, Device Protocol Mediation (DPM) Services, Streaming Media, Health Monitoring and Fault Management (HMFM), and Configuration Services.

3.1.3.3 Platform-Specific Graphics Services

Platform-Specific Graphics Services (PSGS) is where presentation management occurs. PSGS abstracts the interface specifics of Graphics Processing Units (GPU) and other graphics devices from software components within the FACE Reference Architecture.

3.1.4 Transport Services Segment

The TSS is comprised of communication services. The TSS abstracts transport mechanisms and data access from software components facilitating integration into disparate architectures and platforms using different transports. TSS UoCs are responsible for data distribution between PCS and/or PSSS UoCs. TSS capabilities include, but are not limited to, distribution and routing, prioritization, addressability, association, abstraction, transformation, and component state persistence of software component interface information.

3.1.5 Portable Components Segment

The PCS is comprised of software components providing capabilities and/or business logic. PCS components are intended to remain agnostic from hardware and sensors. Additionally, these components are not tied to any data transport or operating system implementations to meet objectives of portability and interoperability.

3.2 FACE Standardized Interfaces

The FACE Reference Architecture defines a set of standardized interfaces providing connections between the FACE Architectural Segments. The standardized interfaces within the FACE Reference Architecture are the Operating System Segment Interface (OSS Interface), the Input/Output Services Interface (IOS Interface), the Transport Services Interfaces, and Component-Oriented Support Interfaces. Software references to these standardized interfaces may be established at states such as initialization, startup, run-time, etc.

3.2.1 Operating System Segment Interface

The OSS Interface provides a standardized means for software to use the services within the operating system and other capabilities related to the OSS. The OSS Interface is provided by an

OSS UoC to UoCs in other segments. This interface includes ARINC 653, POSIX®, and HMFM APIs. The OSS Interface optionally includes one or more of the following networking capabilities: Programming Language Run-Times, Component Frameworks, Life Cycle Management, and the Configuration Services interface.

3.2.2 Input/Output Services Interface

The IOS Interface provides a standardized means for software components to communicate with device drivers. This interface supports several common I/O bus architectures.

3.2.3 Transport Services Interface

The Transport Services Interface provides a standardized means for software to use communication services provided by the TSS. The Type-Specific (TS) Interface is provided by software components within the TSS to/from software components within the PSSS and PCS. The FACE Data Architecture governs the representation of the data traversing the Transport Services Interface.

3.2.4 Component-Oriented Support Interfaces

Component-Oriented Support Interfaces include the Injectable Interface and the Life Cycle Management Services Interface. These interfaces are FACE Standardized Interfaces addressing cross-cutting concerns for component support, and thus are not depicted in the FACE Reference Architecture.

3.2.4.1 Injectable Interface

The Injectable Interface provides a standardized means for integration software to resolve the inherent using/providing interface dependency between software components. In order for a software component to use an interface, it must be integrated in the same address space with at least one software component that provides that interface. The Injectable Interface implements the dependency injection idiom of software development.

3.2.4.2 Life Cycle Management Services Interfaces

The Life Cycle Management (LCM) Services Interfaces provide a standardized means for software components to support behaviors that are consistent with Component Frameworks: initialization, configuration, framework startup/teardown, and operational state transitions. The LCM Services Interfaces are optionally provided by a software component in any of the FACE Reference Architecture segments, and are optionally used by the system integration implementation or by a software component in any of the FACE Reference Architecture segments.

3.3 FACE Data Architecture

3.3.1 FACE Data Architecture Overview

The FACE Data Architecture:

- The FACE Data Model Language leverages the Open Universal Domain Description Language (Open UDDL) to define the FACE Shared Data Model (SDM) and Unit of Portability (UoP) Supplied Models (USM)

- Defines the FACE Data Model Language, specified by an Essential Meta-Object Facility (EMOF) metamodel and a set of Object Constraint Language (OCL) constraints
- Defines a Template Language to specify presentation of data elements across key FACE Interfaces
- Defines the FACE Data Model Language binding specification describing how elements specified in the Open UDDL Technical Standard are mapped to data types and/or structures for each of the supported programming languages
- Provides the SDM which allows standardized definitions to be used across all FACE conformant data models
- Defines the rules of construction of the USM

  Each PCS UoC, PSSS UoC, or TSS UoC using TS Interfaces is accompanied by a USM consistent with the FACE SDM and defines its interfaces in terms of the FACE Data Model Language. A Domain-Specific Data Model (DSDM) captures content relevant to a domain of interest and can be used as a basis for USMs. DSDMs must be consistent with the FACE SDM. For a more detailed definition of the DSDM, see Section 4.9.3.3.

3.3.2 FACE Data Model Language

The FACE Data Model Language enforces a multi-level approach to modeling entities and their associations at the conceptual, logical, and platform levels, enabling gradual and varying degrees of abstraction. Entities, their characteristics, and associations provide context for the specification of views defining the data exchanges between UoPs.

The FACE Data Model Language supports the modeling of abstract UoPs to provide a specification mechanism for procurement, or for defining elements based on the FACE Technical Standard and used in other reference architectures. To address integration between UoPs, the FACE Data Model Language provides elements for describing the high-level connectivity, routing, and transformations to be embodied in an instance of transport services.

The FACE Technical Standard contains different representations of the FACE Data Model Language: a textual listing generated from the metamodel defined in Section 4.9; and an Extensible Markup Language (XML) Metadata Interchange (XMI) listing from an export of the metamodel that conforms to EMOF shown in Section J.4.

3.3.3 Data Architecture Governance

The FACE SDM Governance Plan establishes the authority and operating parameters of the FACE SDM Configuration Control Board (CCB). The FACE SDM Governance Plan manages growth through extensions and ensures the necessary conformance of new SDM elements. The SDM consists of basis elements and any extension(s) in the Conceptual Data Model (CDM), Logical Data Model (LDM), and Platform Data Model (PDM). The FACE SDM Governance Plan details a complete process, a set of rules and roles and responsibilities for the FACE SDM CCB, suppliers, and system integrators.

3.4 Reference Architecture Segment Example

Figure 2 shows an example of the software components that could reside in each of the FACE Architectural Segments. This is a diagram intended to clarify the purpose of the segments and does not represent a prescriptive description of elements required for FACE alignment, nor is it all-inclusive as to the potential functions that could reside in each segment. The rules for and descriptions of the software components that belong in each segment are defined in Section 3.8.

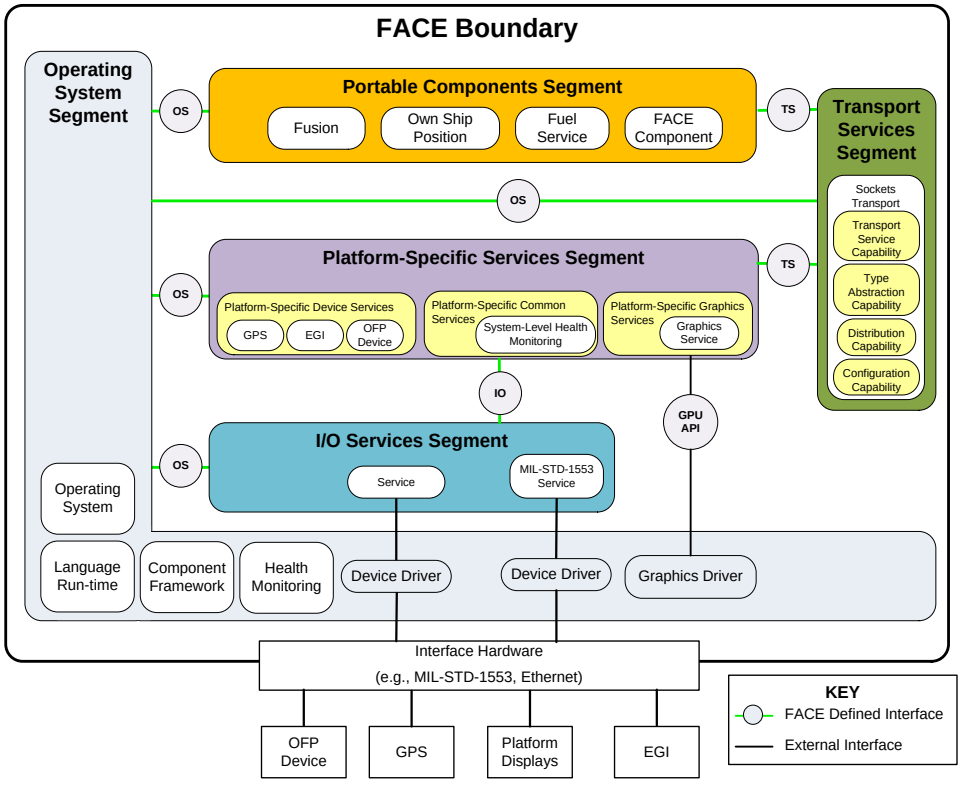

Figure 2: Architectural Segments Example

The following narrative describes a simple example implementation. In this scenario we have a simple system that uses navigation data from a Global Positioning System (GPS) device via a MIL-STD-1553 interface hardware to provide Own Ship Position symbology to a display. The software is implemented as several components designed for reuse and portability.

Within the "FACE Boundary", the OSS provides a well-known set of Application Programming Interfaces (APIs) and services provided by any FACE conformant OS. This allows software written for one conformant OS to run on systems utilizing another FACE conformant OS.

At the bottom of the example diagram, a GPS device collects sensor data and passes navigation data in a device-specific format onto a MIL-STD-1553-compliant bus. The device driver is written to the specific MIL-STD-1553 hardware, and the format of the data is specific to the GPS device.

The data from the device driver is accessed, using the common OS APIs, by a service in the IOSS to convert the unique implementation to a standardized format to abstract the uniqueness of the MIL-STD-1553 device. This abstraction allows software using the same external devices to be deployed on systems using different I/O devices.

The I/O Service then passes this data through a normalized interface defined by the FACE Technical Standard to the PSSS, in this example a GPS Platform Device Service. This device service provides an abstraction of the specific GPS communications typically described in that device's ICD, and transforms this data into a standard structure and semantics according to the FACE Data Architecture.

This data is then transported by the Transport Service Capability and Distribution Capability based on configuration to the software component that needs this data for processing, in this case an Own Ship Position PCS component. In this example, the Transport Service utilizes an OSS provided transport mechanism, specifically POSIX Sockets. All data to and from the PCS is routed through the TSS.

This Own Ship Position component calculates the graphical symbology and sends it back through the Transport Services using a well-defined graphics language. In this example, the TSS is configured to distribute these graphics messages to a Platform-Specific Graphics Service in the PSSS. The Platform-Specific Graphics Service then draws to the display though a Graphics Driver.

This scenario highlights how the FACE Reference Architecture can be used to isolate changes to a system. For example, if the GPS device is replaced with a different GPS, the associated Platform Device Service would be replaced or modified. If the MIL-STD-1553 bus is changed, then the I/O Service would be replaced or modified. If a transport mechanism is changed then the Transport Service would be replaced or modified. In all of these cases the Portable Component is isolated from these changes.

3.5 Programming Language Run-Times

The FACE Technical Standard places restrictions on programming languages. The use of standardized programming languages is fundamental to portability. The FACE Technical Standard includes requirements on the use of C, C++, Ada, and Java for the creation of conformant software.

The interface between an operating system and a software component can be substituted or enhanced by a Programming Language Run-Time. The POSIX API set is often provided in terms of a Programming Language Run-Time. For the purposes of the FACE Reference Architecture, a Programming Language Run-Time is an optional feature and can be supplied as part of the OSS, or included in a software component residing in another segment.

3.6 Component Frameworks

The FACE Technical Standard supports the use of Component Frameworks. Component Frameworks provide supporting functionality for a software component. There are three approaches to using Component Frameworks within the FACE Reference Architecture:

- Operating System Segment (OSS) Provided

  The Component Framework is provided as part of an OSS. A Component Framework provided by the OSS extends the OSS Interface to include the Component Framework's own API. Component Frameworks provided by the OSS are limited to those specified within the OSS Interface requirements and discussed further in Section 4.2.4.

- Internal to PCS or PSSS UoC

  The full Component Framework is an integral part of the PCS or PSSS UoC and adheres to the allowed PCS and PSSS interfaces. This approach leverages the ease of development of Component Frameworks and allows FACE alignment, but may have performance, integration, and/or resource impacts. This approach is discussed in Section 4.6 and Section 4.10.

- Component Framework implements FACE Reference Architecture

  A Component Framework used across multiple FACE segments implements the Framework Support Capability (FSC). The FSC is an abstraction that translates Component Framework interfaces to FACE aligned interfaces and is described in Section 4.7.14.

The OSS and TSS support common frameworks, allowing software to be developed to both a Component Framework and the FACE Technical Standard.

3.7 Operating System Segment Profiles

The FACE Reference Architecture defines three FACE OSS Profiles tailoring the Operating System (OS) APIs, programming languages, programming language features, run-times, frameworks, and graphics capabilities to meet the requirements of software components for differing levels of criticality. The three FACE OSS Profiles are:

- Security
- Safety

  — Base

  — Extended

- General Purpose

The Security Profile constrains the OS APIs to a minimal useful set allowing assessment for high-assurance security functions executing in a single address space (e.g., a POSIX process or ARINC 653 partition). The Security Profile requires ARINC 653 support.

The Safety Profile is less restrictive than the Security Profile and constrains the OS APIs to those that have a safety certification pedigree. The Safety Profile is made up of two sub-profiles. The Safety Base Sub-profile supports single POSIX process applications with a broader OS API set than the Security Profile. The Safety Extended Sub-profile includes all of the Safety Base Sub-profile OS APIs along with additional OS APIs and optional support for multiple POSIX processes. The Safety Profile requires ARINC 653 support.

Although the Security and Safety Profiles are named to reflect their primary design focus, their use is not restricted to services with those requirements (i.e., a software component without a safety or security design focus can be constrained to only use the OS APIs of one of these profiles).

The General Purpose Profile is the least constrained profile and supports OS APIs meeting real-time deterministic or non-real-time non-deterministic requirements depending on the system or

subsystem implementation. ARINC 653 support and multiple POSIX processes are optional for the General Purpose Profile.

Figure 3 shows a representation of OS APIs and restrictions per profile.

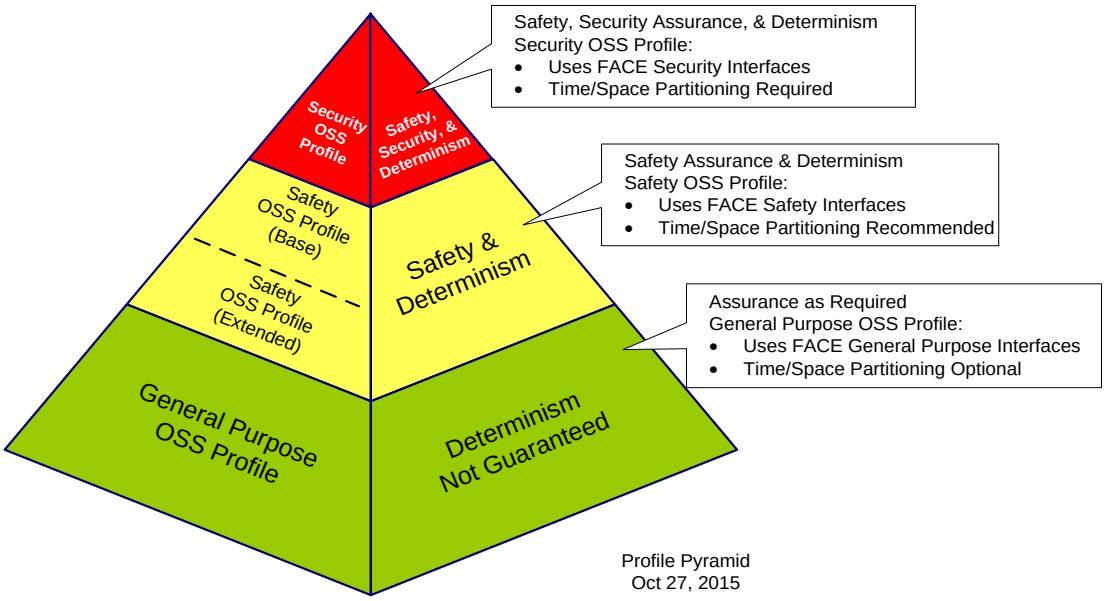

Figure 3: FACE OSS Profile Diagram

3.8 Unit of Conformance and Unit of Portability

A Unit of Conformance (UoC) is a software component or domain-specific data model designed to meet the applicable requirements defined in the FACE Technical Standard. It is referenced as a UoC at any point in its development, and becomes a FACE Certified UoC upon completion of the FACE Conformance Process. The FACE Technical Standard contains individual requirements for UoCs in each FACE Segment and for each FACE OSS Profile.

A Unit of Portability (UoP) is a UoC that resides within a PCS or PSSS.

A UoC Package can be used to include capabilities made up of multiple UoCs. UoC Packages can be developed for conformance to the FACE Technical Standard by following the requirements in Section 4.11.4.2.

3.8.1 Unit of Conformance Applicable Requirements Map

The sections defined in Table 1 are applicable when developing a PCS UoC.

Table 1: Sections Applicable to PCS UoCs

Section	Applicability
4.2.1 Operating System Interface	For operating system usage requirements
4.2.3 Programming Language Run-Time	For programming language usage requirements
4.2.5 Configuration Services	When using configuration services
4.8 Transport Services Interfaces	For communication with PSSS, TSS, and other PCS UoCs
4.9 Data Architecture	For defining data sent through the TS Interface
4.10 Portable Components Segment	Always
4.11 Unit of Conformance	Always
4.12 Graphics Services	When using graphic services
4.13 Life Cycle Management Services	When providing or using life-cycle support

The sections defined in Table 2 are applicable when developing a TSS UoC.

Table 2: Sections Applicable to TSS UoCs

Section	Applicability
4.2.1 Operating System Interface	For operating system usage requirements
4.2.3 Programming Language Run-Time	For programming language usage requirements
4.2.5 Configuration Services	When using configuration services
4.7 Transport Services Segment	Always
4.8 Transport Services Interfaces	For communication with PCS, PSSS, and TSS UoCs
4.9 Data Architecture	For defining data sent through the TS Interface
4.11 Unit of Conformance	Always
4.13 Life Cycle Management Services	When providing or using life-cycle support

The sections defined in Table 3 are applicable when developing a PSSS UoC.

Table 3: Sections Applicable to PSSS UoCs

Section	Applicability
4.2.1 Operating System Interface	For operating system usage requirements

Section	Applicability
4.2.3 Programming Language Run-Time	For programming language usage requirements
4.2.5 Configuration Services	When using configuration services
4.5 I/O Services Interface	For communication with IOSS UoCs
4.6 Platform-Specific Services Segment	Always
4.8 Transport Services Interfaces	For communication with PCS, TSS, and other PSSS UoCs
4.9 Data Architecture	For defining data sent through the TS Interface
4.11 Unit of Conformance	Always
4.12 Graphics Services	When using or providing graphic services
4.13 Life Cycle Management Services	When providing or using life-cycle support

The sections defined in Table 4 are applicable when developing an IOSS UoC.

Table 4: Sections Applicable to IOSS UoCs

Section	Applicability
4.2.1 Operating System Interface	For operating system usage requirements
4.2.3 Programming Language Run-Time	For programming language usage requirements
4.2.5 Configuration Services	When using configuration services
4.4 I/O Services Segment	Always
4.5 I/O Services Interface	For communication to PSSS UoCs
4.13 Life Cycle Management Services	When providing or using life-cycle support

The sections defined in Table 5 are applicable when developing an OSS UoC.

Table 5: Sections Applicable to OSS UoCs

Section	Applicability
4.1 Operating System Segment	Always
4.2.1 Operating System Interface	For operating system calls
4.2.2 Operating System HMFM Interface Requirements	When implementing HMFM

Section	Applicability
4.2.3 Programming Language Run-Time	Programming Language and Run-Time Requirements
4.2.4 Component Framework Interfaces	When using Component Frameworks
4.2.5 Configuration Services	When using configuration services
4.13 Life Cycle Management Services	When providing or using life-cycle support

4 FACE Reference Architecture Requirements

The FACE Reference Architecture, as depicted in Figure 4, shows the segmented approach establishing the separation of concerns necessary to accomplish the overarching FACE goals of affordability, interoperability, and time-to-field. Each segment and interface and their respective requirements are defined in further detail in the following sections.

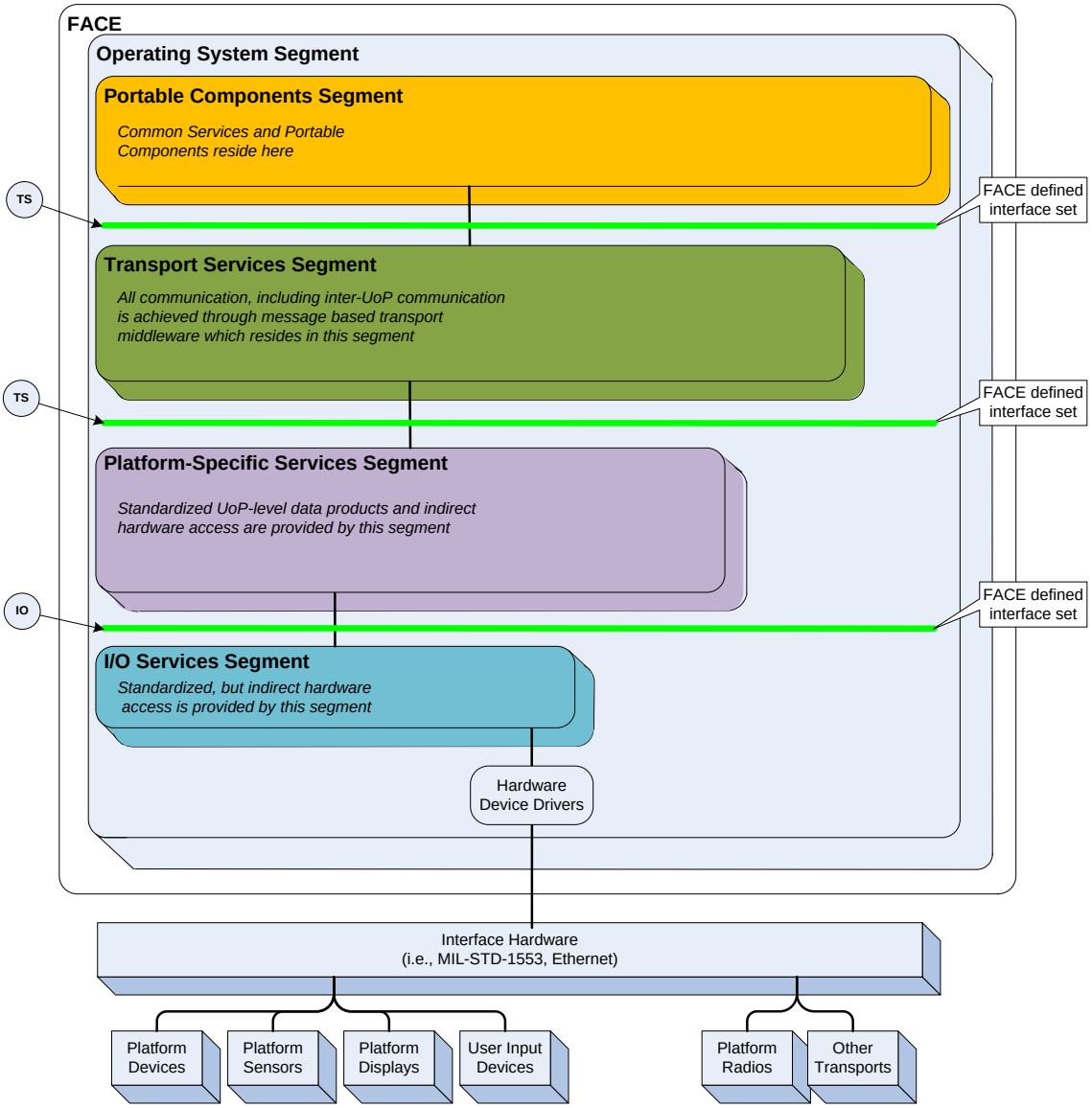

Figure 4: FACE Reference Architecture

4.1 Operating System Segment

The OSS provides and controls access to the computing platform and software environment for the other FACE segments. Access to these capabilities is provided to the other FACE segments through implementations of the OS APIs and through OSS managed configuration data. The OSS uses, as appropriate, processor control mechanisms, such as memory management units and register access controls (e.g., user *versus* kernel privileges) to restrict FACE segments to their required computing platform resources and operational capabilities. These processor control mechanisms permit various levels of independence between FACE segments, permitting greater portability across FACE aligned computing platforms. Whether an OSS capability executes with kernel or user privileges, the capability is still considered part of the OSS.

The OSS requirements are defined in Section 4.1.1. The OS API standards are defined in Section 4.2.1.

The OSS includes networking and file system capabilities adhering to published, standards-based operating system interfaces to meet basic platform requirements (Section 4.1.1). The OSS can also provide HMFM capabilities as described in Section 4.1.3.

Other capabilities the OSS provides are:

- An environment where conformant software capabilities may execute
- A set of software services accessed by managed OS API sets

4.1.1 Operating System Segment Requirements

1. An OSS UoC provides support for partition, POSIX process, ARINC 653 process/POSIX thread, and memory management functionalities.
    a. OSS interfaces for the Security OSS Profile shall support POSIX and ARINC 653 in accordance with Section 4.2.1.3.
    b. OSS interfaces for the Safety OSS Profile shall support POSIX and ARINC 653 in accordance with Section 4.2.1.4.
    c. OSS interfaces for the General Purpose OSS Profile shall support POSIX and ARINC 653 in accordance with Section 4.2.1.5.
2. An OSS UoC shall provide an operating system.
3. An OSS UoC provides support for partitioning.
    a. A General Purpose Profile OSS UoC shall provide space partitioning.

        Note: A General Purpose Profile OSS UoC may provide time partitioning.
    b. A Safety Profile OSS UoC shall provide space partitioning.

        Note: Space partitioning must be used for a UoC running in the Safety Profile. When per-partition access control is used, an OSS UoC uses configuration data to enable access control.
    c. A Safety Profile OSS UoC shall provide time partitioning.

Note: Time partitioning must be used when running Safety Profile-based software components that are dependent upon an ARINC 653 operational environment.

d. A Security Profile OSS UoC shall provide space partitioning.

e. A Security Profile OSS UoC shall provide time partitioning.

4. An OSS UoC shall use a priority preemptive scheduling algorithm across the set of POSIX threads and ARINC 653 processes associated with software components whose partitions have been assigned the same partition time window.

   Note: It is intended that when partitions are assigned to the same partition time window they are scheduled using priority preemptive scheduling.

   Note: The POSIX threads and ARINC 653 processes in these partitions may not be considered sufficiently segregated to allow qualification at different levels of criticality when allocated identical time windows.

   Note: In multicore environments, this requirement is not intended to require support for partitions concurrently executing on different cores.

5. When an OSS UoC provides HMFM support, the UoC shall do so in accordance with Section 4.1.3.

6. When a Programming Language Run-Time is provided by an OSS UoC, the Run-Time shall be in accordance with Section 4.2.3.

7. When a Component Framework is provided by an OSS UoC, the Framework shall be in accordance with Section 4.2.4.

8. When Configuration Services are provided by an OSS UoC, the Services shall be in accordance with Section 4.2.5.

4.1.1.1 OSS Configuration Requirements

An OSS UoC provides support for configuration of OS characteristics.

1. An OSS UoC provides support for ARINC 653-defined configuration data types (XML-based schema) for allocating computing platform resources:

   a. When a General Purpose Profile OSS UoC supports ARINC 653, the UoC shall support ARINC 653-defined configuration data types (XML-based schema) for allocating computing platform resources.

      Note: POSIX does not define types or methods for specifying configuration.

   b. A Safety Profile OSS UoC shall support ARINC 653-defined configuration data types (XML-based schema) for allocating computing platform resources.

   c. A Security Profile OSS UoC shall support ARINC 653 -defined configuration data types (XML-based schema) for allocating computing platform resources.

2. An OSS UoC provides support for configuration of OS-level HMFM.

   a. When a General Purpose Profile OSS UoC supports ARINC 653, the UoC shall support configuration of OS-level HMFM.

      Note: POSIX does not define methods for specifying configuration.

b. A Safety Profile OSS UoC shall support configuration of OS-level HMFM.

c. A Security Profile OSS UoC shall support configuration of OS-level HMFM.

3. An OSS UoC shall support configuration of memory regions and access to those regions.

4. An OSS UoC shall support configuration of device driver access.

5. An OSS UoC shall support configuration of POSIX named semaphore access.

6. An OSS UoC shall support configuration of partitions in accordance with the defined FACE OSS Profiles.

   Note: Support for ARINC 653-defined configuration data types related to multicore is not required.

7. An OSS UoC shall support configuration of partition time windows in accordance with the defined FACE OSS Profiles.

8. An OSS UoC shall support configuration of inter-partition communications (i.e., between partitions) in accordance with the defined FACE OSS Profiles.

9. An OSS UoC shall support configuration of permission to set calendar time visible to all partitions.

10. Note: POSIX does not define methods for specifying configuration.

11. An OSS UoC provides support for network communications configuration.

    a. An OSS UoC shall support the control over which partitions are authorized to bind/connect to a network communication endpoint.

    b. An OSS UoC shall support the control over which partitions are authorized to receive from a network communication endpoint.

    c. An OSS UoC shall support the control over which partitions are authorized to transmit to a network communication endpoint.

4.1.1.2 OSS File System Requirements

An OSS UoC may provide file system support.

1. When an OSS UoC provides file system support, then:

    a. A General Purpose Profile OSS UoC shall support configuration of a file system.

    b. A General Purpose Profile OSS UoC shall support data storing of buffered data.

    c. A General Purpose Profile OSS UoC shall support data flushing of buffered data.

    d. A Safety Profile OSS UoC shall support configuration of a file system.

    e. A Safety Profile OSS UoC shall support data storing of buffered data.

    f. A Safety Profile OSS UoC shall support data flushing of buffered data.

    Note: File systems are not supported by the Security Profile.

2. When an OSS UoC provides support for multiple file system storage elements, then:

a. A General Purpose Profile OSS UoC shall support files storage.

b. A General Purpose Profile OSS UoC shall support directories storage.

c. A General Purpose Profile OSS UoC shall support volumes storage.

d. A Safety Profile OSS UoC shall support files storage.

e. A Safety Profile OSS UoC shall support directories storage.

f. A Safety Profile OSS UoC shall support volumes storage.

Note: There may be multiple file system types available in the implementation to different hardware devices with different attributes for access rights, support for time and space partitioning, authentication, and integrity.

Note: File systems are not supported by the Security Profile.

3. An OSS UoC supports the ability to allocate read and write access permissions of volumes to specific partitions (i.e., no default access permissions) when an OSS UoC provides file system support.

a. A General Purpose Profile OSS UoC shall support individual configuration of read and write access permission settings for each partition to each volume.

b. When supporting multiple POSIX processes, a General Purpose Profile OSS UoC shall support individual configuration of execute access permission settings for each partition to each volume.

c. A Safety Profile OSS UoC shall support configuration of read-access permission settings for each partition to each volume.

d. A Safety Profile OSS UoC shall support configuration of permission settings such that at most one partition has write access to each volume.

Note: Restriction to at most one partition is per ARINC 653 Part 2.

e. When supporting multiple POSIX processes, a Safety Extended Sub-profile OSS UoC shall support configuration of execute access permission settings for each partition to each volume.

Note: Execute permissions are in support of the creation of multiple POSIX processes which is not required in the Safety Base Sub-profile and is optional in the Safety Extended Sub-profile.

Note: File systems are not supported by the Security Profile.

4. When an OSS UoC provides support for portable data media, then:

a. A General Purpose Profile OSS UoC shall support mounting data media and/or insertion/mounting of portable data media.

b. A Safety Profile OSS UoC shall support mounting data media and/or insertion/mounting of portable data media.

Note: File systems are not supported by the Security Profile.

5. When an OSS UoC provides support for atomic file updates, then:

    a. A General Purpose Profile OSS UoC shall support atomically updated data blocks (i.e., the data block is ensured to be completely saved in non-volatile storage or, if interrupted, none of the data block is saved).

    b. A Safety Profile OSS UoC shall support atomically updated data blocks (i.e., the data block is ensured to be completely saved in non-volatile storage or, if interrupted, none of the data block is saved).

    Note: This capability ensures consistency of the stored blocks of data.

    Note: File systems are not supported by the Security Profile.

4.1.1.3 OSS POSIX Clock Requirements

An OSS UoC provides support for POSIX clocks.

1. An OSS UoC shall support the POSIX Monotonic Clock option (CLOCK_MONOTONIC), including setting absolute and relative timers on this clock.

2. An OSS UoC shall support clock read-access based POSIX CLOCK_REALTIME clock (calendar-based clock time).

3. An OSS UoC shall support setting a relative timer-based POSIX CLOCK_REALTIME clock (calendar-based clock time).

4. An OSS UoC shall support clock time write access for POSIX CLOCK_REALTIME clock value under the constraint that configuration data specifies the partition is authorized to set the clock time (using the *clock_settime*() API).

    Note: The time value set by one partition becomes visible to all other partitions. The timezone value set by one POSIX process using the *tzset*() or *setenv*() APIs is local to that POSIX process.

5. An OSS UoC shall return an error when a software component attempts to set an absolute timer on the POSIX CLOCK_REALTIME clock.

    Note: The behavior is necessary (e.g., to prevent backwards and forwards shifts in sequential timers) due to the setting of POSIX_CLOCK_REALTIME clock impacting all partitions.

4.1.1.4 OSS Networking Requirements

An OSS UoC may provide support for an IP-based network stack.

1. When an OSS UoC provides an IP-based network stack, then:

    a. A General Purpose Profile OSS UoC shall provide support for an IP-based network stack as defined by the IETF Requests for Comments (RFCs) listed in Table 38 and Table 39 shown in Section A.5.

    b. A Safety Extended Sub-profile OSS UoC shall provide support for an IP-based network stack as defined by the RFCs listed in Table 38 and Table 39 shown in Section A.5.

c. A Safety Base Sub-profile OSS UoC shall provide support for an IP-based network stack as defined by the RFCs listed in Table 38 shown in Section A.5.

d. A Security Profile OSS UoC shall provide support for an IP-based network stack as defined by the RFCs listed in Table 38 shown in Section A.5.

Note: There are RFCs applicable to security. Requirements for those RFCs are outside the scope of this standard.

2. When providing IPv6 networking capabilities, an OSS UoC shall provide the networking capabilities per the RFCs referenced in Table 40 shown in Section A.5.

3. When providing an integrated IPv4/IPv6 network transition mechanism, an OSS UoC shall provide the network transition mechanism per the RFCs referenced in Table 41 shown in Section A.5.

4.1.2 OSS UoC Life Cycle Management Services Interface Requirements

An OSS UoC is not required to provide an LCM Services Interface.

4.1.3 OSS Health Monitoring and Fault Management

The purpose of the FACE OSS Health Monitoring and Fault Management (HMFM) software component is to provide standardized methods for detecting, reporting, and handling faults and failures within the scope of a single system or platform. The basic goals of OSS HMFM include:

- Detecting and handling faults and errors that occur during run-time
- Detecting and handling faults and errors at the ARINC 653 process (i.e., POSIX thread), partition, and module (i.e., platform) levels
- Providing the flexibility to enable system designers to control the appropriate level of HMFM capabilities given the desired design objectives

Faults and errors can arise from any software component, including the OSS and the OSS HMFM.

4.1.3.1 HMFM Introduction

Figure 5 depicts a fault management cycle state machine that provides a framework for an overall HMFM capability.

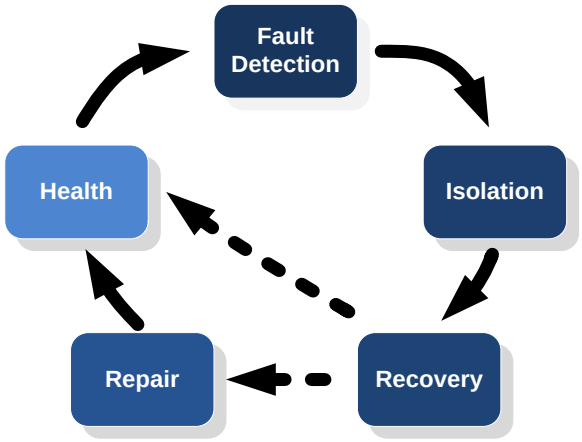

Figure 5: Fault Management Cycle State Machine

The overall HMFM goal is to eliminate service outages through an HMFM system that is aware of the occurrence of faults and errors that occur during run-time. Upon detection of a fault or error, the HMFM cycle is engaged to direct the system back to a healthy state or to some fail-safe state. This direction is controlled by pre-configured Fault Management (FM) policies. Given a start state of a healthy system, management of a real fault (i.e., not a false alarm) can cycle through the FM state machine diagram as follows:

- Fault detection state – a fault or error is detected and declared by HMFM when some defined fault criteria have been met and thus promoted to "fault" status; such faults and errors can originate from any segment within the scope of a system
- Fault isolation state – HMFM determines the scope of a fault or error given the particular source of the fault or error
- Fault recovery state – HMFM enforces a fault recovery policy; this policy can result in:
  - — Full re-establishment of system health resulting in continued service availability
  - — Partial re-establishment of system health (e.g., shutdown a partition but permit other partitions to continue to execute)
  - — Transition to a fail-safe state
  - — Attempt a repair action (when supported)
- Fault repair state – when supported and configured, HMFM restores (e.g., performs a hot-swap) a capability that was impacted by a fault back to a healthy state and then re-integrates the capability back into the system
- Health – the operational state of a system with functional HMFM

Parts or all of the HMFM fault management cycle capabilities may be relevant to a particular design.

4.1.3.2 OSS HMFM Requirements

An OSS UoC supports HMFM capability requirements that include scope, functional capabilities, and configuration properties.

1. An OSS UoC supports HMFM capabilities based on the OSS Profiles.

    a. A Security Profile OSS UoC shall provide the HMFM capability in accordance with requirements in Section 4.2.1.3.

    b. A Safety Profile OSS UoC shall provide the HMFM capability in accordance with requirements in Section 4.2.1.4.

    c. When a General Purpose Profile OSS UoC provides HMFM capabilities, the UoC shall provide the HMFM capability according to the requirements in Section 4.2.1.5.

2. An OSS UoC includes Health Monitor (HM) capabilities at the module level (i.e., computing platform level), partition-level, and thread-level.

    a. Module-level HM capabilities shall include the ability to idle the module.

    b. Module-level HM capabilities shall include the ability to restart the module.

    c. Partition-level HM capabilities shall include the ability to idle the partition.

    d. Partition-level HM capabilities shall include the ability to restart the partition.

    Note: These capabilities are controlled using configuration data managed directly by an OSS UoC.

    Note: Thread-level HM runs in the context of a partition-specific error handler that is included as part of the software component. Thread-level HM can invoke the OS APIs available in this context.

3. An OSS UoC supports HMFM types and services that can be used by other UoCs.

    a. The HMFM capability shall implement ARINC 653 Part 1 Health Monitoring Types and Health Monitoring Services capabilities for use by FACE ARINC 653-based UoCs.

    b. The HMFM capability shall implement FACE HMFM Types and APIs defined in Section 4.2.2 and Appendix F for use by FACE POSIX based UoCs.

4. The HMFM capability shall be initialized at system start-up.

5. The HMFM capability shall provide monitoring to detect faults/errors.

6. The HMFM capability shall execute response and recovery actions for detected faults/errors as configured.

4.2 Operating System Segment Interface

The OSS Interface provides a standardized means for software to use the services within the operating system and other capabilities related to the OSS. This interface is provided by software in the OSS to software in other segments.

Section 4.2.1 describes the POSIX and ARINC 653 make-up of the OSS Interface for each profile. The profiles are separated into General Purpose, Safety, and Security. The Safety Profile contains two Sub-profiles referred to as Safety Base and Safety Extended. Section 4.2.2 describes the Interface to the Health Monitoring and Fault Management (HMFM) capability for

both POSIX and ARINC 653-based implementations. Section 4.2.3 describes constraints on programming language features and run-times that are deployed as part of an OSS UoC. Section 4.2.4 describes constraints on Component Frameworks that are deployed as part of an OSS UoC. Section 4.2.5 describes the Configuration Services which are available for UoCs to utilize. Programming Language Run-Times, Component Frameworks, and Configuration Services that do not use the FACE Standardized Interfaces can be used in FACE conformant solutions only when included as part of an OSS UoC. Figure 6 shows multiple Run-Times and Component Frameworks that use non-standardized FACE Interfaces on the "Bottom side" while still providing aligned FACE Interfaces.

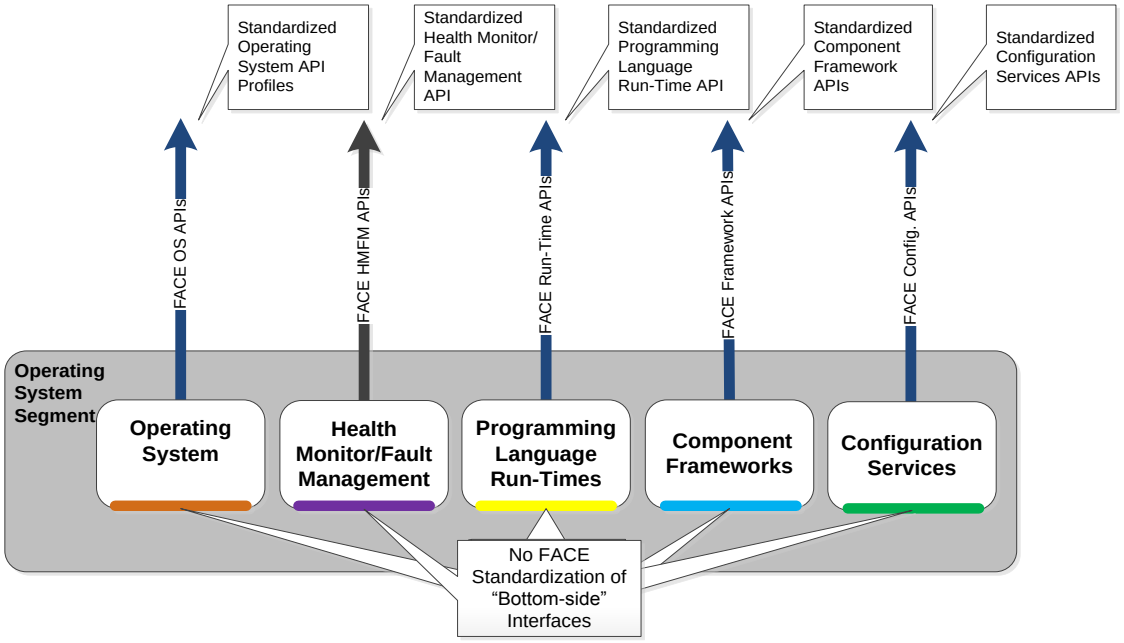

Figure 6: Operating System Segment Interfaces

The software resources (e.g., bottom-side interfaces) used by Operating Systems, Programming Language Run-Times, Life Cycle, Component Frameworks, and Configuration Services are expected to vary and are not prescribed or otherwise controlled by the FACE Technical Standard. As an example, Operating Systems are often fielded on computing hardware with different Board Support Packages (BSPs). Operating Systems may use device drivers to abstract differences between hardware devices.

The number of Programming Language Run-Times and Component Frameworks is not prescribed by the FACE Technical Standard. The use of Programming Language Run-Times and/or Component Frameworks is optional. It may be necessary to have distinct instances of Programming Language Run-Times and/or Component Frameworks in different partitions to support safety and security.

Operating Systems, Programming Language Run-Times, and Component Frameworks provide an environment for the collaborative execution of software components, as depicted in Figure 6.

4.2.1 Operating System Interface

OSS UoCs provide POSIX and ARINC 653-based operating environments and API subsets based on the following profile requirements.

FACE OS APIs correspond to the General Purpose, Safety, and Security Profiles.

The POSIX and ARINC 653 standards include Ada and C language bindings for the OS APIs included in the OSS Profiles. The C language header files for the POSIX and ARINC 653 APIs may be compatible for direct use by C++ based software components. Bindings for Java may be provided by other means such as by a compiler vendor or the development of a standard that defines a binding. Appendix A details the POSIX API required for each OSS Profile, including the identification of the multi-process POSIX APIs.

Partitions and POSIX processes execute software components designed for one of the FACE OSS Profiles.

Partitions may use a different operational environment (POSIX or ARINC 653). Each partition uses either a POSIX or an ARINC 653 operational environment.

A computing platform may simultaneously support partitions of different OSS Profiles and operational environments. The FACE Technical Standard recognizes the APIs associated with the General Purpose, Safety, and Security Profiles provide divergent capabilities. The divergence of these capabilities may prevent or represent considerable safety or security risks, including simultaneous hosting of General Purpose, Safety, and Security Profile OSS UoCs on the same computing platform. When system requirements include software components of different OSS Profiles, the use of multiple computing platforms may be required. Other architectural capabilities (e.g., virtualization), if supported by the processor and OS selected by the system integrator, may provide other means to simultaneously host on the same computing platform software components which are using different FACE OSS Profiles.

Based on the profile and operating environment, OSS UoCs provide support for ARINC 653 sampling ports, ARINC 653 queuing ports, and POSIX sockets to be used for inter-partition communications (i.e., the lower-layer transport mechanism that can be used by other FACE standardized interface software libraries).

A UoC designed to the General Purpose Profile, when communicating with UoCs designed to the Security and/or Safety Profile, uses the same restricted inter-partition communications as a UoC designed to the Security and/or Safety Profile. A UoC designed to the General Purpose Profile, if communicating to another UoC designed to the General Purpose Profile, may use additional sockets services (if supported by the transport mechanism).

4.2.1.1 Operating System Segment Profiles

This section summarizes the OSS Interface characteristics for the General Purpose, Safety, and Security Profiles. These characteristics are based on support of APIs and OS capabilities defined in the ARINC 653 and POSIX standards.

The referenced ARINC 653 standards for all OSS Profiles include API and OS support for utilizing multiple processor cores to concurrently schedule multiple ARINC 653 processes within a partition based on an ARINC 653 operating environment. When a computing platform basis includes multiple equivalent processing cores, the supported APIs provide a means to control how ARINC 653 processes are concurrently scheduled on the processor cores allocated to the partition.

The OS configuration data includes a means to control the number of cores available to each partition. A partition allocated a single processor core of a multicore processor has the same

ARINC 653 scheduling characteristics the partition would have when running on a single-core processor.

4.2.1.2 Operating System Interface Profile Requirements

For General Purpose and Safety Extended POSIX implementations:

1. An OSS UoC implementing POSIX shall support via *setsockopt*() the configuration of:
    a. A TCP socket such that small packets are not delayed (TCP_NODELAY)
2. An OSS UoC shall provide the following configuration information via *getsockopt*() for the specified socket:
    a. The TCP small packet delay characteristics for a socket (TCP_NODELAY)

For General Purpose, Safety, and Security Profile ARINC 653 implementations:

1. An OSS UoC implementing ARINC 653 shall provide messages to UoCs in the order they are received.
2. An OSS UoC implementing ARINC 653 shall support delivery of available messages.
3. An OSS UoC implementing ARINC 653 shall support receipt of complete messages.

    Note: No silently truncated messages.
4. An OSS UoC implementing ARINC 653 shall support sending messages on an ARINC 653 port and receiving the same messages in another partition using an ARINC 653 port.

For General Purpose, Safety, and Security Profile POSIX implementations:

1. An OSS UoC implementing POSIX shall provide messages to UoCs in the order they are received.
2. An OSS UoC implementing POSIX shall support delivery of available messages.
3. An OSS UoC implementing POSIX shall support receipt of complete messages.

    Note: No silently truncated messages.
4. An OSS UoC implementing POSIX shall support sending messages on POSIX sockets and receiving the same messages in another partition using POSIX sockets.
5. An OSS UoC implementing POSIX shall support blocking on an empty socket (receive).
6. An OSS UoC implementing POSIX shall support blocking on a full socket (transmit).
7. An OSS UoC implementing POSIX shall support via *setsockopt*() the configuration of:
    b. The size of the space reserved for a socket's receive buffer (e.g., SO_RCVBUF)
    c. The size of the space reserved for a socket's transmit buffer (e.g., SO_SNDBUF)
    d. The multicast characteristics for a socket

        Note: Multicast characteristics that can be configured on a per socket per partition basis are: IP_MULTICAST_IF, IP_MULTICAST_TTL,

IP_MULTICAST_LOOP, IP_ADD_MEMBERSHIP, IP_DROP_MEMBERSHIP, SO_REUSEADDR.

Note: The *ip_mreq* structure is supported to provide a means to add and drop multicast group memberships.

8. An OSS UoC shall limit based on configuration:
    a. Socket maximum receive message size
    b. Socket maximum transmit message size
9. An OSS UoC shall provide the following configuration information via *getsockopt*() for the specified socket:
    a. The size of the space reserved for a socket's receive buffer (e.g., SO_RCVBUF)
    b. The size of the space reserved for a socket's transmit buffer (e.g., SO_SNDBUF)
    c. The multicast characteristics for a socket

    Note: A socket may have multiple multicast group memberships set for it. The *getsockopt*() API does not provide a means to retrieve a socket's multicast group memberships. All other characteristics can be obtained.
10. An OSS UoC shall define:
    a. A macro in <sys/ioctl.h> named FIONBIO to configure sockets for non-blocking I/O
    b. A macro in <devctl.h> named SOCKCLOSE to close an open socket
11. An OSS UoC shall provide:
    a. The FIONBIO command in the *posix_devctl*() API defined in Section A.1
    b. The SOCKCLOSE command in the *posix_devctl*() API defined in Section A.1

4.2.1.3 Security Profile API Requirements

1. A Security Profile OSS UoC shall support ARINC 653 and POSIX.
2. A Security Profile OSS UoC shall provide the following ARINC 653 APIs for use in an ARINC 653 operational environment:
    a. ARINC 653 Part 1 – all services associated with Avionics Application Software Standard Interface Part 1 – Required Services
        i. For platforms that support multicore partitions, all services from ARINC 653 Part 1-4 (Part 1, Supplement 4, August 2015)
        ii. For platforms that support only single core partitions, all services from ARINC 653 Part 1-3 (Part 1, Supplement 3, November 2010) or ARINC 653 Part 1-4 (Part 1, Supplement 4, August 2015)
    b. Services associated with the following categories of Avionics Application Software Standard Interface Part 2 – Extended Services:
        i. Memory Blocks

Note: All other Part 2 services are intentionally excluded from the Security Profile.

Note: OSS UoCs are permitted to include additional ARINC 653 APIs outside of the profile. Software components are restricted to using only the APIs defined as being part of the profile.

3. A Security Profile OSS UoC shall provide the POSIX APIs defined in Section A.1 and Section A.3 for the Security Profile for use in a POSIX operational environment.

4. A Security Profile OSS UoC for a POSIX operational environment shall support mutex operations (e.g., *pthread_mutexattr_setprotocol*() API) with priority protect protocol (_POSIX_THREAD_PRIO_PROTECT).

5. A Security Profile OSS UoC for a POSIX operational environment shall support memory mapping operations (e.g., *mmap*() API) with shared memory objects (_POSIX_SHARED_MEMORY_OBJECTS).

6. A Security Profile OSS UoC for a POSIX operational environment shall include support for use of ARINC 653 sampling and queueing ports for inter-partition communications.

7. A Security Profile OSS UoC for a POSIX operational environment shall include support for the Health Monitoring APIs defined in Section 4.2.2 to communicate with an ARINC 653 Health Monitor.

4.2.1.4 Safety Profile API Requirements

1. A Safety Profile OSS UoC shall support ARINC 653 and POSIX.

2. A Safety Profile OSS UoC shall provide the following ARINC 653 APIs for use in an ARINC 653 operational environment:

    a. ARINC 653 Part 1 – all services associated with Avionics Application Software Standard Interface Part 1 – Required Services:

        ii. For platforms that support multicore partitions, all services from ARINC 653 Part 1-4 (Part 1, Supplement 4, August 2015)

        iii. For platforms that support only single core partitions, all services from ARINC 653 Part 1-3 (Part 1, Supplement 3, November 2010) or ARINC 653 Part 1-4 (Part 1, Supplement 4, August 2015)

    b. Services associated with the following categories of Avionics Application Software Standard Interface Part 2 – Extended Services:

        i. File System

        ii. Sampling Port Extensions

        iii. Memory Blocks

        iv. Multiple Module Schedules

    Note: All other Part 2 services, including Logbooks and Multiple Processor Core Extensions, are intentionally excluded. File system services can be used instead of Logbooks.

Note: FACE OSS UoCs are permitted to include additional ARINC 653 APIs outside of the profile. Software components are restricted to using only the APIs defined as being part of the profile.

3. A Safety Profile OSS UoC for a POSIX operational environment shall support mutex operations (e.g., *pthread_mutexattr_setprotocol*() API) with priority protect protocol (_POSIX_THREAD_PRIO_PROTECT).

4. A Safety Profile OSS UoC for a POSIX operational environment shall support memory mapping operations (e.g., *mmap*() API) with shared memory objects (_POSIX_SHARED_MEMORY_OBJECTS).

5. A Safety Profile OSS UoC for a POSIX operational environment shall include support for use of ARINC 653 sampling and queueing ports for inter-partition communications.

6. A Safety Profile OSS UoC for a POSIX operational environment shall include support for the Health Monitoring APIs defined in Section 4.2.2 to communicate with an ARINC 653 Health Monitor.

4.2.1.4.1 Safety Base Sub-Profile API Requirements

1. A Safety Base Sub-profile OSS UoC shall provide the POSIX APIs defined in Section A.1 and Section A.3 for the Safety Base Sub-profile.

2. A Safety Base Sub-profile OSS UoC shall provide the *FD_CLR*(), *FD_ISSET*(), *FD_SET*(), *FD_ZERO*(), and *select*() APIs for use with sockets.

3. A Safety Base Sub-profile OSS UoC shall provide message queue support within a partition only (i.e., if not an inter-partition communications mechanism).

4.2.1.4.2 Safety Extended Sub-Profile API Requirements

1. A Safety Extended OSS UoC implementation shall provide the POSIX APIs defined in Section A.1 marked as “INCL” and Section A.3 for the Safety Extended Sub-profile.

   Note: The Safety Extended Sub-profile includes all of the Safety Base Sub-profile OS POSIX APIs.

2. When supporting multiple POSIX processes, a Safety Extended OSS UoC implementation shall provide the multi-process POSIX APIs defined in Section A.1 marked as “MP” for the Safety Extended Sub-profile.

3. A Safety Extended Sub-profile OSS UoC shall provide *FD_CLR*(), *FD_ISSET*(), *FD_SET*(), *FD_ZERO*(), and *select*() for use with sockets.

4.2.1.5 General Purpose Profile API Requirements

1. A General Purpose Profile OSS UoC shall provide the POSIX APIs defined in Section A.1 marked as “INCL” and Section A.3 for the General Purpose Profile.

2. When supporting multiple POSIX processes, a General Purpose Profile OSS UoC implementation shall provide the multi-process POSIX APIs defined in Section A.1 marked as “MP” for the General Purpose Profile.

3. A General Purpose Profile OSS UoC shall provide *FD_CLR*(), *FD_ISSET*(), *FD_SET*(), *FD_ZERO*(), and *select*() for use with sockets.

4. When providing ARINC 653, a General Purpose Profile OSS UoC shall provide the following ARINC 653 APIs:

    a. All services associated with Avionics Application Software Standard Interface Part 1 – Required Services:

        i. For platforms that support multicore partitions, all services from ARINC 653 Part 1-4 (Part 1, Supplement 4, August 2015)

        ii. For platforms that support only single core partitions, all services from ARINC 653 Part 1-3 (Part 1, Supplement 3, November 2010) or ARINC 653 Part 1-4 (Part 1, Supplement 4, August 2015)

    b. Services associated with Avionics Application Software Standard Interface Part 2 – Extended Services:

        i. File System

        ii. Sampling Port Extensions

        iii. Memory Blocks

        iv. Multiple Module Schedules

        Note: All other Part 2 services, including Logbooks and Multiple Processor Core Extensions, are intentionally excluded. File system services can be used instead of Logbooks.

5. When supporting ARINC 653, a General Purpose Profile OSS UoC shall include support for use of ARINC 653 sampling and queueing ports for inter-partition communication.

6. When an ARINC 653 Health Monitor is used, a General Purpose Profile OSS UoC for a POSIX operational environment shall include support for the Health Monitoring APIs defined in Section 4.2.2 to communicate with an ARINC 653 Health Monitor.

4.2.2 Operating System HMFM Interface Requirements

This section contains the interface requirements for an OSS UoC HMFM functions used to support POSIX operational environments.

ARINC 653 operational environments include services for OSS UoC HMFM as part of the required API and operational support. As such, the requirements in the following subsections are applicable only to POSIX operational environments.

For the General Purpose Profile, support for HMFM interfaces is optional. Support for POSIX HMFM can depend upon support for ARINC 653 operational environments.

The scope of the POSIX HMFM interfaces is the partition (i.e., POSIX process) boundary.

4.2.2.1 Initialize(HMFM) Function Requirements

1. The *Initialize*(HMFM) function shall initialize the HMFM POSIX implementation for the current POSIX process.

2. The *Initialize*(HMFM) function shall return one of the return codes as specified in Section F.2.1.

4.2.2.2 Report_Application_Message(HMFM) Function Requirements

1. The *Report_Application_Message*(HMFM) function shall send a message to the HM function.

   Note: Means to retrieve messages sent to the HM function are OS-specific.

2. The *Report_Application_Message*(HMFM) function shall return one of the return codes specified in Section F.2.2.

4.2.2.3 Create_Fault_Handler(HMFM) Function Requirements

1. The *Create_Fault_Handler*(HMFM) function shall register a partition-specific fault handler invoked in the event of thread-level faults detected by the OS.

2. The *Create_Fault_Handler*(HMFM) function shall register a partition-specific fault handler invoked in the event of thread-level faults raised by the software component.

3. The *Create_Fault_Handler*(HMFM) function shall return one of the return codes as specified in Section F.2.3.

4. The *Create_Fault_Handler*(HMFM) function shall activate partition-specific fault handling when the function call is successful.

   Note: POSIX operational environments do not include a concept of initialization and operational phases present in an ARINC 653 operational environment. As such, POSIX process-level fault handling becomes active (i.e., available to be scheduled) as part of a successful function call to create it.

4.2.2.4 Get_Fault_Status(HMFM) Function Requirements

1. The *Get_Fault_Status*(HMFM) function shall return status information related to a detected fault that has been configured to be handled as a thread-level fault.

2. The *Get_Fault_Status*(HMFM) function shall return one of the return codes specified in Section F.2.4.

3. The fault status referenced by the fault parameter shall contain information regarding the fault when the function call is successful.

4. The POSIX thread ID shall be reported for ARINC 653 Part 1 FAILED_PROCESS_ID.

4.2.2.5 Raise_Application_Fault(HMFM) Function Requirements

1. The *Raise_Application_Fault*(HMFM) function shall raise the reported application fault to the thread-level fault handler when a thread-level error handler has been defined and the interface was not invoked by the thread-level error handler.

2. The *Raise_Application_Fault*(HMFM) function shall raise the reported application fault to partition HM when a thread-level error handler has not been defined or the interface was invoked by the thread-level error handler.

3. The *Raise_Application_Fault*(HMFM) function shall return one of the return codes specified in Section F.2.5.

4. The *Raise_Application_Fault*(HMFM) function shall cause the thread-level fault handler to be scheduled when the function call is successful, the thread-level error handler was defined, and the function interface was not invoked by the thread-level error handler.

4.2.3 Programming Language Run-Time

4.2.3.1 Programming Language Run-Time Description

A UoC can use a run-time provided by the OSS following the requirements in this section. A UoC can also use a non-standard run-time if that run-time is included in the UoC and follows all requirements for the segment in which the UoC is implemented. This concept also applies to frameworks, as depicted in Figure 7.

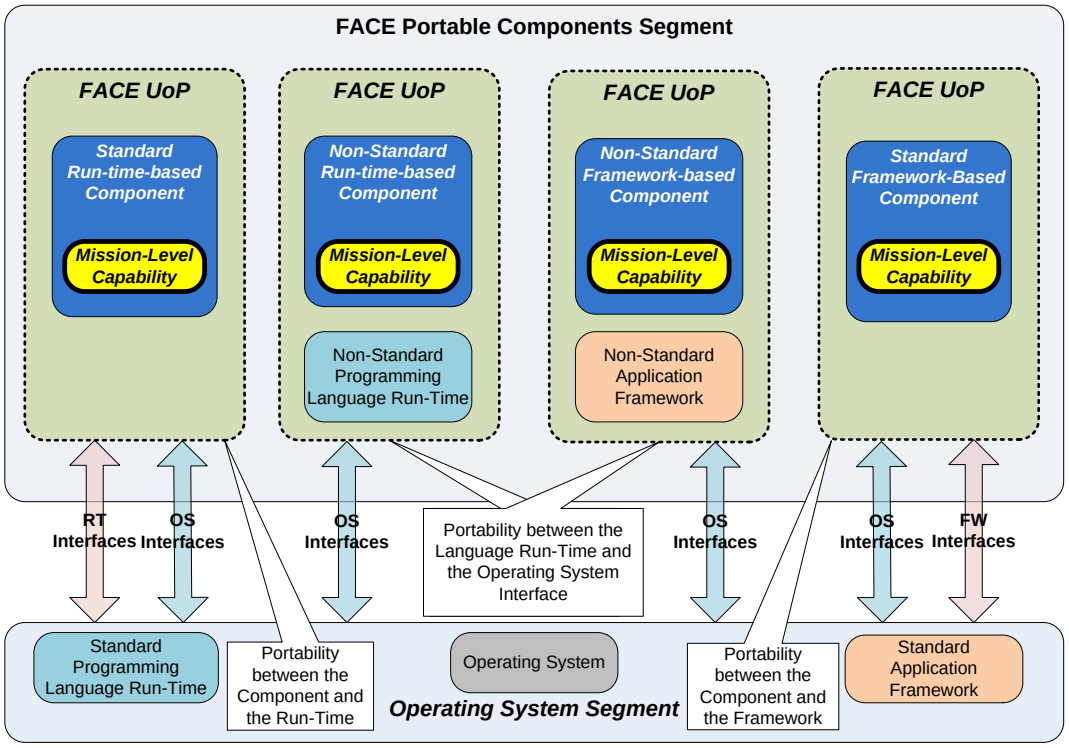

Figure 7: Portability Distinctions

The following subsections define Programming Language capability sets that are considered part of the OSS. The defined Programming Language capability sets include features that typically do not require Programming Language Run-Time support and features that typically require Programming Language Run-Time support. To account for potential differences between compiler/Programming Language Run-Time implementations, minimal discussion is included as to actual Programming Language Run-Time dependencies. OSS UoC support for Programming Language capability sets is optional.

Component use of compiler-specific language extensions is prohibited unless the entire Programming Language Run-Time utilizes only function calls defined in the corresponding FACE OSS Profile.

Note: There are some overlaps in library functions between C++, C, and the POSIX standards. The C++ library capabilities consist of library functions that are unique to

C++ and a set of functions that are in common with the C library (to permit components developed in C to be ported to C++). The POSIX standard includes the same functions as the C library specification, but does not include the additional library functions unique to C++. The number of C library functions supported varies by FACE OSS Profile. As such, for the library functions that are in common between C++ and C, only the functions from the corresponding FACE OSS Profile are supported.

4.2.3.2 C Programming Language

The C Programming Language can be supported on all OSS Profiles.

Component developers may apply their own safety-related restrictions, such as programming restrictions defined by the Motor Industry Software Reliability Association (MISRA C:2004 Guidelines for the Use of the C language in Critical Systems).

4.2.3.2.1 C Programming Language Definition

The language features supported in the C Programming Language adhere to ANSI/ISO/IEC 9899:1999: Programming Languages – C with the following modifications:

- The optional compiler support for the exact-width types in <stdint.h> is included (ANSI/ISO/IEC 9899:1999, §7.18.1.1)
- Component use of the pragma directive (ANSI/ISO/IEC 9899:1999, §6.10.6) for data structure compositions on FACE Interfaces is excluded

  Note: All other uses of pragma directives are permitted.

  Note: Support for pragma directives is compiler implementation-dependent. A compiler ignores pragma directives it does not recognize.
- Component use of wide characters (ANSI/ISO/IEC 9899:1999, §3.7.3), multibyte characters (ANSI/ISO/IEC 9899:1999, §3.7.2), wide strings (ANSI/ISO/IEC 9899:1999, §7.1.4), and multibyte strings (ANSI/ISO/IEC 9899:1999, §7.1.1) is excluded, including library functions that manipulate those types

The C library functions include those defined in ANSI/ISO/IEC 9899:1999, §7 supporting the selected FACE OSS Profile and the Section A.1 FACE OSS Profile APIs whose POSIX Functionality Categories are defined as POSIX_C_LANG_SUPPORT_R.

Note: Use of compiler-specific Programming Language extensions (i.e., not defined in the above standard) is prohibited.

4.2.3.2.2 C Programming Language and Run-Time Requirements

The requirements associated with this language include:

1. UoCs using the C Programming Language Run-Time supplied by the OS shall be restricted to the Programming Language capability set features listed in Section 4.2.3.2.1.

   Note: An OSS UoC must provide the functionality as defined, but may support excluded and/or additional features.

2. UoCs using the C Programming Language Run-Time supplied by the OS shall be restricted to the Programming Language capability set library functions listed in Section 4.2.3.2.1.

Note: An OSS UoC must provide the functionality as defined, but may support excluded and/or additional features.

3. OSS UoCs providing a C Programming Language Run-Time shall support the capability set defined in Section 4.2.3.2.1.

   Note: An OSS UoC must provide the functionality as defined, but may support excluded and/or additional features.

4.2.3.3 C++ Programming Language

The following sections define C++ Programming Language features and library functions for the supported capability sets. These capability sets are listed in the order of most permissive (General Purpose) to most restrictive (Security). As with the OSS Profiles, the restrictions are established to allow deployment of a UoC in an environment developed to any of the more permissive capability sets. Note that the deployment of a UoC into more permissive capability sets may not be adequate for the required design assurance of the UoC.

Additional implementation-specific restrictions may be imposed on software component development that are outside the C++ restrictions imposed in this section (e.g., MISRA Guidelines for the Use of the C++ language in Critical Systems).

4.2.3.3.1 C++ Programming Language Definition for General Purpose Capability Set

The General Purpose C++ Programming Language capability set includes features from the Programming Language specification defined in ISO/IEC 14882:2003: Programming Languages – C++ with the following modifications:

- Component use of the pragma directive (ISO/IEC 14882:2003, §16.6) for data structure compositions on FACE Interfaces is excluded

  Note: All other uses of pragma directives are permitted.

  Note: Support for pragma directives is compiler implementation-dependent. A compiler ignores pragma directives it does not recognize.

- Input/Output library standard iostream objects (ISO/IEC 14882:2003, §27.3) and all dependencies on them (e.g., ISO/IEC 14882:2003, §27.4.2.1.6, 27.4.2.4) are excluded

  Note: Some Input/Output library capabilities (e.g., file streams) may have other implementation-dependent platform dependencies (e.g., file storage device, Input/Output device).

- Component use of wide characters (ISO/IEC 14882:2003, §3.9.1.5), multibyte characters (ISO/IEC 14882:2003, §1.3.8), wide strings (ISO/IEC 14882:2003, §17.3.2.1.3.3), and multibyte strings (ISO/IEC 14882:2003, §17.3.2.1.3.2) is excluded, including library functions that manipulate those types

For the portion of the C++ library functions that are in common with the C library functions, including functions categorized in Section A.1 as POSIX_C_LANG_SUPPORT_R, only the functions defined in Appendix A for the General Purpose Profile are supported.

Exception Handling (ISO/IEC 14882:2003, §15, 18.6, 19.1) is supported except across the FACE defined API boundaries. Exceptions may be thrown and caught within a single UoC.

4.2.3.3.2 C++ Programming Language Definition for Safety Extended Capability Set

The Safety Extended C++ Programming Language capability set includes features from the Programming Language specification defined in ISO/IEC 14882:2003: Programming Languages – C++ with the following modifications:

- Component use of the pragma directive (ISO/IEC 14882:2003, §16.6) for data structure compositions on FACE Interfaces is excluded

  Note: All other uses of pragma directives are permitted.

  Note: Support for pragma directives is compiler implementation-dependent. A compiler ignores pragma directives it does not recognize.

- Input/output library (ISO/IEC 14882:2003, §27) is excluded

  Note: For input/output support (including file system), safety-related C++ components use the input/output functional interfaces defined as part of the corresponding OSS Profile.

- Component use of wide characters (ISO/IEC 14882:2003, §3.9.1.5), multibyte characters (ISO/IEC 14882:2003, §1.3.8), wide strings (ISO/IEC 14882:2003, §17.3.2.1.3.3), and multibyte strings (ISO/IEC 14882:2003, §17.3.2.1.3.2) is excluded, including library functions that manipulate those types

- C++ Standard Template Libraries (STL) (ISO/IEC 14882:2003, §19, 20, 21, 22, 23, 24, 25, 26, 27) are excluded

  Note: No restrictions on software suppliers developing their own template implementations (ISO/IEC 14882:2003, §14).

- Run-Time Type Information (RTTI) (ISO/IEC 14882:2003, §18.5) and use of *dynamic_cast* (ISO/IEC 14882:2003, §5.2.7) are excluded

Dynamic memory management via "operator new" and "operator delete" is supported. Software components may override the default operator new and operator delete (ISO/IEC 14882:2003, §17.4.3.4) to implement software component-specific object memory management systems.

The C++ library functions (ISO/IEC 14882:2003, §18.1, 18.2.2, 19.3, 20.4.6, 21.4, 26.5, 27.8.2) that are supported are those that are in common with the C library functions, including functions categorized in Section A.1 as POSIX_C_LANG_SUPPORT_R, defined in the corresponding OSS Profile defined in Appendix A.

Exception Handling (ISO/IEC 14882:2003, §15, 18.6, 19.1) is supported except across the FACE defined API boundaries. Exceptions may be thrown and caught within a single UoC.

4.2.3.3.3 C++ Programming Language Definition for Safety Base and Security Capability Sets

Restrictions to the C++ Programming Language features and libraries for the Safety Base and Security capability sets are based on recommendations in DO-332 (Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A) and recommendations from the Embedded C++ Committee. The recommendations from these are used to define C++ restrictions appropriate for use in safety-critical and real-time systems.

The Safety Base and Security C++ Programming Language capability sets includes features from the Programming Language specification defined in ISO/IEC 14882:2003: Programming Languages – C++ with the following modifications:

- Component use of the pragma directive (ISO/IEC 14882:2003, §16.6) for data structure compositions on FACE Interfaces are excluded

  Note: All other uses of pragma directives are permitted.

  Note: Support for pragma directives is compiler implementation-dependent. A compiler ignores pragma directives it does not recognize.

- Virtual base classes (ISO/IEC 14882:2003, §10.1) are excluded

- Dynamic memory de-allocation via default "operator delete" (ISO/IEC 14882:2003, §3.7.3.2, 18.4.1) is excluded

  Note: Dynamic memory allocation via "operator new" is supported. Software components can override the default operator new and delete (ISO/IEC 14882:2003, §17.4.3.4) to implement software component-specific object memory management systems.

  Note: The intention is to provide the single object and array allocation functions.

  ```
  o void* operator new ( std::size_t count );
  o void* operator new[]( std::size_t count );
  ```

  However, allocators should not throw an exception. A failing new operation that returns to the caller should return NULL.

- Run-Time Type Information (ISO/IEC 14882:2003, §18.5) and use of dynamic_cast (ISO/IEC 14882:2003, §5.2.7) are excluded

- Exception Handling (ISO/IEC 14882:2003, §15, 18.6, 19.1) is excluded

- C++ Standard Template Libraries (ISO/IEC 14882:2003, §19, 20, 21, 22, 23, 24, 25, 26, 27) are excluded

  Note: There are no restrictions on software suppliers developing their own template implementations (ISO/IEC 14882:2003, §14).

- Input/output library (ISO/IEC 14882:2003, §27) is excluded

  Note: There are no C++ input/output library functions supported. For input/output support (including file system), safety-related C++ components utilize the input/output functional interfaces defined as part of the corresponding OSS Profile.

- Component use of wide characters (ISO/IEC 14882:2003, §3.9.1.5), multibyte characters (ISO/IEC 14882:2003, §1.3.8), wide strings (ISO/IEC 14882:2003, §17.3.2.1.3.3), and multibyte strings (ISO/IEC 14882:2003, §17.3.2.1.3.2) is excluded, including library functions that manipulate those types

The C++ library functions (ISO/IEC 14882:2003, §18.1, 18.2.2, 19.3, 20.4.6, 21.4, 26.5, 27.8.2) that are supported are only those that are in common with the C library functions, including functions categorized in Section A.1 as POSIX_C_LANG_SUPPORT_R, defined in the corresponding OSS Profile.

4.2.3.3.4 C++ Programming Language and Run-Time Requirements

The requirements associated with C++ Programming Language Run-Time include:

1. UoCs using the C++ Programming Language Run-Time supplied by the OS shall be restricted to the Programming Language features for the selected capability set.

   Note: An OSS UoC must provide the functionality as defined, but may support excluded and/or additional features.

2. UoCs using the C++ Programming Language Run-Time supplied by the OS shall be restricted to the Programming Language library functions, including functions categorized in Section A.1 as POSIX_C_LANG_SUPPORT_R, for the selected capability set.

   Note: An OSS UoC must provide the functionality as defined, but may support excluded and/or additional features.

3. OSS UoCs providing a C++ Programming Language Run-Time shall support the capabilities defined in the selected capability set.

   Note: An OSS UoC must provide the functionality as defined, but may support excluded and/or additional features.

   Note: The appropriate capability set sections for the preceding requirements are as follows:

   — General Purpose capability set – Section 4.2.3.3.1

   — Safety Extended capability set – Section 4.2.3.3.2

   — Safety Base capability sets – Section 4.2.3.3.3

   — Security capability set – Section 4.2.3.3.3

4. UoCs using the C++ STL function calls in the Safety (Extended or Base) or Security capability sets shall encapsulate the C++ STL functions within the UoC.

4.2.3.4 Ada Programming Language

The following sections define Ada Programming Language features and library functions for the supported capability sets. These capability sets are listed in the order of most permissive (General Purpose) to most restrictive (Security). As with the OSS Profiles, the restrictions are established to allow deployment of a UoC in an environment developed to any of the more permissive capability sets. Note that the deployment of a UoC into a more permissive capability set may not be adequate for the required design assurance of the UoC.

Restrictions to Ada 95 and Ada 2012 Programming Language features and libraries for the Safety Extended, Safety Base, and Security capability sets are based on the Ravenscar Ada subset profile developed at the Eighth International Real-Time Ada Workshop. This profile is advocated in the ISO/IEC/JTC1/SC22/WG9 Technical Report TR 15942 and is defined in Section D.13 of the Ada 2012 Language Reference Manual. The Ravenscar Ada subset is enforced by a compiler using “pragma Restrictions” in Ada 95 or “pragma Profile(Ravenscar)” in Ada 2012.

4.2.3.4.1 Ada 95 Programming Language Definition for General Purpose Capability Set

The General Purpose Ada 95 Programming Language capability set includes features from the Programming Language specification defined in ANSI/ISO/IEC 8652:1995: Ada Reference Manual, Language, and Standard Libraries (herein referred to as "Ada 95 LRM") with the following modifications:

- Implementation-defined pragmas (Ada 95 LRM, §2.8 (14)) for data structure compositions on FACE Interfaces are excluded

  Note: All other uses of implementation-defined pragma directives are permitted.

  Note: Use of the language-defined pragmas (e.g., pragma Priority, pragma Import, pragma Export) defined throughout the Ada 95 LRM is permitted.

  Note: Support for pragma directives is compiler implementation-dependent. A compiler ignores pragma directives it does not recognize.

- Asynchronous Transfer of Control (Ada 95 LRM, §9.7.4) is excluded

- Wide characters, wide strings, and wide text are excluded

- Input/Output capabilities as defined in Ada 95 LRM, §13.13, A.6, A.7, A.8, A.9, A.10, A.11, A.12, A.13 access to files requiring any external communications interface hardware or to external hardware devices is excluded

  As described in §A.10, *In_File* and *Out_File* must be defined to an internal file. This file definition restriction applies to all of Annex A and §13.13.

  Note: The definition of *external_file* and *file_objects* is restricted to files accessible internally by the OSS.

- The Distributed Systems Annex (Ada 95 LRM, Annex E) is excluded

- The Information Systems Annex (Ada 95 LRM, Annex F) is excluded

For Ada 95-based components, the component uses the tasking/threading capabilities defined as part of the Programming Language.

The supported Ada 95 exception handling is maintained except across the FACE defined API boundaries. Exceptions may be thrown and caught within a single UoC.

4.2.3.4.2 Ada 2012 Programming Language Definition for General Purpose Capability Set

The General Purpose Ada 2012 Programming Language capability set includes features from the Programming Language specification defined in ISO/IEC 8652:2012(E) with Technical Corrigendum 1: Ada Reference Manual, Language, and Standard Libraries (herein referred to as "Ada 2012 LRM"), with the following modifications:

- Implementation-defined pragmas (Ada 2012 LRM, §2.8 (14)) for data structure compositions on FACE Interfaces are excluded

  Note: All other uses of implementation-defined pragma directives are permitted.

  Note: Use of the language-defined pragmas (e.g., pragma Priority, pragma Import, pragma Export) defined throughout the Ada 2012 LRM is permitted.

Note: Support for pragma directives is compiler implementation-dependent. A compiler ignores pragma directives it does not recognize.

- Asynchronous Transfer of Control (Ada 2012 LRM, §9.7.4) is excluded
- Wide characters, wide strings, and wide text are excluded
- Wide wide characters, wide wide strings, and wide wide text are excluded
- Input/Output capabilities as defined in Ada 2012 LRM, §13.13, A.6, A.7, A.8, A.9, A.10, A.11, A.12, A.13 access to files requiring any external communications interface hardware or to external hardware devices is excluded

  As described in §A.10, *In_File* and *Out_File* must be defined to an internal file. This file definition restriction applies to all of Annex A and §13.13.

  Note: The definition of *external_file* and *file_objects* is restricted to files accessible internally by the OSS.
- The Distributed Systems Annex (Ada 2012 LRM, Annex E) is excluded d
- The Information Systems Annex (Ada 2012 LRM, Annex F) is excluded

For Ada 2012-based components, the component uses the tasking/threading capabilities defined as part of the Programming Language.

Ada 2012 exception handling is supported except across the FACE defined API boundaries. Exceptions may be thrown and caught within a single UoC.

4.2.3.4.3 Ada 95 Programming Language Definition for Safety Extended Capability Set

The Safety Extended Ada 95 Programming Language capability set includes Programming Language features based on a subset definition of the ANSI/ISO/IEC 8652:1995: Ada 95 Reference Manual, Language, and Standard Libraries (i.e., Ada 95 LRM), and as restricted by the Ravenscar Profile for High-Integrity Systems, ISO/IEC/JTC1/SC22/WG9 Technical Report TR 1594215942 with the following modifications:

- Component use of implementation-defined pragmas (Ada 95 LRM, §2.8 (14)) for data structure compositions on FACE Interfaces is excluded

  Note: All other uses of implementation-defined pragma directives are permitted.

  Note: Use of the language-defined pragmas (e.g., pragma Priority, pragma Import, pragma Export, etc.) defined throughout the Ada 95 LRM is permitted.

  Note: Support for pragma directives is compiler implementation-dependent. A compiler ignores pragma directives it does not recognize.
- Asynchronous Transfer of Control (Ada 95 LRM, §9.7.4) and dependencies are excluded
- Exception handling (Ada 95 LRM, §11) the *Exception_Information* and *Exception_Message* functions are excluded
- Deallocation in Storage Management (Ada 95 LRM, §13.11) is excluded (i.e., no usage of the Deallocate procedure, *Unchecked_Deallocation*) and memory allocation post startup/initialization is excluded

- Wide characters, wide strings, and wide text are excluded
- Random Number Generation (Ada 95 LRM, §A.5.2) is excluded
- Input/output capabilities (Ada 95 LRM, §13.13, A.6, A.7, A.8, A.9, A.10, A.11, A.12, A.13, A.14, A.15) are excluded
- The Distributed Systems Annex (Ada 95 LRM, Annex E) is excluded
- The Information Systems Annex (Ada 95 LRM, Annex F) is excluded
- The Numerics Annex (Ada 95 LRM, Annex G) is excluded
- Unbounded strings (the type Unbounded_String in Ada.Strings.Unbounded, Ada 95 LRM, Section A.4.5) are excluded

The capability set includes an Ada task's use of secondary stack (if required) limited to a defined size.

Note: The recommended minimum value for this size is 4096 bytes.

The capability set includes the subset of functionality defined for the Predefined Language Environment (Ada 95 LRM, Annex A) based on the above and the Ravenscar Ada 95 subset profile exclusions.

The capability set includes the subset of functionality defined for the Predefined Language Environment (ANSI/ISO/IEC 8652:1995, Annex A) based on the above and Ravenscar Ada 95 subset profile exclusions.

The capability set includes the subset of functionality defined for Interfaces to Other Languages (Ada 95 LRM, Annex B) as follows:

- Sections B.1 and B.2 are included
- Sections B.3.1 and B.3.2 are excluded
- The remainder of Section B.3 is included

The capability set includes the subset of functionality defined for Systems Programming (Ada 95 LRM, Annex C), based on ISO/IEC TR 15942:2000 including Interrupts support (Ada 95 LRM, §C.3) limited to constants and type definitions associated with Ada.Interrupts with the following modification:

- Dependencies on package Task_Attributes (Ada 95 LRM, §C.7.2) are excluded

The capability set includes the subset of functionality defined for Real-Time Systems (Ada 95 LRM, Annex D), based on ISO/IEC TR 15942:2000 including support for monotonic time (Ada 95 LRM, §D.8) with the following modifications:

- Dependencies on package Ada.Calendar (Ada 95 LRM, §9.6) are excluded
- Support for relative delay statements (Ada 95 LRM, §9.6) is excluded

Accuracy information related to the elementary functions (Ada 95 LRM, §A.5) is provided by the run-time supplier.

Ada-based UoCs may use the Ada tasking capability defined as part of the Programming Language (restricted to the Ravenscar subset) or the tasking/threading from the OS environment (i.e., ARINC 653 or POSIX). The supported Ada 95 exception handling is maintained except across the FACE defined API boundaries. Exceptions may be thrown and caught within a single UoC.

4.2.3.4.4 Ada 2012 Programming Language Definition for Safety Extended Capability Set

The Safety Extended Ada 2012 Programming Language capability set includes Programming Language features based on a subset definition of the ISO/IEC 8652:2012(E) with Technical Corrigendum 1: Ada Reference Manual, Language, and Standard Libraries (i.e., Ada 2012 LRM) and as restricted by the Ravenscar Profile (Annex D.13) with the following modifications:

- Component use of implementation-defined pragmas (Ada 2012 LRM, §2.8 (14)) for data structure compositions on FACE Interfaces is excluded

  Note: All other uses of implementation-defined pragma directives are permitted.

  Note: Use of the language-defined pragmas (e.g., pragma Priority, pragma Import, pragma Export, etc.) defined throughout the Ada 2012 LRM is permitted.

  Note: Support for pragma directives is compiler implementation-dependent. A compiler ignores pragma directives it does not recognize.

- Asynchronous Transfer of Control (Ada 2012 LRM, §9.7.4) and dependencies are excluded

- Synchronized, task, and protected interfaces (Ada 2012 LRM 3.9.4) are excluded

- Exception handling (Ada 2012 LRM, §11) the *Exception_Information* and *Exception_Message* functions are excluded

- The only permitted syntax for a formal_package_actual_part in a formal_package_declaration (Ada 2012 LRM 12.7) is:

  ```
  formal_package_actual_part ::= (<>) | [generic_actual_part]
  ```

- Deallocation in Storage Management (Ada 2012 LRM, §13.11) is excluded (i.e., no usage of the Deallocate procedure, *Unchecked_Deallocation*) and memory allocation post startup/initialization is excluded

- Wide characters, wide strings, and wide text are excluded

- Wide wide characters, wide wide strings, and wide wide text are excluded

- Random Number Generation (Ada 2012 LRM, §A.5.2) is excluded

- Input/output capabilities (Ada 2012 LRM, §13.13, A.6, A.7, A.8, A.9, A.10, A.11, A.12, A.13, A.14, A.15) are excluded

- The Containers library (Ada 2012 LRM, §A.18) is excluded

- The Distributed Systems Annex (Ada 2012 LRM, Annex E) is excluded

- The Information Systems Annex (Ada 2012 LRM, Annex F) is excluded

- The Numerics Annex (Ada 2012 LRM, Annex G) is excluded
- Unbounded strings (the type Unbounded_String in Ada.Strings.Unbounded, Ada 2012 LRM, §A.4.5) are excluded

The capability set includes an Ada task's use of secondary stack (if required) limited to a defined size.

Note: The recommended minimum value for this size is 4096 bytes.

The capability set includes the subset of functionality defined for the Predefined Language Environment (Ada 2012 LRM, Annex A) based on the above and the Ravenscar Ada 95 subset profile exclusions.

The capability set includes the subset of functionality defined for Interfaces to Other Languages (Ada 2012 LRM, Annex B) as follows:

- Sections B.1 and B.2 are included
- Sections B.3.1 and B.3.2 are excluded
- The remainder of Section B.3 is included

The capability set includes the subset of functionality defined for Systems Programming (Ada 2012 LRM, Annex C), based on ISO/IEC TR 15942:2000 including Interrupts support (Ada 2012 LRM, §C.3) limited to constants and type definitions associated with Ada.Interrupts with the following modification:

- Dependencies on package Task_Attributes (Ada 2012 LRM, §C.7.2) are excluded

The capability set includes the subset of functionality defined for Real-Time Systems (Ada 2012 LRM, Annex D), based on ISO/IEC TR 15942:2000 with the following restrictions:

- D.2.1 The Task Dispatching Model: Ada.Dispatching.Yield is excluded
- D.2.2 Task Dispatching Pragmas: pragma Priority_Specific_Dispatching is excluded
- D.2.4 Non-Preemptive Dispatching: Excluded
- D.2.5 Round Robin Dispatching: Excluded
- D.2.6 Earliest Deadline First Dispatching: Excluded
- D.4 Entry Queuing Policies: Excluded (since there are no entry queues)
- D.5 Dynamic Priorities: Excluded
- D.5.1 Dynamic Priorities for Tasks: Excluded
- D.5.2 Dynamic Priorities for Protected Objects: Excluded
- D.6 Preemptive Abort: Excluded
- D.10 Synchronous Task Control: Ada.Synchronous_Task_Control.EDF is excluded
- D.10.1 Synchronous Barriers: Excluded
- D.11 Asynchronous Task Control: Excluded

- D.12 Other Optimizations and Determinism Rules: Excluded

Note: In general, optimizations are outside the scope of this Technical Standard.

- D.14 Execution Time: Excluded
- D.15 Timing Events: Excluded
- D.16 Multiprocessor Implementation: Excluded

Accuracy information related to the elementary functions (Ada 2012 LRM, §A.5) is provided by the run-time supplier.

Ada-based UoCs may use the Ada tasking capability defined as part of the Programming Language (restricted to the Ravenscar subset) or the tasking/threading from the OS environment (i.e., ARINC 653 or POSIX). The supported Ada 2012 exception handling is maintained except across the FACE defined API boundaries. Exceptions may be thrown and caught within a single UoC.

4.2.3.4.5 Ada 95 Programming Language Definition for Safety Base and Security Capability Sets

The Safety Base and the Security Ada 95 Programming Language capability sets include Programming Language features based on a subset definition of the ANSI/ISO/IEC 8652:1995: Ada 95 Reference Manual, Language, and Standard Libraries (i.e., Ada 95 LRM), and as restricted by the Ravenscar Profile for High-Integrity Systems, ISO/IEC/JTC1/SC22/WG9, Technical Report TR 1594215942 with the following modifications:

- Component use of implementation-defined pragmas (Ada 95 LRM, §2.8 (14)) for data structure compositions on FACE Interfaces is excluded

  Note: All other uses of implementation-defined pragma directives are permitted.

  Note: Use of the language-defined pragmas (e.g., pragma Priority, pragma Import, pragma Export) defined throughout the Ada 95 LRM is permitted.

- Asynchronous Transfer of Control (Ada 95 LRM, §9.7.4) and dependencies are excluded
- Exception Handling (Ada 95 LRM, §11) is limited to handling predefined exceptions using a single default handler (i.e., pragma Restrictions *No_Exception_Handlers*)
- Storage Management (Ada 95 LRM, §13.11 (i.e., as excluded by pragma Restrictions *No_Allocators*) and dependencies are excluded
- String Handling (Ada 95 LRM, §A.4) is excluded
- Random Number Generation (Ada 95 LRM, §A.5.2) is excluded
- Input/output capabilities (Ada 95 LRM, §13.13, A.6, A.7, A.8, A.9, A.10, A.11, A.12, A.13, A.14, A.15) are excluded
- The Distributed Systems Annex (Ada 95 LRM, Annex E) is excluded
- The Information Systems Annex (Ada 95 LRM, Annex F) is excluded
- The Numerics Annex (Ada 95 LRM, Annex G) is excluded

The capability sets include the subset of functionality defined for the Predefined Language Environment (Ada 95 LRM, Annex A) based on the above and the Ravenscar Ada 95 subset profile exclusions.

The capability sets include the subset of functionality defined for Interfaces to Other Languages (Ada 95 LRM, Annex B) that is limited to constant and type definitions associated with Interfaces.

The capability sets include the subset of functionality defined for Systems Programming (Ada 95 LRM, Annex C), based on ISO/IEC TR 15942:2000 including Interrupts support (Ada 95 LRM, §C.3) limited to constants and type definitions associated with Ada.Interrupts with the following modification:

- Dependencies on package Task_Attributes (Ada 95 LRM, §C.7.2) are excluded

The capability sets include the subset of functionality defined for Real-Time Systems (Ada 95 LRM, Annex D), based on ISO/IEC TR 15942:2000 including support for monotonic time (Ada 95 LRM, §D.8) with the following modifications:

- Dependencies on package Ada.Calendar (Ada 95 LRM, §9.6) are excluded
- Support for relative delay statements (Ada 95 LRM, §9.6) are excluded

Accuracy information related to the elementary functions (Ada 95 LRM, §A.5) is provided by the run-time supplier.

Ada-based UoCs smay use the Ada tasking capability defined as part of the Programming Language (restricted to the Ravenscar subset) or the tasking/threading from OS environment (i.e., ARINC 653 or POSIX).

4.2.3.4.6 Ada 2012 Programming Language Definition for Safety Base and Security Capability Sets

The Safety Base and the Security Ada 2012 Programming Language capability sets include Programming Language features based on a subset definition of the ISO/IEC 8652:2012(E) with Technical Corrigendum 1: Ada Reference Manual, Language, and Standard Libraries (i.e., Ada 2012 LRM) and as restricted by the Ravenscar Profile (Annex D.13) with the following modifications:

- Component use of implementation-defined pragmas (Ada 2012 LRM, §2.8 (14)) for data structure compositions on FACE Interfaces is excluded

  Note: All other uses of implementation-defined pragma directives are permitted.

  Note: Use of the language-defined pragmas (e.g., pragma Priority, pragma Import, pragma Export) defined throughout the Ada 2012 LRM is permitted.

  Note: Support for pragma directives is compiler implementation-dependent. A compiler ignores pragma directives it does not recognize.

- Asynchronous Transfer of Control (Ada 2012 LRM, §9.7.4) and dependencies are excluded
- Exception Handling (Ada 2012 LRM, §11) is limited to handling predefined exceptions using a single default handler (i.e., pragma Restrictions *No_Exception_Handlers*)
- Synchronized, task, and protected interfaces (Ada 2012 LRM §3.9.4) are excluded

- The only permitted syntax for a formal_package_actual_part in a formal_package_declaration (Ada 2012 LRM §12.7) is:

  ```
  formal_package_actual_part ::= (<>) | [generic_actual_part]
  ```

- Storage Management (Ada 2012 LRM, §13.11 (i.e., as excluded by pragma Restrictions *No_Allocators*) and dependencies are excluded
- Wide characters, wide strings, and wide text are excluded
- Wide wide characters, wide wide strings, and wide wide text are excluded
- String Handling (Ada 2012 LRM, §A.4) is excluded
- Random Number Generation (Ada 2012 LRM, §A.5.2) is excluded
- Input/output capabilities (Ada 2012 LRM, §13.13, A.6, A.7, A.8, A.9, A.10, A.11, A.12, A.13, A.14, A.15) are excluded
- The Containers library (Ada 2012 LRM, §A.18) is excluded.
- The Distributed Systems Annex (Ada 2012 LRM, Annex E) is excluded
- The Information Systems Annex (Ada 2012 LRM, Annex F) is excluded
- The Numerics Annex (Ada 2012 LRM, Annex G) is excluded

The capability sets include the subset of functionality defined for the Predefined Language Environment (Ada 2012 LRM, Annex A) based on the above and the Ravenscar Ada 2012 subset profile exclusions.

The capability sets include the subset of functionality defined for Interfaces to Other Languages (Ada 2012 LRM, Annex B) that is limited to constant and type definitions associated with Interfaces.

The capability sets include the subset of functionality defined for Systems Programming (Ada 2012 LRM, Annex C), based on ISO/IEC TR 15942:2000 including Interrupts support (Ada 2012 LRM, §C.3) limited to constants and type definitions associated with Ada.Interrupts with the following modification:

- Dependencies on package Task_Attributes (Ada 2012 LRM, §C.7.2) are excluded

The capability set includes the subset of functionality defined for Real-Time Systems (Ada 2012 LRM, Annex D), based on ISO/IEC TR 15942:2000 with the following restrictions:

- D.2.1 The Task Dispatching Model: Ada.Dispatching.Yield is excluded
- D.2.2 Task Dispatching Pragmas: pragma Priority_Specific_Dispatching is excluded
- D.2.4 Non-Preemptive Dispatching: Excluded
- D.2.5 Round Robin Dispatching: Excluded
- D.2.6 Earliest Deadline First Dispatching: Excluded
- D.4 Entry Queuing Policies: Excluded (since there are no entry queues)
- D.5 Dynamic Priorities: Excluded

- D.5.1 Dynamic Priorities for Tasks: Excluded
- D.5.2 Dynamic Priorities for Protected Objects: Excluded
- D.6 Preemptive Abort: Excluded
- D.10 Synchronous Task Control: Ada.Synchronous_Task_Control.EDF is excluded
- D.10.1 Synchronous Barriers: Excluded
- D.11 Asynchronous Task Control: Excluded
- D.12 Other Optimizations and Determinism Rules: Excluded

Note: In general, optimizations are outside the scope of this Technical Standard.

- D.14 Execution Time: Excluded
- D.15 Timing Events: Excluded
- D.16 Multiprocessor Implementation: Excluded

Accuracy information related to the elementary functions (Ada 2012 LRM, §A.5) is provided by the run-time supplier.

Ada-based UoCs may use the Ada tasking capability defined as part of the Programming Language (restricted to the Ravenscar subset) or the tasking/threading from OS environment (i.e., ARINC 653 or POSIX).

4.2.3.5 Ada Programming Language and Run-Time Requirements

The requirements associated with the Ada Programming Language include:

1. UoCs using an Ada 95 Run-Time supplied by the OS shall be restricted to the Programming Language features for the selected capability set.
2. UoCs using an Ada 2012 Run-Time supplied by the OS shall be restricted to the Programming Language features for the selected capability set.
3. OSS UoCs providing an Ada 95 Run-Time shall support the capabilities defined in the selected capability set.

   Note: An OSS UoC must provide the functionality as defined, but may support excluded and/or additional features.

   Note: The appropriate capability set sections for the preceding requirements are as follows:

   — Ada 95 General Purpose capability set – Section 4.2.3.4.1

   — Ada 95 Safety Extended capability set – Section 4.2.3.4.3

   — Ada 95 Safety Base capability set – Section 4.2.3.4.4

   — Ada 95 Security capability set – Section 4.2.3.4.4
4. OSS UoCs providing an Ada 2012 Run-Time shall support the capabilities defined in the selected capability set.

Note: An OSS UoC must provide the functionality as defined, but may support excluded and/or additional features.

Note: The appropriate capability set sections for the preceding requirements are as follows:

— Ada 2012 General Purpose capability set – Section 4.2.3.4.2

— Ada 2012 Safety Extended capability set – Section 4.2.3.4.4

— Ada 2012 Safety Base capability set – Section 4.2.3.4.6

— Ada 2012 Security capability set – Section 4.2.3.4.6

5. Safety (Extended or Base) and Security capability set UoCs using input/output support (including file system) shall utilize Ada bindings for the input/output functional interfaces defined as part of the corresponding OSS Profile.

4.2.3.6 Java Programming Language

The Java Programming Language may be available in the General Purpose and Safety Extended capability sets. The following sections define the Java Programming Language support.

4.2.3.6.1 Java Programming Language Definition for General Purpose Capability Set

The General Purpose Java Programming Language capability set includes either of the following Programming Language specifications:

- Programming language features described in Java Platform, Enterprise Edition 8 (Java EE 8)
- Programming language features described in Java Platform, Standard Edition 8 (Java SE 8)

For Java-based components, the component utilizes the tasking/threading capabilities defined as part of the Programming Language. Support for communications between partitions includes use of ARINC 653 sampling and queuing port interfaces.

Java exception handling is supported except across the FACE defined API boundaries. Exceptions may be thrown and caught within a single UoC.

4.2.3.6.2 Java Programming Language Definition for Safety Extended Capability Set

The Safety Extended Java Programming Language capability set includes the following Programming Language specification:

- Java Platform, Standard Edition 8 (Java SE 8)

For Java-based components, the component utilizes the tasking/threading capabilities defined as part of the Programming Language or the OS environment (ARINC 653 or POSIX). Support for communications between partitions includes use of ARINC 653 sampling and queuing port interfaces.

Java exception handling is supported except across the FACE defined API boundaries. Exceptions may be thrown and caught within a single UoC.

4.2.3.6.3 Java Programming Language and Run-Time Requirements

The requirements associated with this Programming Language include:

1. UoCs using the Java Run-Time supplied by the OS shall be restricted to the Programming Language features for the selected capability set.

   Note: An OSS UoC must provide the functionality as defined, but may support excluded and/or additional features.

2. OSS UoCs providing a Java Run-Time shall include the Programming Language features for the selected capability set.

   Note: An OSS UoC must provide the functionality as defined, but may support excluded and/or additional features.

   Note: The appropriate capability set sections for the preceding requirements are as follows:

   — General Purpose capability set – Section 4.2.3.6.1

   — Safety Extended capability set – Section 4.2.3.6.2

   Note: There are no Programming Language Safety Base or Security capability sets.

4.2.4 Component Framework Interfaces

4.2.4.1 Component Framework

Component Frameworks can help increase the efficiency of creating new software by allowing developers to focus on the unique requirements of software components without having to use resources to develop lower-level details of how to manage functionality. Component Frameworks are not the same as portable libraries. With portable libraries, components instantiate and invoke the functions and objects provided by the portable library. With Component Frameworks, developers implement functions and objects specific to their own component that are then instantiated and invoked by the Component Framework.

A Component Framework provided by the OSS is extending the OSS Interface to include the Component Framework's own API. An example of a Component Framework is the Java virtual machine.

The following subsection defines the Component Frameworks for each OSS Profile that can be optionally included in the OSS.

4.2.4.2 OSS OSGi Framework Support Requirements

For requirements on Component Frameworks provided as part of a UoC, see the requirements for FSC (Section 4.7.14) and the requirements for the segment relevant to the UoC.

1. When an OSS UoC built to the General Purpose Profile supports OSGi, the OSS UoC OSGi support shall be comprised of the OSGi API described in OSGi Service Platform Release 7 Core for Java-based systems.

2. When an OSS UoC built to the Safety Profile supports OSGi, the OSS UoC OSGi support shall be comprised of the OSGi API described in OSGi Service Platform Release 7 Core for Java-based systems.

Note: The requirement intent is compatibility with an OSS UoC built to the Safety Profile. For example, a UoC not built for safety applications utilizing the OSGi API may be deployed in a partition of an OSS UoC built to the Safety Profile.

A main characteristic of software components developed to the Security Profile is the utilization of a security-related process for development and verification for the software component and its execution environment. This results in significantly restricting the Component Framework features and library functions in support of security-related software objectives.

No OSS Component Frameworks usable by a software component developed to the Security Profile have been defined.

4.2.5 Configuration Services

4.2.5.1 Configuration Services Introduction

Portable software components are designed to be deployable onto different implementations of the FACE Reference Architecture. Successful integration of any software component into varying subsystems/platforms may require the ability to modify the software component's behavior through configuration. This section details the configuration requirements and artifacts provided with UoCs by the software supplier to the system integrator.

Run-time configuration involves requests for, receipt of, and processing of configuration data by an executing software component. The software component's processing of configuration data has the effect of changing the software component's algorithms, behavior, or communication in one of their predefined ways without modification of the software component itself.

4.2.5.2 Configuration Services Requirements

1. When providing the Configuration Services Interface, a UoC shall provide the Configuration Services Interface as specified in Section G.2.
2. The configuration parameters of a UoC shall be described in XML conformant to version 1.1 of the XSD standard.
3. When the Configuration Services Interface is implemented, the IDL to Programming Language Mappings defined in Section 4.14 shall be used.

4.3 Device Drivers

A device driver encapsulates hardware-dependent and operating system-specific interfaces used to control and operate a device, such as an I/O device. A device driver is accessed via a device driver interface. Device driver interfaces may include:

- The FACE OS Interface
- POSIX interfaces that are not part of an existing FACE OSS Profile, such as termios
- Non-POSIX interfaces available through operating system kernels, such as ioctl
- Libraries encapsulating device drivers instantiated in user space

The FACE Technical Standard defines FACE OSS Profiles that constrain the OS APIs that a FACE UoC may use. The FACE Reference Architecture isolates use of device drivers to FACE

UoCs designed to abstract those devices from the portable software. Device drivers provide a mechanism for a UoC to access a hardware device which otherwise would not be allowed due to the UoC's FACE OSS Profile restrictions. Sections 4.4, 4.6.3.1.3, 4.7.2, 4.12.4, and 4.12.9 describe how device drivers can be used in the FACE Reference Architecture.

4.4 I/O Services Segment

The I/O Services Segment (IOSS) abstracts interface hardware and device drivers into I/O Services that implement communication via the IOS Interface between PSSS UoCs and I/O devices. An I/O Service is defined for each of several common I/O bus architectures. A PSSS UoC can use several I/O Services to access multiple I/O bus architectures, and an I/O Service can provide communications with the same I/O bus architecture to multiple PSSS UoCs. An IOSS UoC packages one or more I/O Services.

Figure 8 illustrates the potential relationships between PSSS UoCs and I/O Services. The figure depicts MIL-STD-1553, Serial and Discrete bus architectures. PSSS UoC A requires the MIL-STD-1553 and Serial I/O Services. PSSS UoC B requires the Serial and Discrete I/O Services. The packaging of the I/O Services does not impact either PSSS UoC.

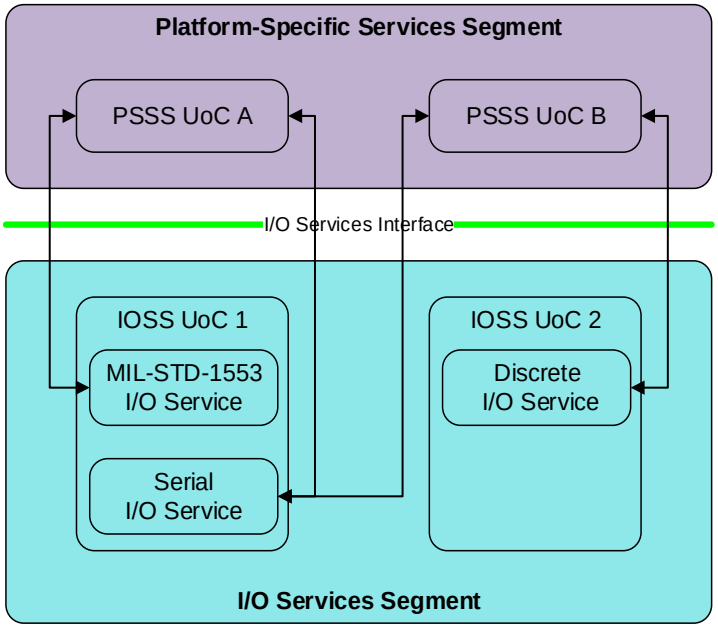

Figure 8: I/O Services Related to PSSS and IOSS UoCs

An I/O connection is the logical relationship between a PSSS UoC and a specific I/O device via the I/O Services Interface. It is implemented by an I/O Service. Figure 9 depicts several potential bus and device configurations that are normalized by an I/O Service. PSSS UoC A has four I/O connections, where each connection is a line to an I/O Service. Two connections are to separate MIL-STD-1553 devices, each on a separate bus that are accessed via a single MIL-STD-1553 I/O Service. Similarly, the other two connections are to serial devices on the same bus that are accessed via a single Serial I/O Service. Each I/O Service encapsulates computing platform details so PSSS UoC A is not impacted.

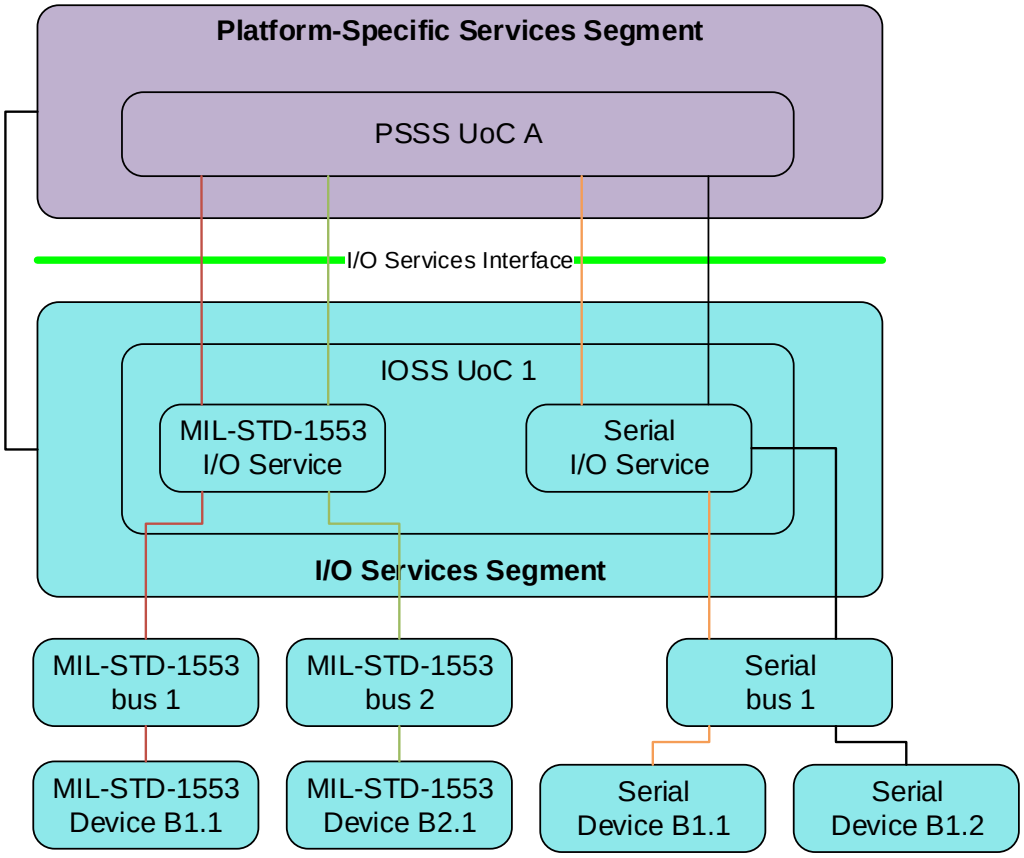

Figure 9: I/O Connections Between PSSS UoCs and I/O Devices

An IOSS UoC is constrained to a FACE OSS Profile except when it may need to call non-standard OS or device driver interfaces of the FACE Computing Environment. The IOSS exists to encapsulate this behavior so that PSSS UoCs can be constrained to FACE Interfaces, including the OSS Profiles and the I/O Services Interface.

The IOSS offers two capabilities to the PSSS. The I/O Service Management Capability addresses initialization, configuration, and status queries. The I/O Data Movement Capability encompasses communication via an I/O connection.

4.4.1 I/O Services Segment Requirements

1. An IOSS UoC shall provide at least one I/O Service.
2. When an IOSS UoC uses the OS Interface, the IOSS UoC shall use the OS Interface as specified by the applicable FACE OSS Profile defined in Section 4.2.1.
3. When an IOSS UoC uses multiple POSIX processes, the IOSS UoC shall use the POSIX multi-process APIs as defined in Section 4.2.1.
4. 4When an IOSS UoC uses OS Health Monitoring, the IOSS UoC defined to operate in a POSIX operational environment shall use the OS HMFM Interface described in Section 4.2.2.

5. When an IOSS UoC uses a Component Framework, the IOSS UoC shall use the Component Framework in accordance with Section 4.2.4.

6. When an IOSS UoC retrieves configuration information locally, the IOSS UoC shall use Configuration Services as defined in Section 4.2.5.

7. When an IOSS UoC uses the OSS Interface, the IOSS UoC shall adhere to the restrictions specified in Section A.6 and Section A.7.

8. An IOSS UoC shall directly access the device and/or use device driver interfaces for interaction with the device when access is not possible through an OS Interface.

9. When providing a LCM Services Interface, an IOSS UoC shall do so in accordance with the requirements of Section 4.13.

10. When using a LCM Services Interface, an IOSS UoC shall do so in accordance with the requirements of Section 4.13.

11. When using Programming Language Run-Times, an IOSS UoC shall do so in accordance with Section 4.2.3.

12. When the Injectable Interface is provided by an IOSS UoC, the Injectable Interface shall be in accordance with Section 4.11.4.1.

4.4.2 I/O Service Management Capability Requirements

1. The I/O Service Management Capability shall provide for initialization of an I/O bus architecture.

2. The I/O Service Management Capability shall provide for configuration of an I/O bus architecture using a configuration resource in accordance with Section 4.2.5.

3. The I/O Service Management Capability shall provide for the return of the status of an I/O bus architecture.

4. The I/O Service Management Capability shall provide for configuration of an I/O connection using a configuration resource in accordance with Section 4.2.5.

5. The I/O Service Management Capability shall provide for the return of the status of an I/O connection.

4.4.3 I/O Data Movement Capability Requirements

1. The I/O Data Movement Capability shall provide the capability to open an I/O connection.

2. The I/O Data Movement Capability shall provide the capability to close an I/O connection.

3. The I/O Data Movement Capability shall move data across an open I/O connection.

4. The I/O Data Movement Capability shall provide the message payload data without modification.

4.4.4 I/O Service Requirements

1. The I/O Service shall provide the I/O Service Management Capability for the corresponding I/O bus architecture per the requirements in Section 4.5.

2. The I/O Service shall provide the I/O Data Movement Capability for the corresponding I/O bus architecture per the requirements in Section 4.5.

3. The I/O Service shall support the I/O connection parameters of its corresponding I/O bus architecture as defined in Appendix C.

4. The I/O Service shall support the I/O connection status values of its corresponding I/O bus architecture as defined in Appendix C.

5. The I/O Service shall supply the return code value as specified in Appendix C.

4.5 I/O Services Interface

The IOS Interface defines a standard interface for communication between a PSSS UoC and an I/O device. This communication is implemented by I/O Services in the IOSS. The logical relationship between the PSSS UoC and the I/O device is called an I/O connection. Refer to Section 4.4 for a detailed description of an I/O Service and an I/O connection.

The functions of the IOS Interface are common for each I/O Service supporting a specific I/O bus architecture. Some functions have parameter types that are tailored for the corresponding I/O bus architecture. Refer to Appendix C for details regarding the function signatures and data types for each defined I/O Service.

The IOS Interface defines functions to configure and query status for both an I/O connection and its associated bus instance. Appendix C describes the configuration and status fields for each defined I/O Service. The I/O bus types specified in Appendix C and the bus type-specific APIs and configuration parameters are not intended to be exhaustive. These status and configuration parameters can be extended by the IOSS developer with previously undefined status and configuration details as needed. These extensions, as well as any previously undefined bus types, could be submitted for potential inclusion in the FACE Technical Standard going forward as required during Conformance Verification. Each PSSS UoC creates connections to an IOSS UoC that supports the corresponding bus type. The connection configuration information is defined so that the I/O Service can correlate a connection with a specific handle. In order to provide portability of PSSS UoCs across different platforms, the type of connection is identified as an analogy for each I/O bus architecture as depicted in Table 6. While these analogies are not requirements, they do represent the perspective from which the IOS Interface is defined and serve as recommended guidance.

Table 6: I/O Connection Analogies

I/O Service Name	I/O Bus Architecture	I/O Connection Analogy
Serial_IO	Serial	Port
Analog_IO	Analog	Analog Signal Channel
Discrete_IO	Discrete	Discrete Signal Channel
M1553_IO	MIL-STD-1553	Subaddress
ARINC429_IO	ARINC 429	Channel

I/O Service Name	I/O Bus Architecture	I/O Connection Analogy
ARINC825_IO	ARINC 825	Channel
Synchro_IO	Synchro	Synchro Channel
PrecisionSynchro_IO	Precision Synchro	Synchro Channel
I2C_IO	Inter-Integrated Circuit	Address
Generic_IO	Other architectures	N/A

4.5.1 I/O Services Interface Requirements

1. An I/O Service UoC shall provide the IOS Interface as specified in Appendix C.
2. An I/O Service UoC shall implement the IOS Interface according to the IDL to Programming Language Mappings defined in Section 4.14.

4.6 Platform-Specific Services Segment

The PSSS creates an infrastructure unique to the platform that provides device data to the components located in the PCS. The components of the PSSS may be portable and may be reused between platforms that share the corresponding platform devices. The PSSS is broken into three sub-segments:

- Platform-Specific Device Services
- Platform-Specific Common Services
- Platform-Specific Graphics Services

Only components that meet the requirements of one of these three sub-segments reside within the PSSS.

The PSDS sub-segment is made up of components to remove platform device-unique variability from the PCS. The components of the PSDS act as software abstractions of platform hardware devices to provide data and control capabilities to the PCS.

The PSCS sub-segment defines a set of service components that may be implemented to include Logging, Device Protocol Mediation (DPM) Services, Streaming Media Services, Configuration Services, and System Level Health Monitoring. The PSCS sub-segment contains only service types defined in Section 4.6.3 and cannot be augmented without updating the FACE Technical Standard.

The PSGS sub-segment provides a set of graphics services to the PCS. The graphics services provided vary by platform requirements and are selected by the system integrator.

A notional FACE Reference Architecture is shown in Figure 10. The figure includes PSSS sub-segments, UoCs, and interfaces.

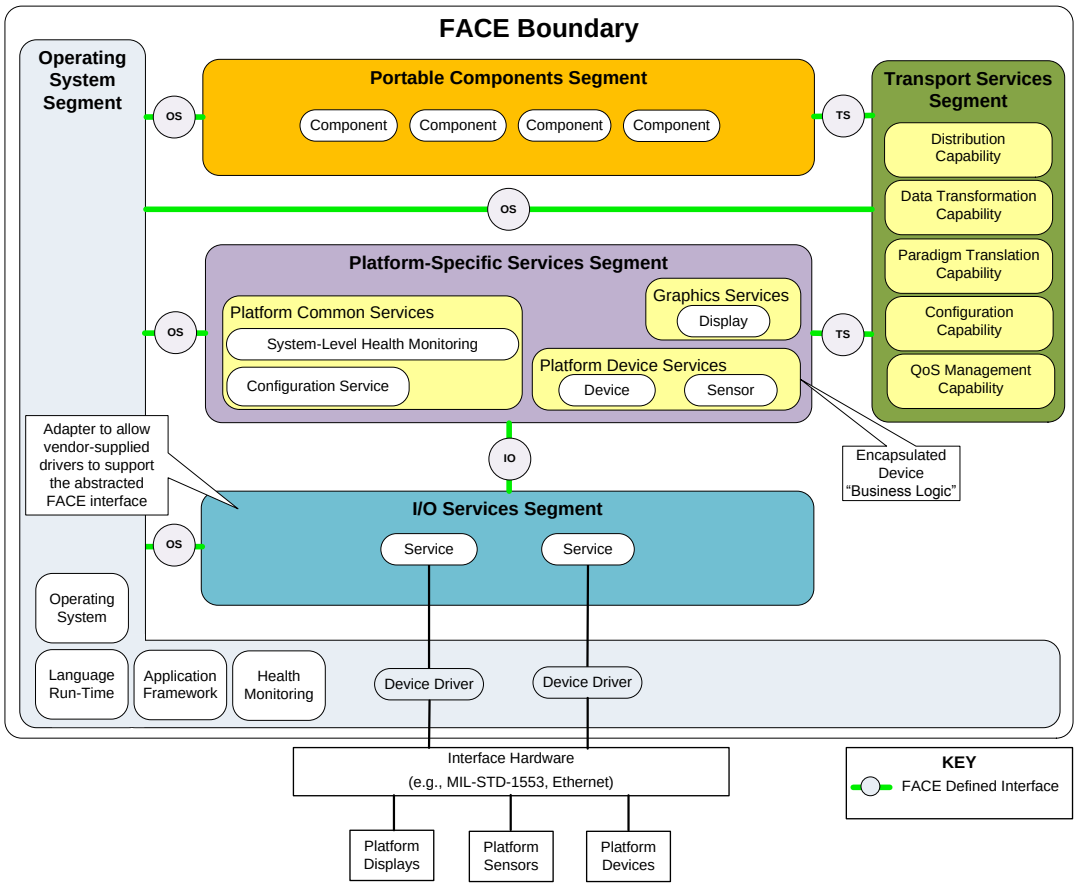

Figure 10: Notional Platform-Specific Services Segment

4.6.1 Platform-Specific Services Segment Requirements

1. A PSSS UoC shall only use the interfaces defined in Section 4.2, Section 4.5, Section 4.8, Section 4.12, and Section 4.13.
2. When a PSSS UoC uses multiple POSIX processes, the PSSS UoC shall use the POSIX multi-process APIs as defined in Section 4.2.1.
3. When a PSSS UoC uses the OSS Interface, the PSSS UoC shall adhere to the restrictions specified in Section A.6 and Section A.7.
4. A PSSS UoC, with the exception of DPM, shall communicate with PCS and PSSS UoCs through the TS Interface.
5. All data communicated over the TS Interface shall be defined by the FACE Data Architecture in accordance with requirements in Section 4.9.
6. A Connection element "name" property shall be a case-insensitive string.
7. The Connection name provided to TSS UoCs to create a connection shall match the "name" property of the corresponding Connection element in the UoC's USM.

   Note: The USM may contain a default name which could be overridden by configuration data.

8. A PSSS UoC shall use the IOS Interface defined in Section 4.5 to communicate with an IOSS UoC.

9. A PSSS UoC shall exist entirely in a single PSSS sub-segment.

10. A PSSS UoC that uses a Programing Language Run-Time shall use the Programming Language Run-Time Interfaces defined in Section 4.2.3.

    Note: If a Programming Language Run-Time is implemented in the PSSS as part of the UoC, then the Programming Language Run-Time must conform exclusively to a subset of the following interfaces for all data exchange crossing the UoC boundary per FACE segment requirements:

    — TS Interface

    — IOS Interface

11. A PSSS UoC that uses a Component Framework shall use the Component Framework Interface defined in Section 4.2.4.

    Note: If a Component Framework is implemented in the PSSS as part of the UoC, then the Component Framework must conform exclusively to any subset of the following interfaces for all data exchange crossing the UoC boundary per the segment requirements:

    — TS Interface

    — IOS Interface

12. When implementing a Component Framework, a PSSS UoC shall do so in accordance to the requirements in Section 4.6.1.2.

13. When a PSSS UoC retrieves Configuration Information, the UoC shall use the Configuration API as defined in Section 4.2.5.

14. When using Centralized Configuration, a PSSS UoC shall use the TSS API.

    Note: Section 4.6.3.1.4 describes the Centralized Configuration Service.

15. When a PSSS UoC uses Data Stores, the PSSS UoC shall use the TS Interface to store and retrieve data.

16. When a PSSS UoC uses the TSS Component State Persistence (CSP) Capability, it shall be in accordance with the CSP Interface defined in Section 4.8.3.

17. When a PSSS UoC uses CSP, the PSSS UoC shall use the CSP Interface defined in Section E.3.5 to store and retrieve its Checkpoint Data.

18. When the Injectable Interface is provided by a PSSS UoC, the Injectable Interface shall be in accordance with Section 4.11.4.1.

    Note: The Injectable Interface is used to resolve the interface dependency between UoCs.

4.6.1.1 PSSS UoC Life Cycle Management Services Requirements

1. When providing a LCM Services Interface, a PSSS UoC shall do so in accordance with the requirements of Section 4.13.

2. When using a LCM Services Interface, a PSSS UoC shall do so in accordance with the requirements of Section 4.13.

4.6.1.2 Component Frameworks Provided as Part of a PSSS UoC Requirements

FACE requirements allow the use of Component Frameworks as integral parts of PSSS UoCs as long as the libraries are FACE aligned and the entire Component Framework is provided as part of a conformant PSSS UoC. There are no specific requirements to use Component Frameworks as integral parts of PSSS UoCs.

1. When exchanging data using a framework, a PSSS UoC shall use the TS Interface.
2. When accessing framework configuration interfaces, a PSSS UoC shall use the FACE Configuration Interface.
3. When accessing framework device drivers, a PSSS UoC shall use the IOS Interface.
4. When storing private and checkpoint data, a PSSS UoC shall use the CSP Interface.
5. When accessing framework capabilities not listed in requirements 1-4 (i.e., persistent storage, time interfaces, logging), a PSSS UoC shall use the TS Interface.

   Note: A PSSS UoC must use the FACE TS Interface (*Send_Message*(TS)) to access framework persistent storage create and update interfaces.

   Note: A PSSS UoC must use the FACE TS Interface (*Send_Message*(TS)) to access framework persistent storage request interfaces.

   Note: A PSSS UoC must use the FACE TS Interface (*Receive_Message*(TS)) to access framework persistent storage response interfaces.

   Note: A PSSS UoC must use the FACE TS Interface (*Receive_Message*(TS) to access framework time get time interfaces.

   Note: A PSSS UoC must use the FACE TS Interface (*Send_Message*(TS)) to access framework time set time interfaces.

   Note: A PSSS UoC must use the FACE TS Interface to access framework error and logging interfaces.
6. When a Component Framework is implemented as part of a PSSS UoC, the Component Framework shall use the Initializable Capability of the LCM Services to initialize an instance of a PSSS UoC.
7. When a Component Framework is implemented as part of a PSSS UoC, the Component Framework shall use the Initializable Capability of the LCM Services to finalize an instance of a PSSS UoC.
8. When a Component Framework is implemented as part of a PSSS UoC, the Component Framework shall use the Configurable Capability of the LCM Services to configure an instance of a PSSS UoC.
9. When a Component Framework is implemented as part of a PSSS UoC, the Component Framework shall use the Connectable Capability of the LCM Services to connect an instance of a PSSS UoC.

10. When a Component Framework is implemented as part of a PSSS UoC, the Component Framework shall use the Connectable Capability of the LCM Services to disconnect an instance of a PSSS UoC.

11. When a Component Framework is implemented as part of a PSSS UoC, the Component Framework shall use the Stateful Capability of the LCM Services to query the state of an instance of a PSSS UoC.

12. When a Component Framework is implemented as part of a PSSS UoC, the Component Framework shall use the Stateful Capability of the LCM Services to change the state of an instance of a PSSS UoC.

4.6.1.3 PSSS Security Transformation Requirements

Security Transformations perform transformations of data for security purposes as described in Section 5.2.2. The FACE Technical Standard does not specify or constrain where transformations are performed.

1. When a Security Transformation is implemented as part of a PSSS UoC, all data crossing the Security Transformation boundary shall be defined in accordance with the FACE Data Architecture in Section 4.9.

   Note: Recommend the Security Transformation uses a TS Interface when traversing the transform boundary internal to the PSSS.

   Note: Given the sensitivity of the internal interface data model, there may be restrictions on availability and distribution of the detailed data models levied by the platform and/or security-relevant transform supplier.

2. When a Security Transformation is implemented as part of a PSSS UoC, the characterization of the transformation shall include a detailed description of the transformation.

   Note: The detailed description of the Security Transformation should be sufficient to enable interoperability with similar transformations.

   Note: Given the sensitivity of the transformation characterization data, there may be restrictions on availability and distribution of the detailed data models levied by the platform and/or security-relevant transform supplier.

4.6.2 Platform-Specific Device Services

The Platform-Specific Device Services (PSDS) sub-segment is made up of UoCs to abstract platform device-unique variability from the PCS. This sub-segment may include UoCs that provide control for, receive data from, and send data to platform devices or external systems. It may also contain Legacy Operational Flight Program (OFP) adapters or other Platform-Specific software components used to provide integration support for other software components on the platform.

4.6.2.1 Platform-Specific Device Services Requirements

PSDS UoCs communicate with platform devices using data as defined by the associated ICD and may possess the ability to resequence the messages.

1. A PSDS UoC shall perform the translation of data between the TS Interface and IOS Interface.

2. A PSDS UoC shall use the IOS Interface to access an I/O device.

   Note: The functionality on either side of the IOS Interface is dependent on developer and integrator implementation. I/O Device control can be allocated to the I/O Service, thus making the PSDS more portable.

3. When a PSDS UoC communicates with a DPM Service, it shall use the IOS Interface defined in Appendix C.

4.6.3 Platform-Specific Common Services

The Platform-Specific Common Services (PSCS) sub-segment provides services to other FACE segments per system requirements. The PSCS communicate using the TS Interface to the PCS and the IOS Interface to the IOSS. The PSCS support Logging, Device Protocol Mediation (DPM), Streaming Media, Centralized Configuration Services, and System Level Health Monitoring, as defined in Sections 4.6.3.1.1, 4.6.3.1.1.2, 4.6.3.1.3, 4.6.3.1.4, and 4.6.3.1.5 respectively. The PSCS sub-segment defines an exhaustive list of software components that may be implemented in a FACE Reference Architecture:

- Logging
- Device Protocol Mediation
- Streaming Media
- Centralized Configuration Services
- System Level Health Monitoring

4.6.3.1 Platform-Specific Common Services Requirements

1. A PSCS UoC shall use the IOS Interface to access an I/O device.

   Note: The functionality on either side of the IOS Interface is dependent on developer and integrator implementation. I/O Device control can be allocated to the I/O Service, thus making the PSCS more portable.

2. Communication between a PSCS UoC and an IOSS UoC shall use the IOS Interface.

3. Communication between a PSCS UoC and software components of the PSSS, TSS, and PCS shall use the TS Interface.

4.6.3.1.1 Logging Services Requirements

Logging services fall into one of two categories:

- Centralized
- Localized

4.6.3.1.1.1 Centralized Logging

1. When a centralized logging service is provided, it shall exchange data over the TS Interface formatted in accordance with IETF RFC 5424: The Syslog Protocol.

2. When a centralized logging service is provided, it shall exchange data over the IOS Interface formatted in accordance with IETF RFC 5424: The Syslog Protocol.

   Note: Faults are logged by the Health Monitoring and Fault Manager described in Section 4.1.3.

4.6.3.1.1.2 Localized Logging

Localized logging is handled within the UoC according to the individual UoC's implementation. Software suppliers may choose to implement a localized logging method.

4.6.3.1.2 Device Protocol Mediation Requirements

DPM services are UoCs of the PSCS sub-segment acting as a protocol mediator for platform devices using transport protocols (e.g., SNMP, SNMP V3, HTTP, HTTPS, FTP, and SFTP) supported by the OSS Interface. The DPM Service is only accessible by UoCs of the PSDS sub-segment of the PSSS. See Figure 11 for a visual depiction of DPM services.

1. The DPM Service shall communicate with UoCs of the PSDS sub-segment of the PSSS through the IOS Interface defined in Appendix C.

2. The DPM Service shall communicate with the platform device using data as defined by the platform device ICD.

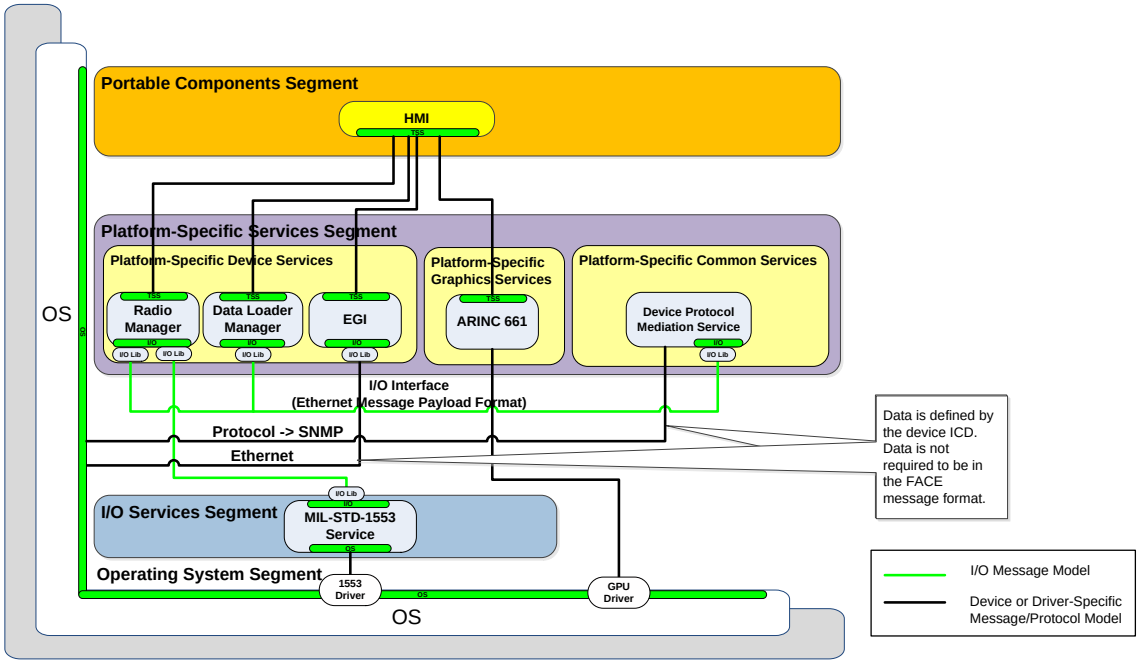

Figure 11: DPM Example

4.6.3.1.3 Streaming Media Requirements

Streaming Media Services are UoCs of the PSCS sub-segment acting as a streaming media adapter for platform imaging devices using media protocols (e.g., MPEG Formats, SMPTE-292). See Figure 12 for a visual depiction of Streaming Media Services.

1. A Streaming Media Service UoC shall communicate with the streaming media driver.

Note: When hardware-accelerated decoding of streaming media protocols and formats is required, the Streaming Media Service may have access to streaming media hardware drivers.

Note: Recommended for low latency high bandwidth implementations.

2. A Streaming Media Service UoC shall exchange data over the TS Interface.

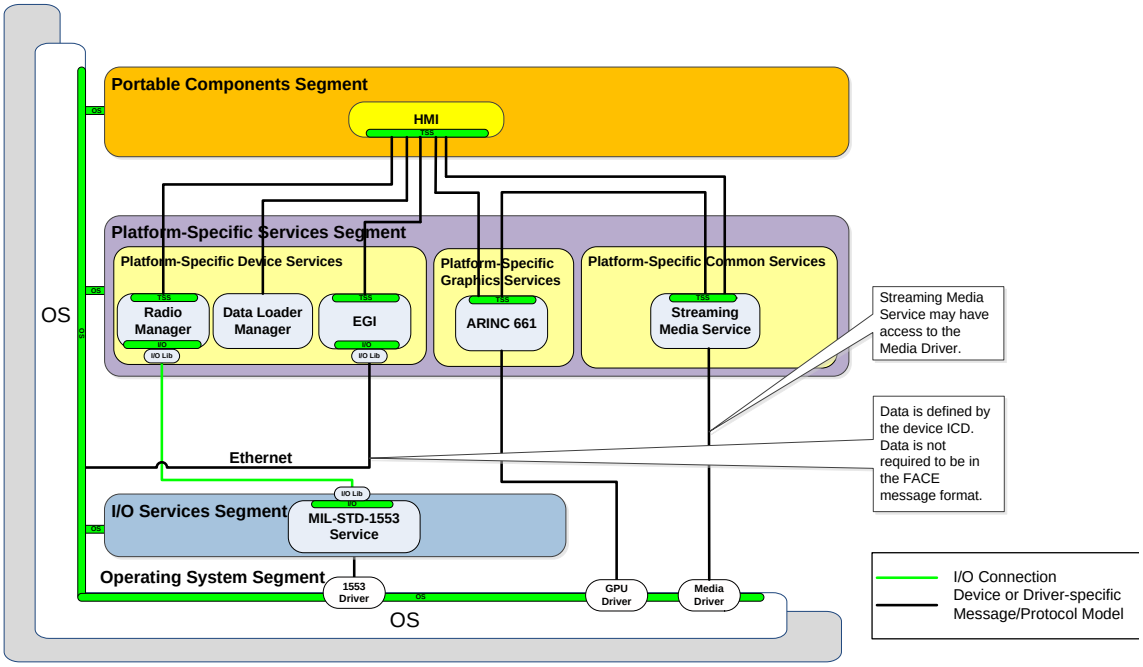

Figure 12: Streaming Media Services Notional Example

4.6.3.1.4 Centralized Configuration Service Requirements

The primary purpose of the Centralized Configuration Service is to enable sharing of configuration information among system components and services.

1. The Centralized Configuration Service shall provide a mechanism to provide configuration information for the following FACE segments:

    a. PCS

    b. TSS

    c. PSSS

    Note: The Centralized Configuration Service provides a mechanism for a software component to make configuration information available to other components.

    Note: The Centralized Configuration Service is intended to communicate with software components residing in the PSSS, TSS, and PCS, using the TS Interface.

    Note: The Centralized Configuration Service is intended to communicate with software components residing in the IOSS, using the IOS Interface.

2. When a Centralized Configuration Service uses the Configuration Interface, it shall be as defined in Appendix G.

4.6.3.1.5 System Level Health Monitoring Requirements

The primary purpose of the System Level Health Monitor is monitoring and reporting system and application faults and failures. The System Level Health Monitor configuration may be administratively viewable at run-time. The System Level Health Monitor may support one or more redundancy models as fault recovery and avoidance mechanisms. The System Level Health Monitor may support the "repair" option, such that HMFM attempts to resurrect failed or faulted resources. The System Level Health Monitor may monitor/manage the instantiation and termination of components. The System Level Health Monitor may generate alarms and notifications to indicate internal state transitions experienced by the components.

1. The System Level Health Monitor shall use the HMFM Interface defined in Section 4.2.2.

4.6.4 Platform-Specific Graphics Services

The Platform-Specific Graphics Services (PSGS) sub-segment provides a set of graphics services to the PCS. The graphics services provided vary by platform requirements and are selected by the system integrator. See Section 4.12.9.

4.6.4.1 Platform-Specific Graphics Services Requirements

1. When communicating with the graphics driver, the Graphics Services UoC shall do so in accordance with requirements in Section 4.12.9.

4.7 Transport Services Segment

4.7.1 Introduction

The Transport Services Segment (TSS) abstracts data access and access to common technical functions and facilitates integration of PCS and PSSS software components into disparate architectures and platforms. UoCs within the TSS provide capabilities related to data access. They also provide standardized interfaces for UoCs located in the PCS and PSSS. Support capabilities specified within the TSS may include distribution, routing, prioritization, addressability, association, abstraction, and transformation of software component information. In the FACE Technical Standard, some of the TSS support capabilities are identified, and requirements are allocated to these capabilities. There are additional support capabilities that fit into the TSS that are not individually identified and do not have requirements for implementation. An example of a capability that may be allocated to TSS UoCs is specific security concerns such as encryption of data, or enforcement of access control. The TSS Interfaces are defined in Section 4.8 and Appendix E.

Figure 13 depicts the TSS support capabilities, and the inter-segment and intra-segment interfaces.

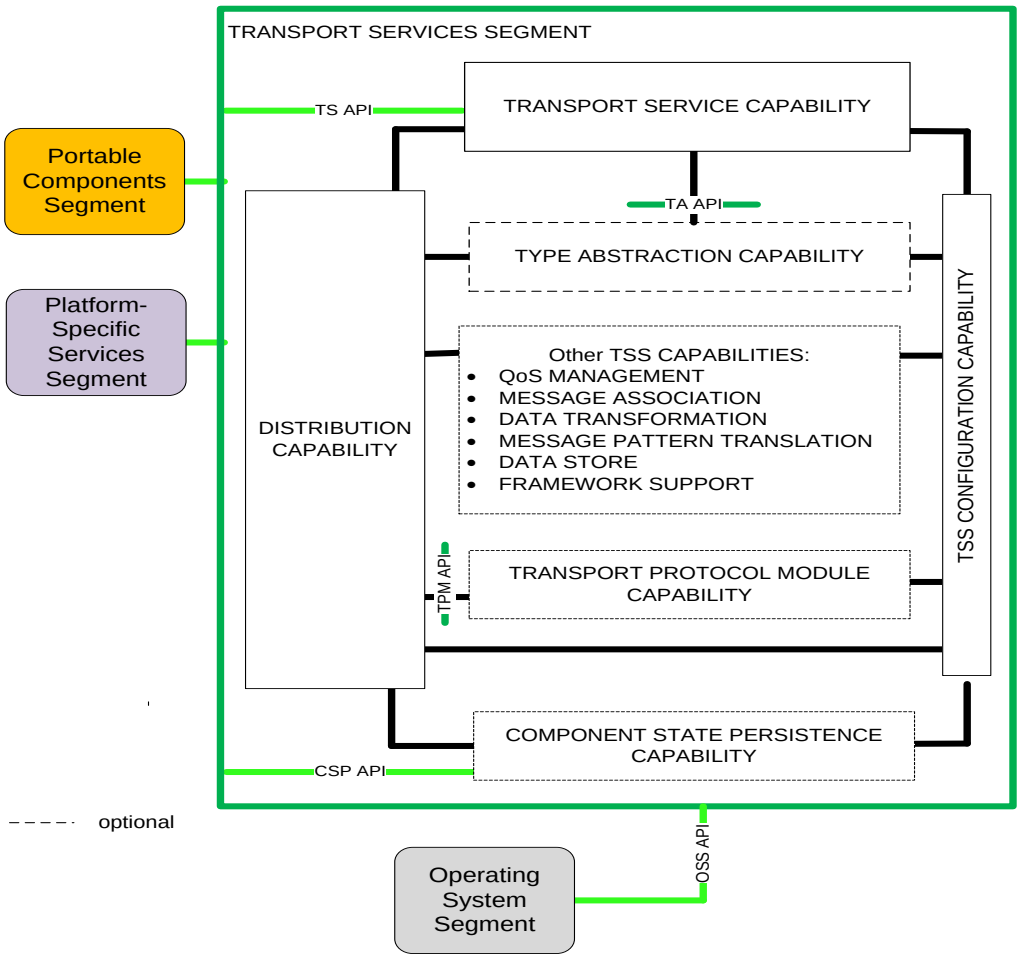

Figure 13: Transport Services Segment Capabilities

TSS UoCs support data access to a variety of common technical services. The TSS UoCs provide these common technical services to a PSSS or PCS UoC through the TSS Inter-segment Interfaces: Transport Service (TS) API, and the Component State Persistence (CSP) API. TSS UoCs provide data transport between PCS/PSSS UoCs, as well as mediation between messages with different data models. TSS UoCs support a variety of data transport protocols and messaging patterns. Additionally, TSS UoCs provide access to persistent Data Stores and the ability for UoCs to checkpoint their internal state. The TSS includes support for the capabilities identified below, and shown in Figure 13:

- Transport Service Capability (Section 4.7.3)
- Distribution Capability (Section 4.7.4)
- Configuration Capability (Section 4.7.5)
- Type Abstraction Capability (Section 4.7.6)
- QoS Management Capability (Section 4.7.7)
- Message Association Capability (Section 4.7.8)

- Data Transformation Capability (Section 4.7.9)
- Messaging Pattern Translation Capability (Section 4.7.10)
- Transport Protocol Module (TPM) Capability (Section 4.7.11)
- Data Store Support Capability (Section 4.7.12)
- Component State Persistence Capability (Section 4.7.13)
- Framework Support Capability (Section 4.7.14)

Within a system composed from UoCs, it is not a requirement that all TSS capabilities be supported. Different TSS UoCs may provide the whole set or a subset of TSS capabilities including but not limited to Quality of Service (QoS) Management, Message Association, Data Transformation, and Message Pattern Translation. The TSS capabilities help isolate PCS/PSSS data architecture variances and messaging to the TSS. To achieve the minimum capability set required to function as a TSS, the UoCs collectively comprising the TSS provide Transport Service Capability (TS Capability) with the type-specific interface, the Distribution Capability, and the TSS Configuration Capability. Providing additional TSS capabilities depends on requirements for a given instance of the TSS UoCs.

The TSS Intra-segment Interfaces are provided and used by modules, libraries, or software components that form TSS UoCs. Intra-segment interfaces provide points of conformance testing that support the creation of UoCs implementing those interfaces. A TSS then can be composed of independently created TSS UoCs which interoperate. The TSS Intra-segment Interfaces cannot be used by PCS or PSSS UoCs.

The Intra-segment Interfaces that may be used to form a proper subset of capabilities into TSS UoCs include:

- Type Abstraction (TA) API
- Transport Protocol Module (TPM) API

Section 4.7.1.1 provides the sets of TSS Capabilities that can be used to form a TSS UoC.

PCS and PSSS UoCs define messages within the FACE Data Architecture to derive a type-specific interface for the TSS. The FACE Data Architecture is described in Section 4.9.

4.7.1.1 TSS UoC Conformance Options

A TSS provides TSS Inter-segment Interfaces such as the type-specific interface used by PCS or PSSS UoCs and the Component State Persistence Interface to control PCS or PSSS UoCs. A TSS UoC can base its implementation on providing only the Inter-segment Interfaces. However, to enable portability of TSS libraries, TSS UoCs can also be built against subsets of TSS segment requirements when they provide intra-segment interfaces. UoCs providing intra-segment interfaces do not provide the complete set of TSS capabilities required for the intended system design.

For example, a UoC can realize the Type Abstraction capability and provide the TA API. This UoC, a Type Abstraction UoC, does not provide the type-specific interface nor realize the Transport Service Capability. To incorporate this Type Abstraction UoC, the system design would also include a type-specific interface to type abstraction interface adapter as its own UoC.

As another example, a UoC can provide the TPM API and realize the Transport Protocol Module Capability to support interoperability between non-homogeneous TSS implementations in the system. Table 7 summarizes the sets of TSS Capabilities used to form different TSS UoCs.

Table 7: Sets of TSS Capabilities that Form a TSS UoC

	Units of Conformance					
TSS Capabilities	**TS**	**TA**	**TS-TA Interface Adapter**	**TPM**	**CSP**	**FS**
TS Capability	Required		Required			
TSS Distribution Capability	Required	Required				
TSS Configuration Capability	Required	Required	Optional	Optional	Optional	Optional
Type Abstraction Capability		Required				
QoS Management Capability	Optional	Optional				
Message Association Capability	Optional	Optional				
Data Transformation Capability	Optional	Optional	Optional			
Message Pattern Translation Capability	Optional	Optional				
Transport Protocol Module Capability				Required		
Data Store Support Capability	Optional	Optional				
Component State Persistence Capability					Required	
Framework Support Capability	Optional	Optional				Required

Note: "Required" indicates the capability is required by the UoC. "Optional" indicates the capability is optional for the UoC.

Note: As discussed in Section 4.11.3, TSS UoCs provide an Injectable Interface for each FACE Interface declared by IDL that it uses.

4.7.2 Transport Services Segment Requirements

The TSS is a logical grouping of software components that provide access to data and other common technical services. The following are requirements placed upon software components intended to implement capabilities allocated to the segment:

1. A TSS UoC shall provide one or more of the following capabilities:

    a. Transport Service Capability

    b. TSS Distribution Capability

    c. TSS Configuration Capability

    d. Type Abstraction Capability

    e. QoS Management Capability

    f. Message Association Capability

    g. Data Transformation Capability

    h. Messaging Pattern Translation Capability

    i. Transport Protocol Module Capability

    j. Data Store Support Capability

    k. Component State Persistence Capability

    l. Framework Support Capability

    Note: The aggregate UoCs within the TS segment must provide at least the TS Capability, TSS Distribution, and Configuration Capabilities.

2. When a TSS UoC uses multiple POSIX processes, the TSS UoC shall use the POSIX multi-process APIs as defined in Section 4.2.1.

3. A TSS UoC shall use the OS Interface as specified by the applicable FACE OSS Profile defined in Section 4.2.1.

    Note: When a Programming Language Run-Time is implemented in the TSS as part of the UoC, then the Programming Language Run-Time must conform exclusively to a subset of the following interfaces for all data exchange crossing the UoC boundary per the segment requirements:

    — TS Interface

    — OSS Interface

    Note: When a Component Framework is combined with a software component as part of a TSS UoC, then the Component Framework must conform exclusively to a subset of the following interfaces for all data exchange crossing the TSS boundary per the segment requirements for the following:

    — TS Interface

    — OSS Interface

— CSP Interface

4. When using Programming Language Run-Times, a TSS UoC shall do so in accordance with Section 4.2.3.

5. When a TSS UoC uses a Component Framework, the TSS UoC shall use the Component Framework in accordance with Section 4.2.4.

6. When a TSS UoC uses the OSS Interface, the TSS UoC shall adhere to the restrictions specified in Section A.6 and Section A.7.

7. When using OSS Health Monitoring, a TSS UoC defined to operate in a POSIX operational environment shall use the OSS HMFM Interface described in Section 4.2.2.

8. When a TSS UoC retrieves Configuration Information locally, the UoC shall use the Configuration API as defined in Section 4.2.5.

9. When using Centralized Configuration, a TSS UoC shall use the TSS API.

   Note: Section 4.6.3.1.4 describes the Centralized Configuration Service.

10. When storing Private or Checkpoint Data, a TSS UoC shall use the CSP Interface as defined in Section 4.8.4.2 to store the private and checkpoint data.

11. When providing a LCM Services Interface, a TSS UoC shall do so in accordance with the requirements of Section 4.13.

12. When using a LCM Services Interface, a TSS UoC shall do so in accordance with the requirements of Section 4.13.

13. When the Injectable Interface is provided by the TSS UoC, the Injectable Interface shall be in accordance with Section 4.11.4.1.

14. A TSS UoC implementing the TPM Capability shall directly access the device and/or use device driver interfaces for interaction with the device when access is not possible through an OS Interface.

15. A TSS UoC implementing the CSP Capability shall directly access the device and/or use device driver interfaces for interaction with the device when access is not possible through an OS Interface.

4.7.3 Transport Service Capability

The Transport Service Capability provides a type-specific interface, and is responsible for providing the functionality of the Type-Specific Interface to UoCs. The Transport Service Capability provides the interface for data exchange.

1. A Transport Service Capability shall provide the TS Interface as defined in Section 4.8.4.1.

2. A Transport Service Capability shall supply a library for PCS and PSSS UoCs to use in all cases.

3. When communicating with the Type Abstraction Capability, the Transport Service Capability shall use the Type Abstraction Interface.

4. A Transport Service Capability shall be configured using the TSS Configuration Capability.

5. Transport Service Capability configuration data shall be specified in accordance with the Configuration Services requirements in Section 4.2.5.

4.7.4 Transport Services Segment Distribution Capability Requirements

The Distribution Capability controls or manages the distribution of data within a TSS. Since a TSS UoC may require several capabilities in addition to the basic physical exchange of data between endpoints, it is useful to allocate the requirements for management and control of those additional capabilities to a single TSS capability. It provides support functionality for data marshalling and transformations which may be used by other TSS capabilities within a TSS UoC.

1. A Distribution Capability shall be configured using the TSS Configuration Capability.

2. A Distribution Capability configuration data shall be specified in accordance with the Configuration Services requirements in Section 4.2.5.

3. A Distribution Capability shall populate the TS Header Parameter Instance from data transmitted by a UoC Output EndPoint for received messages.

4. A Distribution Capability shall provide the data contained in the TS Header Parameter Instance to the receiving endpoint for sent messages.

5. When multiple independent message patterns are implemented, a Distribution Capability shall be configurable to accommodate multiple independent message patterns.

6. A Distribution Capability shall accept data from data producers and distribute it to data consumers based on the configuration information.

7. A Distribution Capability message connection shall be a case-insensitive named entity.

8. A Distribution Capability shall use the Message Association Capability to manage information about data messaging associations and tagging.

9. When transformations are performed, a Distribution Capability shall use one or more of the Data Transformation Capabilities to perform the following functions:

    a. Data transformations

    b. Data marshalling

10. When message pattern translations are performed, a Distribution Capability shall use Message Pattern Translation Capabilities to accommodate translations between disparate paradigm solutions (e.g., Publish/Subscribe to Request/Reply).

11. When exchanging data between TS Domains, a Distribution Capability shall use the TPM Interface to communicate with a TPM Capability.

12. When communicating between TS Domains, a Distribution Capability shall provide the selection of a protocol from one or more TPMs.

4.7.5 Transport Services Segment Configuration Capability Requirements

The TSS Configuration Capability is responsible for managing the configuration of a TSS UoC. It uses the Configuration Services (either centralized or distributed) to obtain the necessary TSS UoC configuration data, and it then manages configuration of TSS Capabilities. TSS Configuration Capability enables capabilities (during design, compile, link, initialization, or execution time, either static or dynamic), that are implemented by various software modules, or through the loading of libraries as required by different TSS UoC designs.

1. A TSS Configuration Capability shall provide configuration data as described by the FACE Configuration Services defined in Section 4.2.5.
2. A TSS Configuration Capability shall include configuration information for the following capabilities when the respective capability is provided:
    a. Transport Service Capability
    b. Distribution Capability
    c. Type Abstraction Capability
    d. QoS Management Capability
    e. Message Association Capability
    f. Data Transformation Capability
    g. Message Pattern Translation Capability
    h. Transport Protocol Module Capability
    i. Data Store Support Capability
    j. Component State Persistence Capability
    k. Framework Support Capability
3. A TSS Configuration Capability shall use the Configuration Interface to the Configuration Services as defined in Section 4.2.5 to access configuration data.
4. A TSS Configuration Capability shall support TPM library references being set.
5. When a TPM library reference has been set which uses serialization interfaces, a TSS Configuration Capability shall provide the *FACE::TSS::Serialization* interface to allow TPMs to access message serialization functions.
6. When a TPM library reference has been set which uses serialization interfaces, a TSS Configuration Capability shall provide a type specific function with the *FACE::TSS::Message_Serialization* interface for each message the TPM is to serialize or deserialize.
7. When a TSS Configuration Capability provides message serialization, each message serialization function provided shall serialize each element of the message in the order of the structure defined by the associated UoPModel Template as specified in Section J.2.5.
8. When a TPM provides the primitive marshalling interface, the message serialization function provided by a TSS Configuration Capability shall use the TPM primitive marshalling interface to serialize base types as defined in Section 4.7.11.2.

9. When a TSS Configuration Capability provides message deserialization, each message deserialization function provided shall deserialize each element of the message in the order of the structure defined by the associated UoPModel Template as specified in Section J.2.5.

10. When a TPM provides the primitive marshalling interface, the message deserialization function provided by a TSS Configuration Capability shall use the TPM primitive marshalling interface to deserialize base types in accordance with requirements in Section 4.7.11.2.

11. When a TPM library reference has been set which uses serialization interfaces, the *FACE::TSS::Message_Serialization* interface shall meet the specification of Section E.3.3.

12. When a TPM library reference has been set which uses serialization interfaces, Serialization interface shall meet the specification of Section E.3.3.

13. When TPM library references have been set, a TSS Configuration Capability shall use the TPM's *initialize*() function to initialize the TPM.

14. When TPM library references have been set, a TSS Configuration Capability shall manage the configuration states of the TPM libraries.

4.7.5.1 Transport Services Segment Configuration Information Requirements

Figure 14 describes the configuration data elements internal to the TSS. The data element definition used is specific to a TSS UoC and used to configure QoS, message routing, and message conversions. The integration model, described in Section 4.9.1.5, can provide the source of information within implementations for many of the TSS configuration data elements.

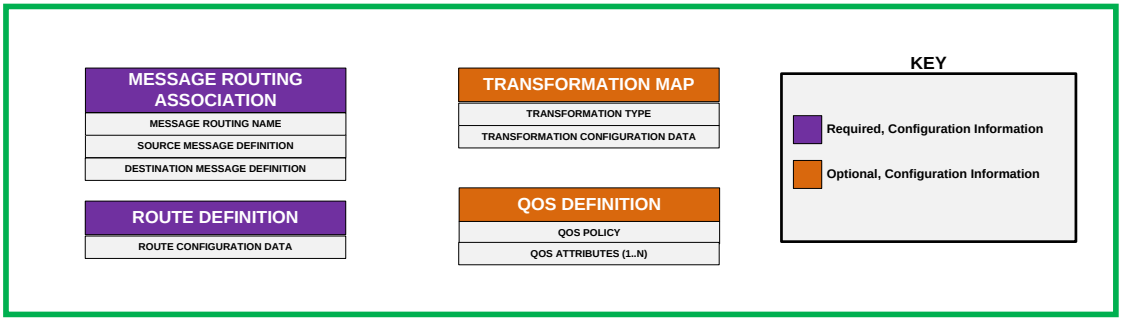

Figure 14: TSS Configuration Data Element Relationships

4.7.5.1.1 Configuration Data Elements

1. When a TSS UoC provides message routing, the TSS UoC shall contain the following Configuration Data Elements:

    a. Message Routing Association

    b. Route Definition

2. When a TSS UoC provides message routing, the TSS UoC shall contain zero or more of the following Configuration Data Elements:

    a. QoS Definition

b. Transformation Map

Note: Configuration of the data associated with the Configuration Data Elements is achieved through Configuration Services (defined in Section 4.2.5) or by statically defining the configuration data within the TSS.

4.7.5.1.2 Message Routing Association

The Message Routing Association defines the routing of messages within the system by the TSS.

Note: This is not intended to imply any particular implementation inside the TSS. Implementation details within the TSS are not standardized and are intentionally left as an area of variability within the architecture.

1. The Message Routing configuration information shall associate a Route Definition between a given Source Message Definition and Destination Message Definition pair.
2. The Message Routing configuration information shall associate a name for the route defined between a given Source Message Definition and Destination Message Definition pair.
3. When a TSS UoC configures transformations, the Message Routing configuration information shall associate a Transformation Map between a given Source Message Definition and Destination Message Definition pair.
4. When a TSS UoC configures QoS, the configuration information shall associate QoS Attributes Values for the QoS Definition which applies to the Route Definition.
5. When a TSS UoC configures message associations, the configuration information shall associate one Message Definition to another Message Definition.

   Note: The message association can be TSS configuration data supplied or an association built during run-time.

4.7.5.1.3 Route Definition

1. The Route Definition configuration information shall consist of the following data elements:

   c. Route Configuration Data

4.7.5.1.4 QoS Definition

1. The QoS Definition configuration information shall consist of the following data elements:

   a. QoS Policy

   b. QoS Attributes

4.7.5.1.5 Transformation Map

The Transformation Map configuration information provides the means to define conversions to be used by the TSS on a given Message Parameter Interface's definition entity.

1. When a TSS UoC implements Transformation Map configuration, the Transformation Map configuration information shall consist of the following data elements:

a. Transformation Type

b. Transformation Configuration Data

4.7.6 Type Abstraction Capability Requirements

Within a TSS UoC, the Type Abstraction (TA) Interface provides portability of a TSS UoC when implemented using any programming language (such as C) not supporting function overloading. While maintaining the strongly typed TS Interface, the Type Abstraction Interface provides an Intra-segment Interface for use by the TSS UoC, as shown for PCS A and PSSS B of Figure 15. The use of the Type Abstraction interface is optional, as shown for PCS C and PSSS D of Figure 15.

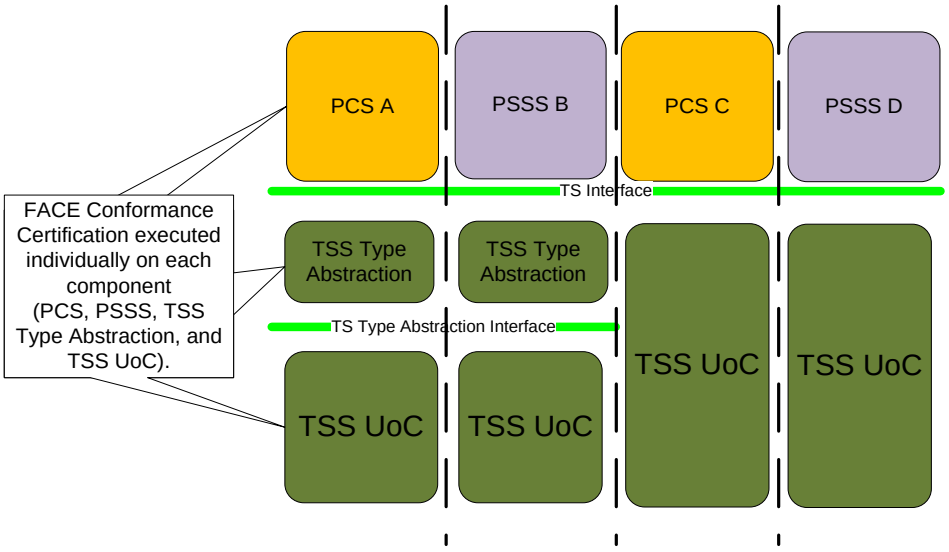

Figure 15: Type Abstraction and Interfaces Examples

1. A Type Abstraction Capability shall be configured using TSS Configuration Capability.
2. A Type Abstraction Capability's configuration data shall be specified in accordance with the Configuration Services requirements in Section 4.2.5.
3. When implementing the Type Abstraction Capability, a TSS UoC shall also provide the Distribution Capability.
4. When implementing the Type Abstraction Capability, a TSS UoC shall also provide zero or more of the following capabilities:
    a. QoS Management Capability
    b. Message Association Capability
    c. Data Transformation Capability
    d. Message Pattern Translation Capability
    e. Transport Protocol Module Capability

f. Data Store Support Capability

g. Component State Persistence Capability

h. Framework Support Capability

5. A Type Abstraction Capability shall provide the Type Abstraction Interface as specified in Section 4.7.6.1.

6. A Type Abstraction Capability shall provide the Base Interface in accordance with requirements in Section 4.8.2.

4.7.6.1 Type Abstraction Interface Requirements

The TA Interface is a TSS intra-segment interface and cannot be used by PCS or PSSS UoCs. The TA Interface is used by modules, libraries, or software components within the TSS layer.

1. The Type Abstraction Interface shall meet the specification of Section E.4.1.

2. A *FACE::TS::Base::Initialize*(TS) shall be non-blocking regardless of the underlying transport mechanism.

   Note: *Initialize*(TS) may need to complete asynchronously after *Initialize*(TA) returns if the time to complete exceeds some threshold.

3. A *FACE::TS::Base::Create_Connection*(TS) function shall be non-blocking regardless of the underlying transport mechanisms.

   Note: *Create_Connection*(TS) may need to complete asynchronously after *Create_Connection*(TS) returns if the time to complete exceeds some threshold.

4. A Type Abstraction Capability shall supply the return code value returned from the TA Interface operations as specified in Section E.4.1.

5. The Type Abstraction Capability shall use the OMG IDL to Programming Language Mappings defined in Section 4.14 for its Type Abstraction Interface.

4.7.7 QoS Management Capability Requirements

QoS refers to the collective behavior of a data exchange. This behavior is usually specified as a set of functional and non-functional parameters. Within a TSS UoC, there are several places where QoS may be enforced, or managed. These include: within the channel between two or more TSS UoCs (physical protocols), on the data as it is being exchanged, and as the data is being presented to the consuming UoC. The QoS parameters and the required values are documented in configuration data as specified in Section 4.2.5.

1. A QoS Management Capability shall be configured through information provided by the TSS Configuration Capability.

2. A QoS Management Capability configuration data shall be specified in accordance with the Configuration Services requirements in Section 4.2.5.

3. A QoS Management Capability shall manage the QoS of the messages being transported by TSS UoCs as specified in the information provided by the TSS Configuration Capability.

4.7.8 Message Association Capability Requirements

Message Association is a general capability that allows a TSS UoC to maintain an association between two data elements. Examples of use include the association of security tags to data structures, the association between a fast updating structure, and a slowly updating structure that are both part of a message delivered to a UoC, or extended metadata used by the TSS for internal optimization of transmission of a message.

1. A Message Association Capability shall be configured through information provided by the TSS Configuration Capability.

2. A Message Association Capability configuration data shall be specified in accordance with the Configuration Services requirements in Section 4.2.5.

3. A Message Association Capability shall manage message associations and data tagging in accordance with the information provided by the TSS Configuration Capability.

4.7.9 Data Transformation Capability Requirements

Data Transformation Capability performs transformations of data to deliver instances of message parameters defined by the consuming software component in accordance with their USM's UoPModel Template(s). The Data Transformation Capability includes Security Transformations performed on data for security purposes as described in Section 5.2.2, and are part of the overall TSS Data Transformation Capability. The FACE Technical Standard does not specify or constrain when transformations are performed, such that transformations can occur as data is output or as data input is received.

1. A Data Transformation Capability shall be configured through information provided by the TSS Configuration Capability.

2. A Data Transformation Capability's configuration data shall be specified in accordance with the Configuration Services requirements in Section 4.2.5.

3. A Data Transformation Capability shall manage data transformations in a TSS UoC.

4. A Data Transformation Capability shall use information from data producers to assemble instances of the Message parameter for data consumers.

5. A Data Transformation Capability shall supply data to consumers at the configured consumer request rate.

6. A Data Transformation Capability shall provide data transformations configured by the TSS Configuration Capability.

7. When a TSS UoC provides security services (e.g., encryption, labeling), a Data Transformation Capability shall perform Security Transformations.

8. When a Security Transformation is implemented as part of a TSS UoC, all data crossing the Security Transformation boundary shall be defined in accordance with the FACE Data Architecture in Section 4.9.

   Note: Given the sensitivity of the internal interface data model, there may be restrictions on availability and distribution of the detailed data models levied by the platform and/or security-relevant transform supplier.

9. When using a Security Transformation, the characterization of the transformation shall include detailed description of the transformation.

   Note: The detailed description of the Security Transformation should be sufficient to enable interoperability with similar transformations from an independent source.

   Note: Given the sensitivity of the transformation characterization data, there may be restrictions on availability and distribution of the detailed data models levied by the platform and/or security-relevant transform supplier.

4.7.10 Messaging Pattern Capability Requirements

Messaging Pattern Capabilities manage the message pattern, message communication types, transformation from one message pattern to another message (e.g., Request/Reply to Publish/Subscribe), and other message processing requirements.

1. A Messaging Pattern Capability shall be configured through information provided by the TSS Configuration Capability.
2. A Messaging Pattern Capability's configuration data shall be specified in accordance with the Configuration Services requirements in Section 4.2.5.
3. A Messaging Pattern Capability shall provide transformation from one distinct message pattern to another distinct message pattern.
4. A Messaging Pattern Capability shall instantiate independent Message Pattern Translations individually.
5. A Messaging Pattern Capability shall translate message patterns in conjunction with other independent translation capabilities within a distributed TSS instantiation.
6. A Messaging Pattern Capability shall perform transformations from one message pattern to another message pattern (e.g., Publish/Subscribe to Request/Reply).
7. A Messaging Pattern Capability shall provide for instantiation of one or more of the following message patterns:
   a. Publish/Subscribe
   b. Request/Reply

   Note: Request/Reply is also used as a synonym for Command/Response.
8. A Messaging Pattern Capability shall provide for instantiation of one or more of the following message communication types:
   a. Synchronous Communications
   b. Asynchronous Communications
9. A Messaging Pattern Capability shall provide for instantiation of one or more of the following communication mechanisms as allowed by the FACE OS Interface based upon the applicable FACE OSS Profile:
   a. Networking Standards (e.g., POSIX sockets)
   b. Cache

c. Shared Memory

10. A Messaging Pattern Capability shall provide for instantiation of one or more of the following message structures:

    a. Fixed Length

    b. Variable Length

11. A Messaging Pattern Capability shall process messages using one or more of the following message processing types:

    a. Periodic

    b. Sporadic

    c. Aperiodic

12. A Messaging Pattern Capability shall provide one or more of the following message distribution types:

    a. Broadcast

    b. Multicast

    c. Unicast

    d. Anycast

4.7.11 Transport Protocol Module Capabilities Requirements

As systems built from UoCs become more prevalent, it becomes necessary to connect one system to another. The ability to do this without modification to the software components of either system, including the TSS Support Components, helps meet the goals of UoCs. TS to TS Interoperability across TS Domains is achieved by managing and abstracting three levels of system interaction. First is a semantic understanding of the exchanged data, second is a syntactic understanding of the method of exchange, and finally is a technical understanding of the physical mode of exchange.

In the FACE Technical Standard, the semantic understanding is provided by the FACE Data Architecture. The syntactic and technical interoperability is provided, not by an interoperability protocol, but by abstracting the interface used to access protocols. This allows for flexibility and specialization of protocols to meet the varied system requirements that are imposed upon systems intended to be built from UoCs. The FACE TPM Capability allows the selection between multiple protocols depending on the connectivity requirements. The TPM Capability supports multiple protocols, such as TCP, UDP, RTPS, IIOP, Queues, Inter-Process Communication, and inter-partition communication as well as physical transports such as Internet Protocol and Shared Memory using the OS Interface. Additional physical transports and protocols may be supported through device drivers as described in Section 4.3.

1. A TPM Capability shall provide one or more TPM(s) for data exchange between TS Domains to achieve interoperability.

2. A TPM Capability shall perform data transport between one instance of a TSS UoC and another instance of a TSS UoC.

3. A TPM shall be configured using information provided by the TSS Configuration Capability through the TPM *Initialize*() function.

4. TPM configuration data shall be specified in accordance with the Configuration Services requirements in Section 4.2.5.

5. A TPM shall provide a protocol-specific serialization function for the IDL described base types.

6. A TPM shall provide a protocol-specific deserialization function for the IDL described base types.

7. A TPM shall serialize data to send on its transport protocol.

   Note: A TPM may use its own serialization function or the type-specific serialization interface provided by the TS Capability.

8. A TPM shall deserialize data received from its transport protocol.

   Note: A TPM may use its own deserialization function or the type-specific deserialization interface provided by the TS Capability.

9. A TPM shall supply the data contained in a TS message header to the receiving endpoint.

10. A TPM shall populate the TS message header from data transmitted by a sending TPM.

11. A TPM shall populate a QoS Data Structure with the actual QoS parameter values of the received TS message instance.

12. A TPM shall return the most current instance of the received TS message payload to the QoS Management capability.

4.7.11.1 Intra-Segment Transport Protocol Module Interface

The TSS Intra-segment Interfaces are provided and used by modules, libraries, or software components within the TSS layer. Intra-segment Interfaces support the creation of UoCs implementing those interfaces. The TSS Intra-segment Interfaces cannot be used by PCS or PSSS UoCs.

TSS UoCs provide a *TS_TPM Interface* to enable the reuse of transport protocol plugin modules which support interoperability between different implementations of TSS UoCs.

4.7.11.2 Transport Protocol Module Interface Requirements

1. A TPM shall provide the TPM Interface as specified in Section E.4.2.

2. A TPM shall supply the return code value returned from TPM Interface operations as specified in Section E.4.2.

3. The Transport Protocol Module Capability shall use the IDL Programming Language defined in Section 4.14 for its TPM Interface.

4.7.12 Data Store Support Capability Requirements

Access to Data Stores is a necessary common technical function. This function is usually provided by the OS segment. However, it is no longer always the case that OS-level interfaces provide the access to Data Stores. For example, in some architectures, this functionality could be

provided by Data Base Server, Data Services, or File Systems. In fact, some architectures do not provide OS-level storage access. To maintain portability of PCS and PSSS UoCs across a wide variety of Data Store access, a common point of presence for Data Store access is required. The Data Store Access Capability provides this functionality within the TSS so that PCS and PSSS UoCs can remain agnostic to the source of this information.

1. A Data Store Support Capability shall be configured through information provided by the TSS Configuration Capability.
2. A Data Store Support Capability configuration data shall be specified in accordance with the Configuration Services requirements in Section 4.2.5.
3. A Data Store Support Capability shall provide access to data stores.
4. A Data Store Support Capability shall use the TS Capability to exchange information between Data Stores and PCS UoCs.
5. A Data Store Support Capability shall use the TS Capability to exchange information between Data Stores and PSSS UoCs.

4.7.13 Component State Persistence Capability Requirements

The CSP Interface provides a standardized interface for PCS, PSSS, and TSS UoCs to checkpoint their internal state, or store private data. The standardization of this interface allows software suppliers to create reusable software components/products, while protecting the intellectual property associated with the internal data structures of software components. It also facilitates the integration of these software components into disparate architectures and platforms. There are two types of data that may use this interface:

- Private data – data only accessed by a single UoC
- Checkpoint data – data used for backup of the state of a UoC to allow for redundancy and reversion

Note: Neither Checkpoint nor Private data need to be modeled in the FACE Data Architecture. All other types of data are modeled in the FACE Data Architecture and use the TS Interface for IO operations.

The data passing through the CSP interface is exempt from being modeled in the FACE Data Architecture. To limit the use of this interface to a single UoC, the interface and the data store contain a reference to the UoC that created the stored data. In order to ensure portability of UoCs across systems, the file locations and permissions are described in a configuration file passed into the CSP Capability at initialization.

1. A CSP Capability shall be configured through information provided by the TSS Configuration Capability.
2. The CSP Capability configuration data shall be specified in accordance with the Configuration Services requirements in Section 4.2.5.
3. A CSP Capability shall provide the CSP Interface as defined in Section 4.8.4.2.
4. A CSP Capability shall provide access to the Checkpoint data when the UUID passed in matches the UUID associated with the Checkpoint data requested.
5. A CSP Capability shall store Checkpoint data and its associated UUID.

6. A CSP Capability shall provide access to the Private data when the UUID passed in matches the UUID associated with the Private data requested.

7. A CSP Capability shall store Private data and its associated UUID.

8. A CSP Capability shall supply a library for PCS or PSSS UoCs to use.

4.7.14 Framework Support Capability Requirements

Framework Support Capability (FSC) is an abstraction that translates Component Framework interfaces to FACE defined interfaces. The FSC requirements are presented here and shown in Figure 16 and Figure 17. The PCS and PSSS requirements to use framework software components are presented in the respective sections.

Component Frameworks ease development of new software by allowing developers to focus on the unique domain requirements of their software components. The software that provides infrastructure to support the functional requirements ("common technical functions") is provided by the framework, and accessed by framework-specific APIs. Frameworks, however, are different from portable libraries. With libraries, software components invoke the functions and objects provided by the library, but with frameworks, developers implement functions and objects specific to their software component that are then instantiated and invoked by the framework. Inclusion of framework support standardizes deployment and control operations to the integrator, such as independent executables *versus* container managed executables or inversion of control of software components.

Many different commercial software components and industry product line frameworks exist today. These frameworks perform software infrastructure functions such as Life Cycle Management (LCM), logging, persistence, time processing, etc. However, each framework provides unique interfaces for each of these functions. These unique interfaces introduce a barrier to portability of the modules and software components between different frameworks and often between different implementations of the same frameworks. In order to achieve portability of UoCs across Component Frameworks, these unique interfaces need to be abstracted.

In a software environment targeting the FACE Reference Architecture, abstraction is accomplished through the use of the FACE Interfaces (TSS, IOS, Config, and HMFM). Each framework-unique interface can be proxied by the FSC and mapped to the FACE Interfaces provided by other TSS Capabilities. This makes the UoC agnostic to the specific framework. In order to reduce the effort of adding and maintaining the additional syntactic framework abstractions, and to promote interoperability, the FACE SDM is used to define and isolate the variation points between frameworks to a data message to/from the TS Interface. The FSC can be used across the OSS Profiles and across many commercial and military frameworks.

This abstraction strategy places a framework software component in the PCS or PSSS layer, and the container in the TSS layer. The use of FACE Interfaces to provide abstractions between the modules and the container isolate the integration changes needed to port the software components across component and product line frameworks.

Figure 16 depicts the FACE abstraction strategy for a PCS UoC. The PCS UoC is intended to be a component within a framework container.

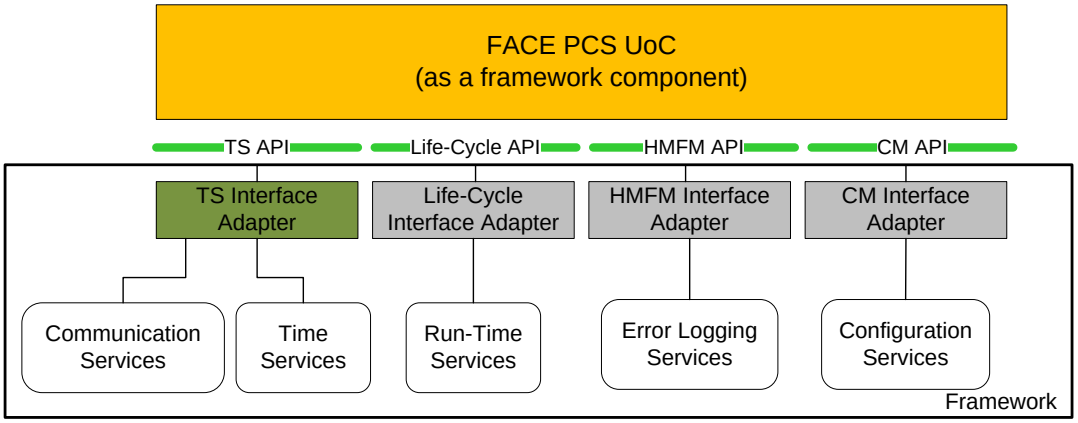

Figure 16: PCS UoC as a Framework Component

Figure 17 depicts the FACE abstraction strategy for a PSSS UoC. The PSSS UoC is intended to be a component within a framework container.

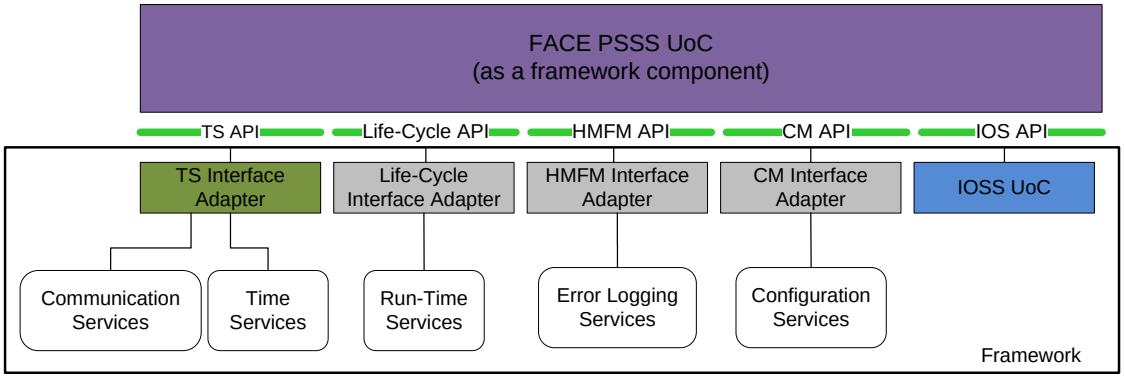

Figure 17: PSSS UoC as a Framework Component

4.7.14.1 Framework Support Capability Requirements

1. A Framework Abstraction Capability shall be configured through information provided by the TSS Configuration Capability.
2. The FSC configuration data shall be specified in accordance with the Configuration Services requirements in Section 4.2.5.
3. An FSC shall adapt the specific framework interfaces to the respective TSS Inter-segment Interfaces.
4. An FSC shall adapt the specific framework interfaces to the Initializable Capability of the LCM Services as defined by Section 4.13.2.
5. An FSC shall adapt the specific framework interfaces to the Configurable Capability of the LCM Services as defined by Section 4.13.3.
6. An FSC shall adapt the specific framework interfaces to the Connectable Capability of the LCM Services as defined by Section 4.13.4.
7. An FSC shall adapt the specific framework interfaces to the Stateful Capability of the LCM Services as defined by Section 4.13.5.

8. An FSC shall map the LCM technical function provided by a Framework Implementation to the FACE Data Architecture UoPModel Template that specifies the Life Cycle state data.

9. An FSC shall map the System Time technical function provided by a Framework Implementation to the FACE Data Architecture UoPModel Template that specifies the system time data.

10. An FSC shall map the system HMFM technical functions provided by a Framework Implementation to the FACE Data Architecture UoPModel Template that specifies the system HMFM data.

11. An FSC shall map the OS HMFM technical functions provided by a Framework Implementation to the FACE HMFM Interface.

12. An FSC shall map the Data Exchange technical functions provided by a Framework Implementation to the FACE Data Architecture UoPModel Template that specifies the Message parameter.

13. An FSC shall map other technical functions provided by a Framework Implementation to the FACE Data Architecture UoPModel Template that specifies the function data.

14. An FSC shall use the native interfaces of the framework implementation required to perform its technical functions.

4.7.14.2 Interfaces to Support Frameworks

The FSC may call the native interfaces of a Component Framework for core SW Infrastructure functions. The typical SW Infrastructure function interfaces provided by a Component Framework container may include (but are not limited to):

- Life Cycle Management
- Error Reporting
- Logging
- Persistent Storage to Data Stores
- Persistent Storage of PCS or PSSS UoC Private data or internal state
- Get/Set Time

To promote portability, the interfaces provided by a framework are accessed through interfaces defined by the FACE Technical Standard. The LCM functions are accessed through the LCM Services Interface provided by PCS, PSSS, or TSS UoCs. If Inversion of Control is desired, the PCS or PSSS UoC provides an execute operation in the UoC Interface. Other functions provided by the framework are accessed through TSS Inter-segment Interfaces. The Logging and System Error Reporting Framework functions are accessed through the TS Interface, run-time errors are accessed through the FACE HMFM Interface, the persistence of PCS/PSSS UoC internal state uses the CSP Interface, and the remaining framework functions are accessed through the TS Interface using data types specified by the FACE Data Architecture.

4.8 Transport Services Interfaces

4.8.1 Introduction

The Transport Services Interfaces provide a set of standardized interfaces for software component communication. UoCs in both the PCS and the PSSS use the interfaces provided by the TSS libraries for data exchange and access to other software infrastructure services defined by Section 4.7. The standardization of these interfaces allows software suppliers to create reusable software components/products, and facilitates the affordable integration of these software components into disparate architectures and platforms.

4.8.1.1 Inter-Segment Transport Services Interfaces

The TSS Inter-segment Interfaces are provided by TSS UoCs to be used by the PCS, and PSSS UoCs to access common technical functions provided by the SW Infrastructure such as data transport, access to reference data, or CSP information. As defined in Section 4.7.1.1, a TSS may be composed of UoCs which implement the inter-segment interfaces.

The Inter-segment Interfaces that TSS libraries provide include:

- Type-Specific Interface (TS Interface)
- CSP Interface

Note: Data Store uses the TS Interface to access Data Stores, so it is not a unique interface provided to users of TS components.

To ensure portability of PCS and PSSS UoCs, and to enable conformance verification, the TS Interface is strongly typed per the PCS or PSSS UoP Supplied Model (USM). The TS Interface uses declared and structured data. The Transport Services Capability implemented as a TSS library provides the TS Interface. The Transport Services Capability manages the TS Interface, and the Distribution Capability provides for the distribution of information according to configuration data from the TSS Configuration Capability.

The Inter-segment Transport Services Interfaces support the communication and message services required to meet the system performance (e.g., throughput, latency, delivery guarantees) for the associated data exchanges. These interfaces support data transport industry standards. Examples of standards that may be used to implement the TS Interface include, but are not limited to, POSIX, ARINC 653, CORBA, and DDS.

4.8.2 TS Interface Description

- The TS Interface supports the following communication styles:
  - — Buffering/queuing
  - — Single Instance Messaging (e.g., sampling port, shared memory)
- The TS Interface supports the following synchronization styles:
  - — Blocking
  - — Non-blocking

- The TS Interface supports the following message structures:
  - — Fixed Length
  - — Variable Length
- The TS Interface supports the following message timing types:
  - — Periodic
  - — Sporadic
  - — Aperiodic
- The TS Interface supports the following message distribution types:
  - — Broadcast
  - — Multicast
  - — Unicast
  - — Anycast
- The TS Interface is strongly typed

The TS Interface services and data types are defined in Appendix E.

4.8.3 The Component State Persistence Interface Description

- The CSP Interface supports storage mechanisms, such as, but not limited to:
  - — File
  - — Database
  - — Object Data Store
  - — Network Attached Storage
- The CSP Interface supports storage media, such as, but not limited to:
  - — Spinning Media
  - — Non-volatile Memory
  - — Local Disk Storage
  - — Removable Storage

The CSP Interface Specification is defined in Section E.3.5.

4.8.4 Transport Services Segment Inter-Segment Interface Requirements

4.8.4.1 TS Interface

1. The TS Interface shall meet the Base interface specification of Section E.3.1.

   Note: When using a Type Abstraction component, only one implementation of the Base interface specified by Section E.3.1 is required.

2. The TS Interface shall meet the Type-Specific Typed interface specification of Section E.3.2.

3. The TS Typed module as defined in Section E.3.2 shall be parameterized with the DATATYPE_TYPE as described in a USM and in accordance with the FACE language bindings, defined in Section 4.14.

4. The fully qualified name of the created TS Typed module shall be FACE::TSS::<UOP_MODEL_NAME>::<DATATYPE_TYPE>::TypedTS.

   Note: DATATYPE_TYPE is the short name of a Template or CompositeTemplate, not a fully qualified name. UOP_MODEL_NAME is the name of the root UoPModel in which DATATYPE_TYPE is a member. The resulting declarations follow programming language mapping rules as if the interface was declared in a directory tree and IDL file as FACE/TSS/<UOP_MODEL_NAME>/<DATATYPE_TYPE>/TypedTS.idl.

5. The *FACE::TS::Base::Initialize*(TS) function shall be non-blocking regardless of the underlying transport mechanism.

   Note: *Initialize*(TS) may need to complete asynchronously after *Initialize*(TS) returns if the time to complete exceeds some threshold.

6. The *FACE::TS::Base::Create_Connection*(TS) function shall be non-blocking regardless of the underlying transport mechanism.

   Note: *Create_Connection*(TS) may need to complete asynchronously after *Create_Connection*(TS) returns if the time to complete exceeds some threshold.

7. A TS Interface shall supply the return code value returned from the *Type-Specific Base Interface functions* as specified in Section E.3.1.

8. A TS Interface shall supply the return code value returned from the *Type-Specific Typed Interface functions* as specified in Section E.3.2.

9. The TS Type-Specific Extended Typed module as defined in Section E.3.4 shall be parameterized with the DATATYPE_TYPE and RESPONSE_DATATYPE as described by the USM and in accordance with the FACE language bindings, defined in Section 4.14.

10. The fully qualified name of the created TS Type-Specific Extended Typed module shall be FACE::TSS::<UOP_MODEL_NAME>::<DATATYPE_TYPE>_<RESPONSE_DATATYPE>::TypedTS.

    Note: Most of the time, DATATYPE_TYPE and RESPONSE_DATATYPE are in the same model. DATATYPE_TYPE and RESPONSE_DATATYPE are the short names of a Template or CompositeTemplate, not fully qualified names. UOP_MODEL_NAME is the name of the root UoPModel in which DATATYPE_TYPE or RESPONSE_DATATYPE is a member. The resulting declarations follow programming language mapping rules as if the interface was declared in a directory tree and IDL file as FACE/TSS/<UOP_MODEL_NAME>/<DATATYPE_TYPE>_<RESPONSE_DATATYPE>/TypedTS.idl.

4.8.4.2 CSP Interface

1. The CSP Interface shall meet the specification in Section E.3.5.

2. The CSP Interface shall supply the return code value returned from CSP Interface functions as specified in Section E.3.5.

4.8.5 Transport Services Segment Inter-Segment Message Parameter Data Requirements

Figure 18 depicts the parameters of TSS Inter-segment Message. Many of the parameters within the TSS interface, including instances of the message parameter and *qos_parameter* parameter, are defined using the FACE Data Architecture. Others are defined in IDL in Appendix E. The TSS message parameter is modeled as a UoPModel Template. The TSS Header is specified in IDL in Appendix E. The *qos_parameter* is either Configuration Information or UoPModel Template(s). The Header and *qos_parameter* can provide metadata about the transported message to the receiving PCS or PSSS UoC.

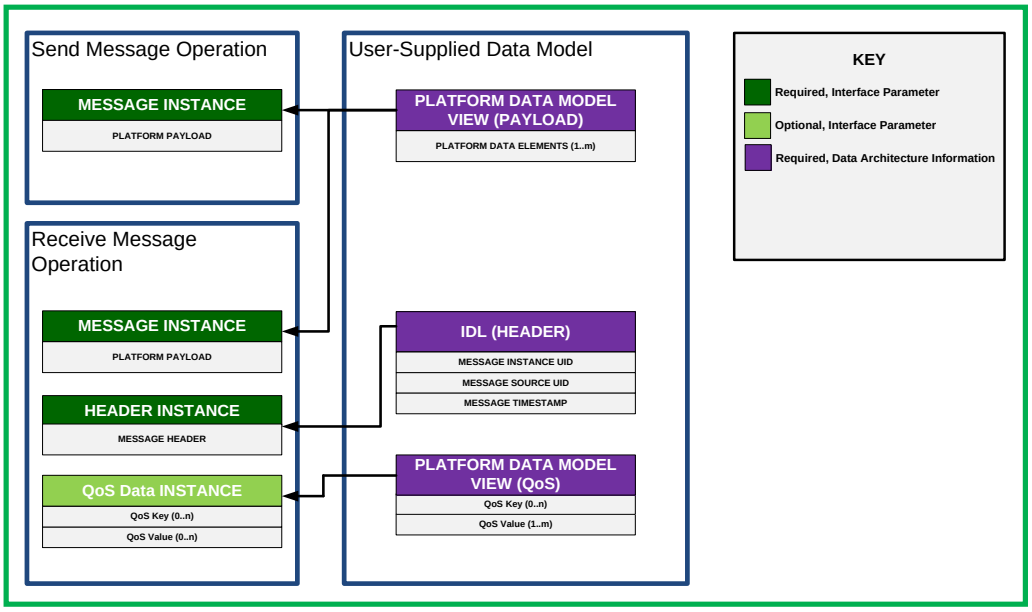

Figure 18: TSS Inter-Segment Interface Data Parameters

4.8.5.1 TS Interface Data Parameters

An instance of the Message in Figure 18 is the basic element of data exchange within the TSS. The Message Instance is a concrete instantiation of the message parameter type for the TSS. The Message Instance is the data known throughout the TSS between UoCs at run-time. Transport of Message Instances between TSS endpoints may be optimized by the transport method. The packing and encoding of an instance of the Message are not prescribed by the FACE Technical Standard. The Message parameter does not include the TSS Data Elements associated with routing, QoS, definition, data, hierarchy, and conversion.

1. The TS Interface shall provide the following data elements:
    a. Header parameter
    b. Message parameter
2. When QoS Management is performed, the TS Interface shall provide content for the QoS parameter.

4.8.5.2 Message Parameter

1. Instances of the Message parameter shall be a language-specific data structure of the UoPModel Template as specified by the FACE Data Architecture.

4.8.5.3 Header Parameter

1. Instances of the Header parameter shall consist of the following data elements in the order listed below:

    a. Message Instance UID

    b. Message Source UID

    c. Message Timestamp

4.8.5.4 QoS Parameter

1. Instances of the QoS parameter shall define the QoS values which have been applied to the transport of data within the system by the TSS UoCs.

2. When QoS Management is performed, instances of the QoS parameter shall consist of repeating sets of the following data elements in the order listed below:

    a. QoS Key

    b. QoS Attribute Values

    Note: A QoS Key may have multiple possible values, but an instance must have one value for one key.

4.8.6 Transport Services Segment FACE Data Architecture Requirements

1. The TSS Header Instance shall use the IDL to Programming Language Mappings defined in Section 4.14 per the TSS Header IDL in Section E.2.1.

    Note: For the capabilities it supports, the TSS uses the Integration Model to build configuration data for QoS, data routing, message associations, and/or transformations, as described in Section 4.9.1.5.

2. Instances of Time parameters passed by a TSS UoC through the TS Interface shall be implemented through a USM UoPModel Template in accordance with requirements in Section 4.9.4.

3. Instances of LCM parameters shall be implemented through a UoPModel Template in accordance with requirements in Section 4.9.4.

4. Instances of Data Stores parameters passed by a TSS UoC through the TS Interface shall be implemented through a UoPModel Template in accordance with requirements in Section 4.9.4.

5. Instances of framework storage message parameters passed by a TSS UoC through the TS Interface shall be implemented through a UoPModel Template in accordance with requirements in Section 4.9.4.

6. The Transport Service Capability shall use the IDL to Programming Language Mappings defined in Section 4.14 for its TS Interface.

7. A TSS UoC providing the TS Interface shall provide data to PCS UoCs in conformance with the FACE Data Model Language bindings defined in Section 4.9.2.

8. A TSS UoC providing the TS Interface shall provide data to PSSS UoCs in conformance with the FACE Data Model Language bindings defined in Section 4.9.2.

9. A TSS UoC providing the TS Interface shall accept data from PCS UoCs in conformance with the FACE Data Model Language bindings defined in Section 4.9.2.

10. A TSS UoC providing the TS Interface shall accept data from PSSS UoCs in conformance with the FACE Data Model Language bindings defined in Section 4.9.2.

4.9 Data Architecture

The FACE Data Architecture consists of the FACE Data Model Language, a common Shared Data Model (SDM) that establishes foundational elements for modeling, and a set of rules for its application. The FACE Data Architecture is applied to create UoP Supplied Models (USM) and Domain-Specific Data Models (DSDM) to provide for interoperability.

The FACE Data Model Language leverages the Open Universal Domain Description Language (Open UDDL), the Meta Object Facility (MOF) metamodel language, the Template Language, and constraints defined in the OMG Object Constraint Language (OCL) that further defines the structure and rules for construction of model elements, rules for the interaction of model elements, and rules for software code generation.

The FACE Data Model Language consists of four primary model groupings:

1. Data Model (defined in the Open UDDL Technical Standard)

2. UoP Model

3. Integration Model

4. Traceability Model

The Shared Data Model (SDM) establishes a foundation of core data elements used as building blocks to create all other data models. The SDM is Configuration Control Board (CCB) managed per the FACE SDM Governance Plan providing confidence in the core modeling elements and utility of reuse and potential of reduced data conversion needs.

Software code generation is defined through a set of Template Language rules and IDL Type bindings that map FACE Data Model Language elements to IDL and from IDL to each of the supported programming languages.

4.9.1 FACE Data Model Language Overview

The FACE Data Model Language is specified by a metamodel providing a formal and unambiguous description of UoP data exchanges (interfaces between and integration of). The Modeling Language leverages the Data Model Language provided by the Open UDDL Technical Standard with FACE specific extensions to support describing the data exchanged by UoCs. Additional OCL constraints supply semantic rules to which data model content must adhere. The Template Language specifies the presentation of data.

Figure 19 extends Figure 1 from the Open UDDL Technical Standard and illustrates several aspects of the Modeling Language. Groupings of Modeling Language elements, roughly aligned to the model groupings introduced in Section 4.9, are shown as boxes with rounded corners; relationships between elements as short, stubbed arrows. Grouping of Modeling Language elements is shown vertically, from top to bottom, showing definition of data model, interface model, and integration model elements respectively. Horizontally, from left to right, levels (or perspectives) show refinement of model elements from a more abstract level, to a more concrete one.

Traceability Model Elements are not shown in order to avoid diagram clutter. While not part of the Modeling Language, artifact generation and code and configuration are shown for context.

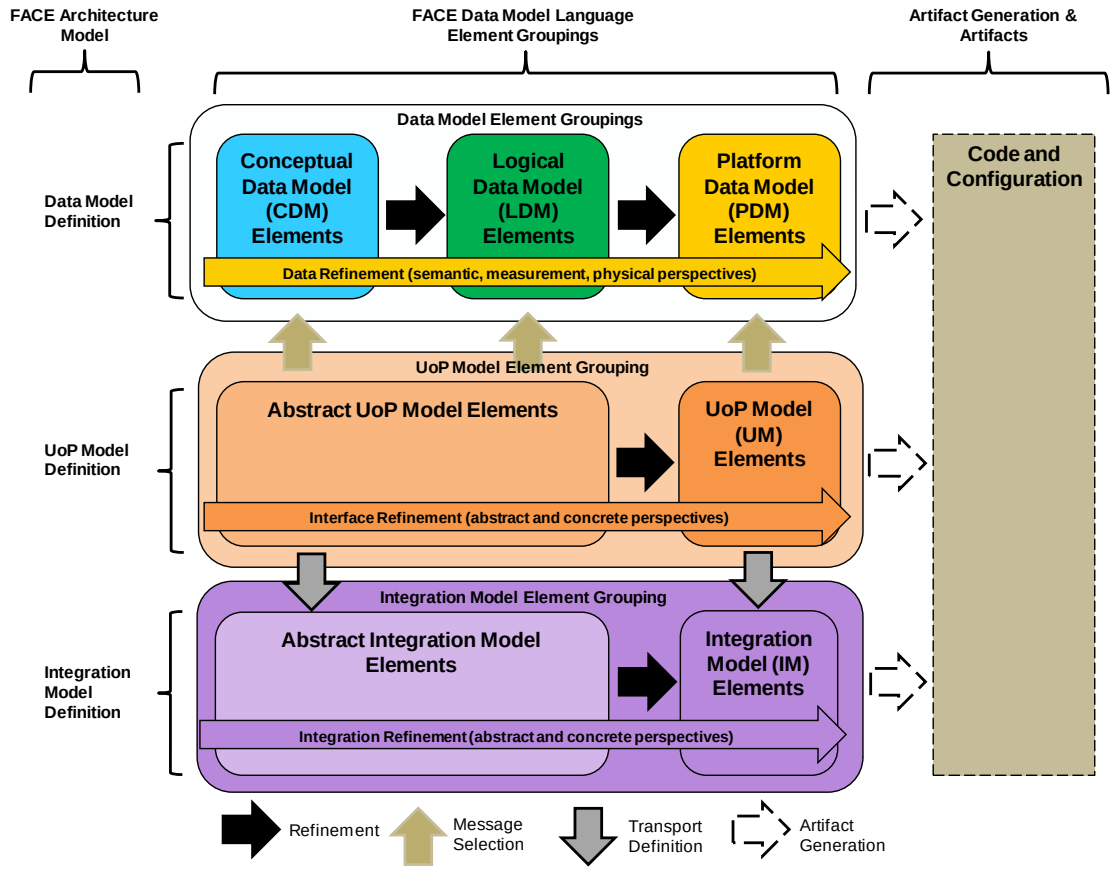

Figure 19: Data Model Language

Figure 19 shows the various constituent models that make up the FACE Data Architecture and the relationships between the models. The progression from left to right is from abstract to concrete. With each level of refinement, the degree of specificity increases, moving the model closer to a complete definition of the software application solution required for generation of code.

The first of the Architectural model elements, the *datamodel*, is used to define data elements with unambiguous specificity for use in the FACE Technical Standard. The *datamodel* is divided into three different data elements with unambiguous specificity for use by FACE Units of Portability (UoP) or Domain-Specific Data Models (DSDM) as defined by the FACE Technical Standard.

- The *Conceptual Data Model* provides the semantic definition of the entities and their relationships characterized by Observables
- The *Logical Data Model* adds measurement information to each characterization by defining value type, units, measurement, and frame of reference (through the measurement system)
- The *Platform Data Model* adds physical data type information to the logical measurement characterization
- The *UoPModel* provides the definition of a software component and its defined interfaces
- The *IntegrationModel* provides a mechanism to describe the Transport Services Segment (TSS) integration details between two or more UoPs
- Lastly, the *TraceabilityModel*, not shown in the figure, provides a mechanism to trace model elements to an external model

The following sections give an overview of each Data Model Language element grouping.

4.9.1.1 Conceptual Data Model Overview

The Conceptual Data Model (CDM) is defined by the Open UDDL Technical Standard. A CDM is comprised of entities, characteristics, and associations that provide definitions of concepts, their characteristics, and the context relating them. Observables that are fundamental to the domain and have no further decomposition are used to specify these defining features. Domain concepts can be captured in the CDM through the definition of basis entities. A basis entity represents a unique domain concept and establishes a foundation from which conceptual entities can be specialized. Basis entities are axiomatic. This allows for separation of concerns, allowing multiple domains to be modeled.

4.9.1.2 Logical Data Model Overview

The Logical Data Model (LDM) is defined by Open UDDL Technical Standard. An LDM consists of entities, characteristics, and associations that realize their definition from the CDM. An LDM provides terms of measurement systems, coordinate systems, reference points, value domains, and units. The principal level of detail added in an LDM is provided through frames of reference for representing characteristic values. Multiple LDM elements may realize a single CDM element.

4.9.1.3 Platform Data Model Overview

The Platform Data Model (PDM) is defined by the Open UDDL Technical Standard. A PDM consists of entities, characteristics, and associations that realize their counterpart definition from an LDM. In a PDM, specific representation details such as data type and precision are provided to represent characteristics. Multiple PDM elements may realize a single LDM element. Additionally, the PDM specifies how data is presented across the TS Interface using views.

4.9.1.4 Unit of Portability Model Overview

A UoP Data Model consists of elements that provide the means to formally specify the interfaces of a UoP. The interfaces are specified using reference to PDM elements to allow "message typing" of the interface. Abstract UoP elements support a platform-independent specification of the UoP and its interfaces through references to LDM and CDM elements. Connection model

elements are representations of “logical” connections and do not necessarily correspond to the actual communication channels for exchanging data. The UoP Data Model is specific to the FACE Technical Standard.

4.9.1.5 Integration Model Overview

An Integration Model consists of elements that provide the means to model the exchange of information between UoPs. An Integration Model captures data exchanges, view transformations, and integration of UoPs for documentation of integration efforts. An Integration Model relies on UoP Models for expressing interconnectivity. The focus is on documenting UoP data exchange details. The Integration Model is specific to the FACE Technical Standard.

4.9.2 FACE Data Model Language IDL Bindings

Figure 20 illustrates the language bindings from the PDM to each of the supported programming languages. The language bindings define the mapping from a specified Template to data structures in code. These data structures are sent or received by a software component through the TSS *send_message*(TS) and *receive_message*(TS) methods. The language bindings preserve the byte size and value range when translating to various processor architectures and programming languages. The language bindings ensure software components are portable at the API level. It is the responsibility of the TSS to manage serialization and deserialization of the data structure and to transmit the data.

Section J.8 details the FACE Data Model Language IDL bindings.

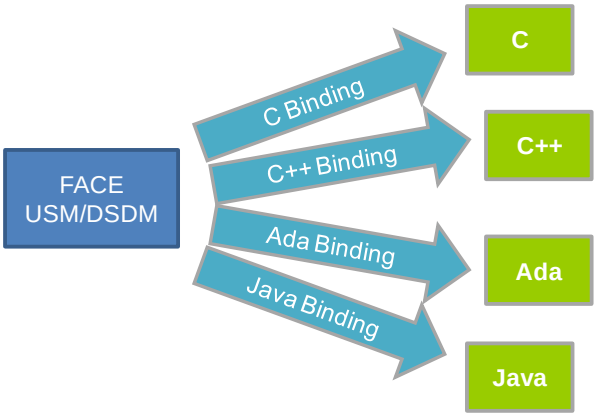

Figure 20: FACE Data Model Language Bindings

4.9.2.1 Specification

The first part of the language bindings specifies a mapping from a USM or DSDM to IDL. The second portion of the language bindings defines the map from IDL to each of the supported programming languages as specified in Section 4.14. These two parts form the FACE Data Model Language IDL bindings.

The language bindings define the rules for mapping from PDM elements to programming language definition of the data types. A specific tool is not required to be used in implementing these rules.

Note: A software supplier may directly generate or manually create the data structures from a PDM without leveraging IDL as an intermediate format.

4.9.2.1.1 IDL to Programming Language Mappings

The FACE Data Architecture follows the IDL to Programming Language Mappings detailed in Section 4.14 and the FACE Data Model Language IDL bindings detailed in Section J.8.

4.9.3 Definitions

4.9.3.1 FACE Shared Data Model

The FACE Shared Data Model (SDM) is the primary point of interaction between software suppliers and system integrators as it is the common foundation for all USMs. The FACE SDM provides the core extensible elements from which USMs are built. The FACE SDM Governance Plan defines the policies for management of the FACE SDM. The FACE SDM CCB manages the FACE SDM.

4.9.3.2 UoP Supplied Model

Software suppliers develop USMs from the SDM using the Data Model Language to represent information about each UoP, including the data each UoP sends and receives. The Data Model Language provides and promotes common understanding and meaning of data exchanged.

4.9.3.3 Domain-Specific Data Model

A Domain-Specific Data Model (DSDM) is a data model designed to the FACE Data Architecture Requirements. It captures domain-specific semantics and generally does not contain UoP Models.

4.9.4 Data Architecture Requirements

The FACE Data Model Language is formally specified by the Open UDDL Technical Standard, the FACE metamodel, OCL constraints, and template grammar and constraints. The metamodel, grammar, and associated constraints are specified in Appendix J.

Single Observable Modeling is when the SDM, a DSDM, or a USM is developed to follow the Single Observable Constraint in Section J.7.1. The Single Observable Constraint limits Conceptual Entities to composing at most one element of a single Observable type. Models that follow the Single Observable Constraint provide a clearer understanding of Entities by reducing the likelihood of semantic information being embedded in the multiple composition of Observables. This is considered a data modeling best practice.

Entity Uniqueness Modeling is when each conceptual Entity in a DSDM or USM is unique. An Entity's Uniqueness is defined as each Entity must have a different Identity from all other Entities in the model. An Entity's Identity is defined by the complete set of the Entity's characteristics. This is considered data modeling best practice and is encouraged as it helps in Entity clarity and integration between different USM and DSDM models.

4.9.4.1 USM Requirements

A valid USM conforms to the metamodel, OCL constraints, template grammar, and template constraints.

1. Each UoC using the TS Interface shall be accompanied by a USM.

2. The USM shall be an XMI file conforming to the EMOF 2.0 metamodel specified in Appendix J.

3. The USM shall conform to the Open UDDL requirements.

4. The USM shall use the "xmi:id" attribute with a unique UUID as the ID for all elements.

5. The USM shall adhere to the OCL constraints in Section J.6.

6. When developed to Single Observable Modeling, the USM shall adhere to the conditional OCL constraint in Section J.7.1.

   Note: This is considered data modeling best practice and is encouraged but may not be required of all USMs.

7. When developed to Entity Uniqueness Modeling, the USM shall adhere to the conditional OCL constraint in Section J.7.2.

   Note: This is considered data modeling best practice and is encouraged but may not be required of all USMs.

8. The USM shall include all data elements sent by the UoC across the TS Interface.

9. The USM shall include all data elements received by the UoC across the TS Interface.

10. The USM shall include all data elements used by the UoC in a Security Transform.

11. The USM shall include all data elements sent by the UoC across the LCM Stateful Interface.

12. The USM shall include all data elements received by the UoC across the LCM Stateful Interface.

13. The USM shall adhere to the FACE Shared Data Model Governance Plan.

    Note: The FACE SDM Governance Plan describes how to extend the SDM.

14. The USM shall follow the FACE Data Model Language to IDL bindings specified in Section J.8 and the IDL bindings specified in Section 4.14.

4.9.4.2 Domain-Specific Data Model Requirements

A valid DSDM conforms to both the metamodel and associated constraints.

1. A DSDM shall be an XMI file conforming to the EMOF 2.0 metamodel specified in Appendix J.

2. The DSDM shall conform to the Open UDDL requirements.

3. The DSDM shall use the "xmi:id" attribute with a unique UUID as the ID for all elements.

4. The DSDM shall adhere to the OCL constraints in Appendix J.

5. When developed to Single Observable Modeling, the DSDM shall adhere to the conditional OCL constraint in Section J.7.1.

   Note: This is considered data modeling best practice and is encouraged but may not be required of all DSDMs.

6. When developed to Entity Uniqueness Modeling, the DSDM shall adhere to the conditional OCL constraint in Section J.7.2.

   Note: This is considered data modeling best practice and is encouraged but may not be required of all DSDMs.

7. The DSDM shall adhere to the FACE Shared Data Model Governance Plan.

   Note: The FACE SDM Governance Plan describes how to extend the SDM.

4.10 Portable Components Segment

Software components are designated as portable when they can be redeployed on different software environments without requiring more than a recompilation of the software component and a re-linking of software libraries, Programming Language Run-Times, and/or Component Frameworks.

The Portable Components Segment (PCS) logically contains a wide range of software components. Software components are considered to be PCS UoCs when they share the following properties:

- The software component provides software capabilities or services
- The software component is capable of executing in varying instantiations of FACE infrastructures
- The software component exclusively uses the TS Interface for data exchanges
- The software component exclusively uses the OSS Interface for OS support
- The software component adheres to the requirements of the PCS

4.10.1 Portable Components Segment Requirements

1. A PCS UoC shall only use the interfaces defined in Section 4.2, Section 4.8, Section 4.12, and Section 4.13.

   Note: If a Programming Language Run-Time is implemented in the PCS as part of the UoC, then the Programming Language Run-Time must conform exclusively to a subset of the TS Interface for all data exchange crossing the UoC boundary per the segment requirements.

2. When a PCS UoC uses multiple POSIX processes, the PCS UoC shall use the POSIX multi-process APIs as defined in Section 4.2.1.

3. When using Programming Language Run-Times, a PCS UoC shall do so in accordance with Section 4.2.3.

4. 4. When a PCS UoC uses a Component Framework, the PCS UoC shall use the Component Framework in accordance with Section 4.2.4.

5. When a PCS UoC uses the OSS Interface, the PCS UoC shall adhere to the restrictions specified in Section A.6 and Section A.7.

6. A PCS UoC shall communicate with other software components, through the TS Interface as defined in Section 4.8.

   Note: This includes Inter-partition/process and Intra-partition/process UoC to UoC communication.

7. All data communicated over the TS Interface shall be defined by the FACE Data Architecture in accordance with requirements in Section 4.9.

8. A Connection element "name" property shall be a case-insensitive string.

9. The Connection name provided to TSS UoCs to create a connection shall match the "name" property of the corresponding Connection element in the UoC's USM.

   Note: The USM may contain a default name which could be overridden by configuration data.

10. When a PCS UoC provides a graphical user interface, the UoC shall use graphics services per the requirements in Section 4.12.8.

11. When a PCS UoC retrieves Configuration Information, the UoC shall use the Configuration API per the requirements in Section 4.2.5.

12. When using Centralized Configuration, a PCS UoC shall use the TSS API.

    Note: Section 4.6.3.1.4 describes the Centralized Configuration Service.

13. When implementing a Component Framework, a PCS UoC shall do so according to the requirements in Section 4.10.1.3.

    Note: If a Component Framework is implemented in the PCS as a part of the UoC, then the Component Framework must conform exclusively to a subset of the TS Interface for all data exchanges crossing the UoC boundary per the segment requirements.

14. When a PCS UoC uses Data Stores, the PCS UoC shall use the TS Interface to store and retrieve data as described in Section 4.8.4.1.

15. When using CSP, a PCS UoC shall use the CSP Interface to store and retrieve its Checkpoint Data as described in Section 4.8.4.2.

16. When a PCS UoC provides the Injectable Interface, the PCS UoC shall provide the Injectable Interface as described in Section 4.11.4.1.

4.10.1.1 Portable Components Segment Operational Environment Requirements

1. When using OSS Health Monitoring, a PCS UoC defined to operate in a POSIX operational environment shall use the FACE Health Monitoring APIs described in Section 4.2.2.

2. A PCS UoC shall conform to the requirements in Section 4.2.4 when using OSGi.

4.10.1.2 PCS UoC Life Cycle Management Services Requirements

1. When providing a LCM Services Interface, a PCS UoC shall do so in accordance with the requirements of Section 4.13.

2. When using a LCM Services Interface, a PCS UoC shall do so in accordance with the requirements of Section 4.13.

4.10.1.3 Component Frameworks Provided as Part of PCS UoC Requirements

FACE requirements allow the use of Component Frameworks as integral parts of PCS UoCs as long as the libraries are FACE aligned and the entire Component Framework is provided as part of an aligned PCS UoC. There are no specific requirements to use Component Frameworks as integral parts of PCS UoCs.

1. When exchanging data using a framework, a PCS UoC shall use the TS Interface.
2. When accessing framework configuration interfaces, a PCS UoC shall use the Configuration Interface.
3. When storing Private and Checkpoint data, a PCS UoC shall use the CSP Interface.
4. When accessing framework capabilities not listed in requirements 1-3 (i.e., persistent storage, time interfaces, logging), a PCS UoC shall use the TS Interface.

   Note: A PCS UoC must use the TS Interface (*Send_Message*(TS)) to access framework-persistent storage create and update interfaces.

   Note: A PCS UoC must use the TS Interface (*Send_Message*(TS)) to access framework-persistent storage request interfaces.

   Note: A PCS UoC must use the TS Interface (*Receive_Message*(TS)) to access framework-persistent storage response interfaces.

   Note: A PCS UoC must use the TS Interface (*Receive_Message*(TS) to access framework time get time interfaces.

   Note: A PCS UoC must use the TS Interface (*Send_Message*(TS)) to access framework time set time interfaces.

   Note: A PCS UoC must use the TS Interface to access framework error and logging interfaces.
5. When a Component Framework is implemented as part of a PCS UoC, the Component Framework shall use the Initializable Capability of the LCM Services to initialize an instance of a PCS UoC.
6. When a Component Framework is implemented as part of a PCS UoC, the Component Framework shall use the Initializable Capability of the LCM Services to finalize an instance of a PCS UoC.
7. When a Component Framework is implemented as part of a PCS UoC, the Component Framework shall use the Configurable Capability of the LCM Services to configure an instance of a PCS UoC.
8. When a Component Framework is implemented as part of a PCS UoC, the Component Framework shall use the Connectable Capability of the LCM Services to connect an instance of a PCS UoC.
9. When a Component Framework is implemented as part of a PCS UoC, the Component Framework shall use the Connectable Capability of the LCM Services to disconnect an instance of a PCS UoC.

10. When a Component Framework is implemented as part of a PCS UoC, the Component Framework shall use the Stateful Capability of the LCM Services to query the state of an instance of a PCS UoC.

11. When a Component Framework is implemented as part of a PCS UoC, the Component Framework shall use the Stateful Capability of the LCM Services to change the state of an instance of a PCS UoC.

4.10.1.4 Security Transformation Requirements

Security Transformations perform transformations of data for security purposes as described in Section 5.2.2. The FACE Technical Standard does not specify or constrain where transformations are performed.

1. When a Security Transformation is implemented as part of a PCS UoC, all data crossing the Security Transformation boundary shall be defined in accordance with the FACE Data Architecture in Section 4.9.

   Note: Recommend the security transform use a TS Interface when traversing the transform boundary internal to the PCS.

   Note: Given the sensitivity of the internal interface data model, there may be restrictions on availability and distribution of the detailed data models levied by the platform and/or security relevant transform supplier.

2. When a Security Transformation is implemented as part of a PCS UoC, the characterization of the transformation shall include a detailed description of the transformation.

   Note: The detailed description of the Security Transformation should be sufficient to enable interoperability with similar transformations.

   Note: Given the sensitivity of the transformation characterization data, there may be restrictions on availability and distribution of the detailed data models levied by the platform and/or security-relevant transform supplier.

4.11 Unit of Conformance

A Unit of Conformance (UoC) is developed to meet the conformance requirements of the OSS Profile(s) and a single segment. System software solutions can be made up of multiple UoCs. UoCs can be integrated to support these software solutions. Integration of UoCs requires UoC to UoC communication.

4.11.1 Unit of Conformance Instantiation

One or more UoCs can be deployed within the same address space. When more than one UoC is deployed in a single address space, it is important for the UoCs to be coordinated so that they do not conflict with one another. One method of coordinating is to utilize an OSS profile that optionally includes multiple POSIX process support such as General Purpose or Safety Extended. Another method to coordinate the UoCs is through the use of a container design pattern where the container software includes an entry point such as “main”, and the container instantiates the objects or libraries within all UoCs within the address space.

Once the UoCs are instantiated, the communications between the UoCs can be established using the Injectable Interface.

4.11.2 Unit of Conformance Communications

UoC-to-UoC communication is required to use Transport Services Interfaces as defined in the FACE Technical Standard. Communication between software components, capabilities, and services within a UoC are not required to use defined Transport Services Interfaces. Use of defined Transport Services Interfaces for communication between software components, capabilities, and services within a UoC are permitted but not required. Figure 21 depicts an example of PCS inter-UoC and intra-UoC communications. See Section A.1 and Section A.6 for definitions of APIs and capabilities whose inter-UoC communication usages are restricted.

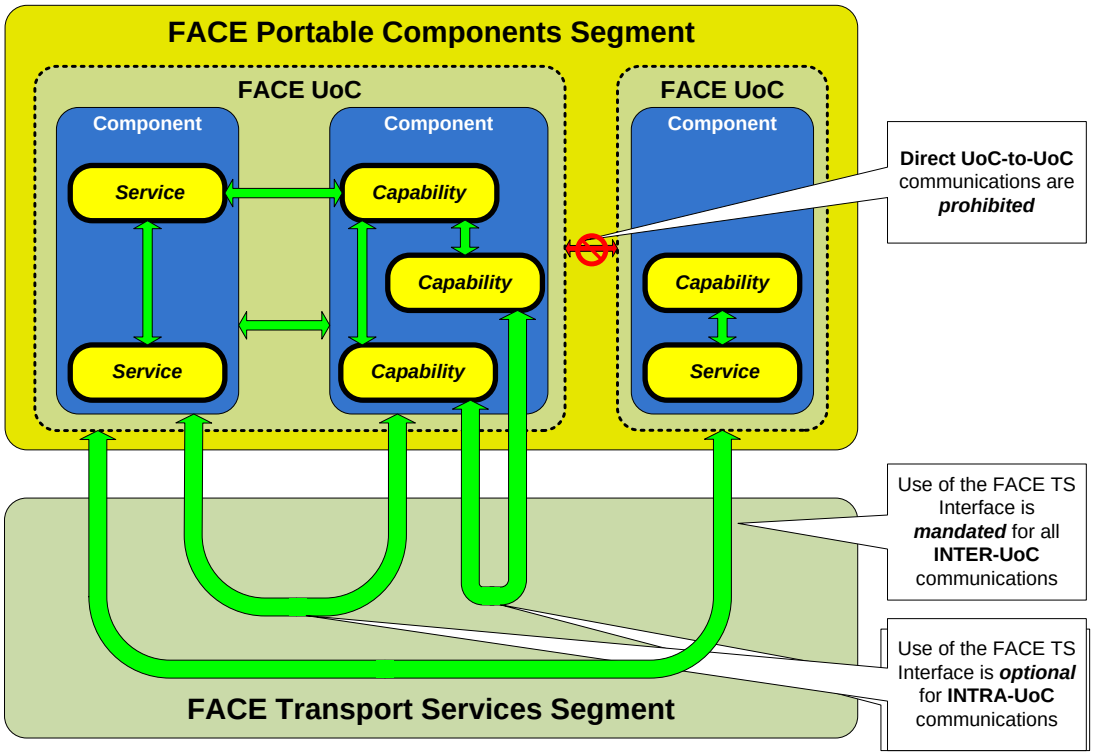

Figure 21: Example PCS Inter-UoC and Intra-UoC Communications

4.11.3 Injectable Interface

FACE Interfaces create an inherent using/providing dependency between UoCs. In order for a UoC to use an interface, it must be integrated in the same address space with at least one UoC that provides that interface. One of two strategies is employed to resolve this dependency for any given FACE Interface. When the FACE Interface is declared by a programming language binding directly, such as the C function prototypes for HMFM in Appendix F, the dependency is resolved by the linker. When the FACE Interface is declared by IDL, the dependency is resolved by integration software using the Injectable Interface. The Injectable Interface implements the dependency injection idiom of software development.

FACE Interfaces declared by IDL have the feature of supporting more than one interface provider in the same address space. This allows UoCs to be deployed to the same partition using different interface providers as needed to satisfy system requirements. In order to leverage this

feature, integration software is responsible for associating the desired instance of the interface provider to the interface user during initialization of the partition, after all the UoC instances have been created. The Injectable Interface provides the mechanism for that association.

Table 8 lists, for each FACE segment UoC, the potentially used FACE Interfaces declared by IDL. When a segment UoC uses one of these interfaces, it must provide the corresponding Injectable Interface. The user of that Injectable Interface is the integration software. Note that while the Injectable Interface itself is declared by IDL, it is not used by a segment UoC and is thus not listed.

Table 8: FACE Interfaces Requiring UoC to Provide Injectable Interface

Segment UoC	**Potentially Used FACE Interfaces Defined by IDL**	
PCS UoC	Transport Services	Type-Specific Base (Section E.3.1) Type-Specific Typed (Section E.3.2)[1] Type-Specific Extended (Section E.3.4)[1] Component State Persistence (Section E.3.5)
	Life Cycle Management Services	Initializable (Section D.2) Configurable (Section D.3) Connectable (Section D.4) Stateful (Section D.5)[2]
	Configuration Services	Configuration (Section G.2)
TSS UoC	Transport Services	Component State Persistence (Section E.3.5) Type Abstraction (Section E.4.1) Transport Protocol Module (Section E.4.2) Serialization (Section E.3.3)
	Life Cycle Management Services	Initializable (Section D.2) Configurable (Section D.3) Connectable (Section D.4) Stateful (Section D.5)[2]
	Configuration Services	Configuration (Section G.2)
PSSS UoC	Transport Services	Type-Specific Base (Section E.3.1) Type-Specific Typed (Section E.3.2)[1] Type-Specific Extended (Section E.3.4)[1] Component State Persistence (Section E.3.5)
	Life Cycle Management Services	Initializable (Section D.2) Configurable (Section D.3) Connectable (Section D.4) Stateful (Section D.5)[2]

Segment UoC	Potentially Used FACE Interfaces Defined by IDL	
	I/O Services	Supported I/O Bus Architectures (Section C.3) Extending I/O Bus Architectures (Section C.4)
	Configuration Services	Configuration (Section G.2)
IOSS UoC	Life Cycle Management Services	Initializable (Section D.2) Configurable (Section D.3) Connectable (Section D.4) Stateful (Section D.5)[2]
	Configuration Services	Configuration (Section G.2)

[1] This interface is instantiated for each specific message type, and each instantiation requires a separate instantiation of Injectable.

[2] This interface is instantiated for each state representation, and each instantiation requires a separate instantiation of Injectable.

4.11.4 Unit of Conformance Requirements

1. A UoC shall exist entirely in a single segment.

4.11.4.1 Injectable Interface Requirements

1. A UoC shall provide an instance of the Injectable Interface for each FACE Interface declared by IDL that it uses. (See Table 8 for the list of IDL Defined interfaces.)
2. Instances of the Injectable Interface provided by a UoC shall be in accordance with Appendix I.
3. A UoC shall document the FACE Interfaces declared by IDL that it provides.
4. The fully qualified name of the instantiated Injectable Interface module shall be <SCOPED_FACE_INTERFACE>_Injectable.

   Note: SCOPED_FACE_INTERFACE is the fully qualified name of the FACE Interface being bound by the instantiation. The resulting declarations follow programming language mapping rules as if the interface was declared in a directory tree and IDL file where the IDL file is the FACE Interface IDL file name with the ".idl" extension replaced with "_Injectable.idl".

   Note: As an example, for the FACE Interface that is fully scoped FACE::Configuration declared in FACE/Configuration.idl, the instantiated Injectable Interface module per this requirement is FACE::Configuration_Injectable and the programming language mapping rules are followed as if it was declared in FACE/Configuration_Injectable.idl.

   Note: As an example, for the FACE Interface that is fully scoped FACE::IOSS::Generic::IO_Service declared in FACE/IOSS/Generic.idl, the instantiated Injectable Interface module per this requirement is FACE::IOSS::Generic::IO_Service_Injectable and the programming language mapping

rules are followed as if it was declared in FACE/IOSS/Generic_Injectable.idl. Be aware all interfaces in the IOSS are named IO_Service, but have distinct fully scoped names.

Note: As an example, for the FACE Interface that is fully scoped FACE::TSS::<UOP_MODEL_NAME>::<DATATYPE_TYPE>::TypedTS, the instantiated Injectable Interface module per this requirement is FACE::TSS::<UOP_MODEL_NAME>::<DATATYPE_TYPE>::TypedTS_Injectable and the programming language mapping rules are followed as if it was declared in FACE/TSS/<UOP_MODEL_NAME>/<DATATYPE_TYPE>/TypedTS_Injectable.idl.

4.11.4.2 Unit of Conformance Packaging Requirements

1. A UoC Package shall be composed of UoCs from one of the following combinations of FACE segments:
    a. TSS and PCS
    b. PSSS and TSS
    c. IOSS and PSSS
    d. IOSS, PSSS, and TSS
    e. Multiple UoCs within same FACE segment

    Note: Figure 22 depicts an example of different combinations of UoC Packages.

    Note: PCS UoCs and PSSS UoCs must not be part of the same UoC Package.

    Note: PCS UoCs and IOSS UoCs must not be part of the same UoC Package.
2. All UoCs in a UoC Package shall be designed to operate in the same partition.

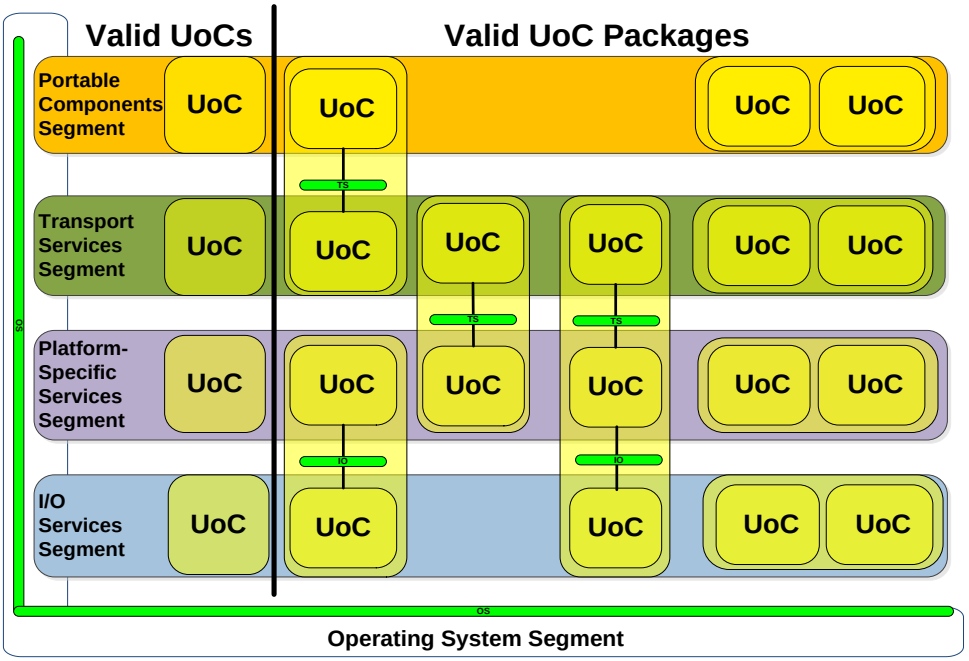

Figure 22: Valid UoC Packaging

4.12 Graphics Services

FACE Graphics Services include the capabilities used to create Human Machine Interfaces (HMI) and can support 2D Rendering, 3D Rendering, Text Rendering, Context Creation, Windowing Capabilities, and Distributed Graphics Rendering. The following standards have been selected to enable the FACE Graphics Services:

- OpenGL/EGL
- ARINC 739
- ARINC 661

There are two classifications of Graphics Services within the FACE Reference Architecture: Graphics Rendering Services and Graphics Display Management Services. Graphics Rendering Services are appropriate for platforms where a single graphics standard needs to be implemented and a single service controls access to the display. Graphics Display Management Services provide for shared display resources and display management.

4.12.1 Graphics Portability Considerations

Graphics Services creates the ability to develop portable UoCs in other segments which need to generate displays. Graphics Services prevents those UoCs from having knowledge about end system displays. Portability is also achieved through the use of well-defined and widely adopted graphics standards. Being widely adopted implies that the latest revision of a standard may not be typically supported due to lack of industry adoption.

4.12.2 Relationship to FACE Reference Architecture

Graphics Services specified in this section can be placed in the FACE Reference Architecture, as shown in Figure 23. A Graphics Services UoC can exist in the PCS or the PSSS. A PCS Graphics Services UoC is required to use a well-defined set of interfaces and graphics standards, such as OpenGL SC, OpenGL ES, ARINC 739 Client, ARINC 739 Server, ARINC 661 Cockpit Display System (CDS), or ARINC 661 UA. A PSSS Graphics Services UoC is subject to the requirements of the PSSS Graphics Services sub-segment. The OSS provides OpenGL/EGL drivers when graphics drivers are needed for the platform.

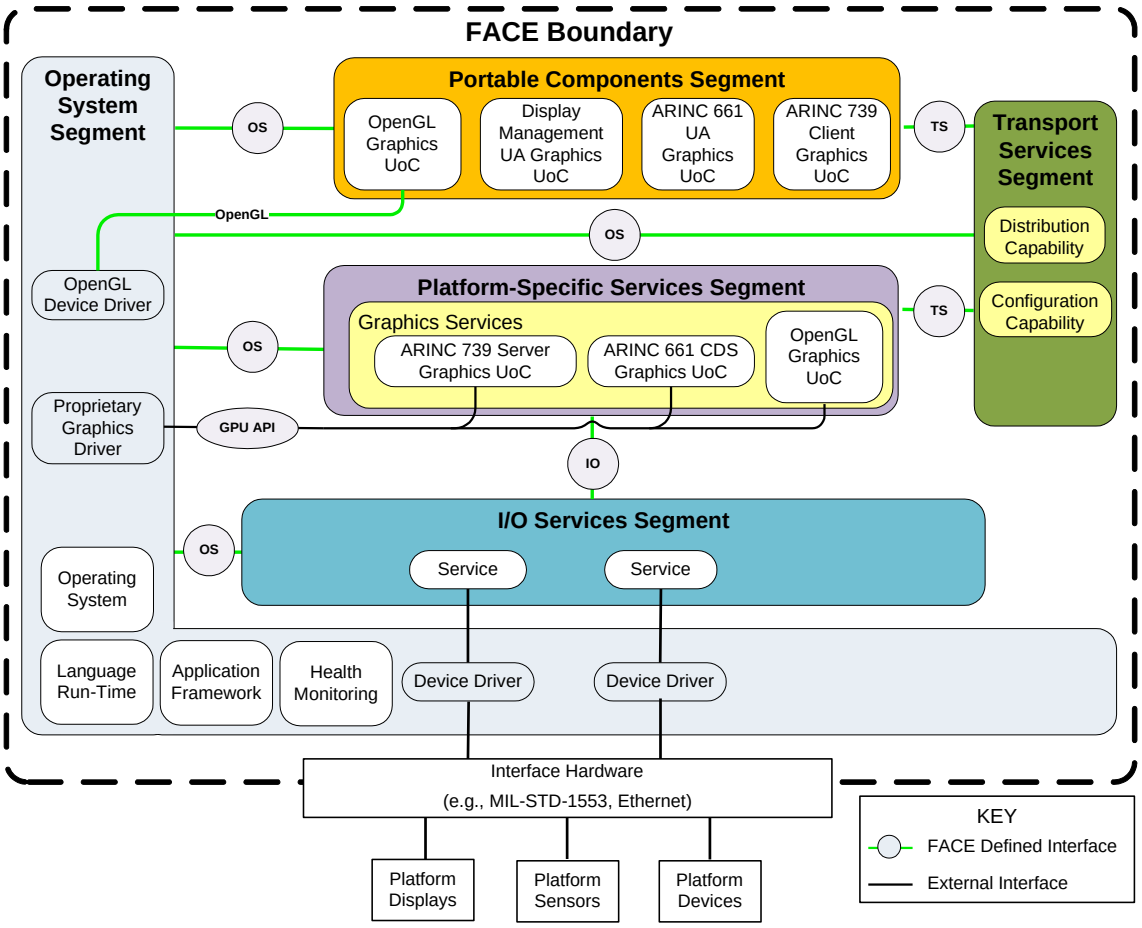

Figure 23: Graphics Services UoCs in the FACE Reference Architecture Context

4.12.3 PSSS Graphics

PSSS Graphics Services sub-segment UoCs enable normalization of implementation-specific graphics rendering APIs. The FACE Technical Standard enables a PSSS Graphics Service UoC to abstract services beyond the graphics standards specified by the FACE Technical Standard.

4.12.4 Graphics Services

Table 9 defines the graphics standards supported for each Graphics Service. Graphics Services contribute to multiple FACE segments: OSS, PSSS, and PCS. When configuring a platform using more than one of the graphics standards, or when it is required that a platform has the capability for Graphics UoCs to be added in the future, it is recommended that the platform provide Graphics Display Management Services.

Table 9: Graphics Services

Screen Sharing Approach	Description	Platform Provided Graphics Services	Graphics Standard
Cooperative screen sharing Applications need knowledge of other applications sharing the same screen.	No support is provided by the system for sharing the display screen between Graphics Services UoCs. Graphics Services UoCs can be used on a platform with Basic Graphics Services implementing one standard for all Graphics UoCs.	Graphics Rendering Services	ARINC 661 UAs + CDS or OpenGL + EGL or ARINC 739A
Facilitated screen sharing A separate function needs to facilitate the sharing of the screen without an application requiring knowledge.	ARINC 661 provides system support for the sharing of the display screen between Graphics UoCs. Graphics Services UoCs can be used on a platform with Graphics Display Management Services, providing the platform implements the standards required by all Graphics Services UoCs.	Graphics Display Management Services	ARINC 661 UAs + CDS, plus either one or both of OpenGL + EGL ARINC 739A

4.12.4.1 ARINC 661

The ARINC 661 standard defines interface protocols between an ARINC 661 Cockpit Display System (CDS) and a User Application (UA). UAs transmit data to the CDS, which is responsible for managing the rendering of the widgets defined in the ARINC 661 Definition File (DF) using the graphics infrastructure available on the rendering platform.

Figure 24 provides a visual representation of the relationship of the various ARINC 661 software components in a generic rendering environment. ARINC 661 UAs incorporate the logic which drives run-time changes to the ARINC 661 CDS.

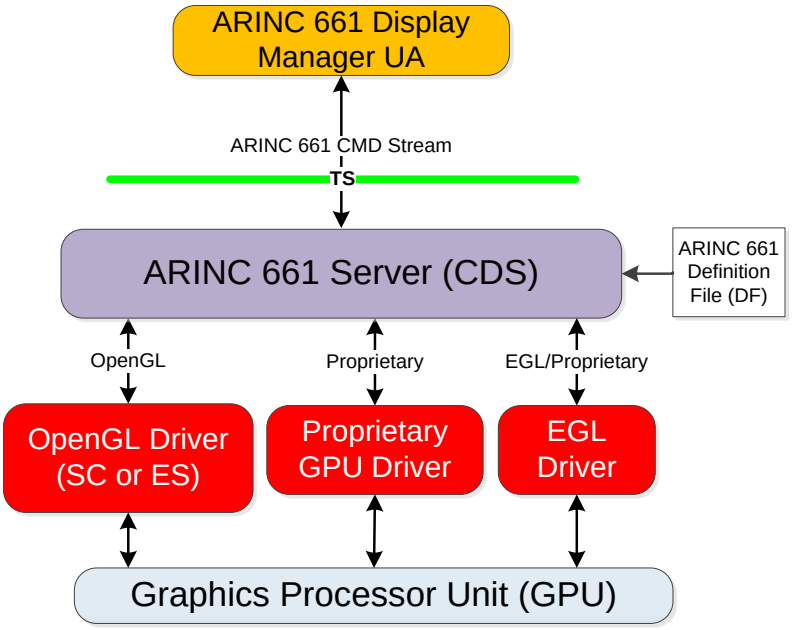

Figure 24: ARINC 661 Graphics Services Relationships

4.12.4.1.1 Applicability in FACE Profiles

ARINC 661 can be used in the following FACE Profiles:

- Security
- Safety
- General Purpose

4.12.4.1.2 ARINC 661 Widgets

ARINC 661 includes a very large set of widgets which are not all necessary in every implementation. In order to reduce the requirement for every implementation to support all widgets, a minimum set of widgets required to be supported by the CDS is defined in Table 10.

Table 10: ARINC 661-5 Widget Subset

Full Widget Set	Minimally Required Widget Subset
Active Area	
Basic Container	x
Blinking Container	x
Buffer Format	x
Check Button	
Combo Box	

Full Widget Set	Minimally Required Widget Subset
Connector	x
Cursor Pos Overlay	
Edit Box Masked	
Edit Box Numeric	
Edit Box Text	
GpArcEllipse	x
GpArcCircle	x
GpCrown	x
GpLine	x
GpLinePolar	x
GpRectangle	x
GpTriangle	x
Picture	x
Label	x
LabelComplex	x
MapHorz_ItemList	
MapHorz_Source	
MapHorz	
Mask Container	x
Panel	x
Picture Push Button	
Picture Toggle Button	
Pop Up Panel	
Pop Up Menu	
Pop Up Menu Button	
Push Button	

Full Widget Set	Minimally Required Widget Subset
Radio Box	
Rotation Container	x
Scroll Panel	
Scroll List	
Symbol	x
Tabbed Panel	
Tabbed Panel Group	
Toggle Button	
Translation Container	x
Map Grid	
External Source	x
Map Vert	
Map Vert Source	
Map Vert Item List	
Edit Box Multi Line	
Combo Box Edit	
Menu Bar	
Mutually Exclusive Container	x
Proxy Button	
Watchdog Container	x
Slider	
Picture Animated	
Symbol Animated	
Selection List Button	
Edit Box Numeric BCD	
Cursor Ref	

Full Widget Set	Minimally Required Widget Subset
Cursor Over	
Focus Link	
Focus In	
Focus Out	
Size to Fit Container	
Shuffle To Fit Container	
Symbol Push Button	
Symbol Toggle Button	
Pop Up Panel Button	
GpPolyline	x
Paging Container	x
Numeric Readout	
Map Horz Container	
Map Horz Panel	
Data Scaling Long	x
Data Scaling Ulong	x
Data Scaling FR180	x
Broadcast Receiver	
No Service Monitor	x

4.12.4.2 ARINC 739A

ARINC 739A defines a message-oriented interface between avionics subsystems and a Multi-purpose Control and Display Unit (MCDU) through an ARINC 429 serial data bus. An MCDU provides a keyboard and a display with line select keys for the display and control of connected subsystems within a hierarchical menu-based structure. ARINC 739A was based on ARINC 739 and included changes to allow for a smaller form factor and included considerations for the MCDU display and dual subsystems installations.

Only the subsystem serial data communication and the related MCDU display functionality may be part of the ARINC 739A Server in the PSSS.

The following specifications in Table 11 define the ARINC 739A functional behavior and protocol.

Table 11: ARINC 739A Functional Behavior Specifications

Function	Specification
Single Subsystems Specification	ARINC 739-1: Multi-purpose Control and Display Unit (MCDU)
Dual Subsystems Specification	ARINC 739A-1: Multi-purpose Control and Display Unit (MCDU)

The FACE General Purpose, Safety, and Security OSS Profile implementations of this graphics interface are identical.

4.12.4.3 OpenGL

The Khronos Open Graphics Language (OpenGL) is a cross-language, cross-platform API for rendering 2D and 3D vector graphics. OpenGL is typically used with a Graphics Processing Unit (GPU) to achieve hardware-accelerated rendering. The Khronos Native Platform Graphics Interface (EGL) is an interface between OpenGL APIs and the underlying native platform windowing system. The Khronos Group manages the OpenGL and EGL specifications. The FACE Technical Standard specifies the use of EGL, OpenGL Safety-Critical (SC), or OpenGL Embedded Systems (ES) profiles.

Graphics Services UoCs using OpenGL make direct calls to the OpenGL and EGL drivers to render graphics content from either the PCS or PSSS segments. Figure 25 illustrates this.

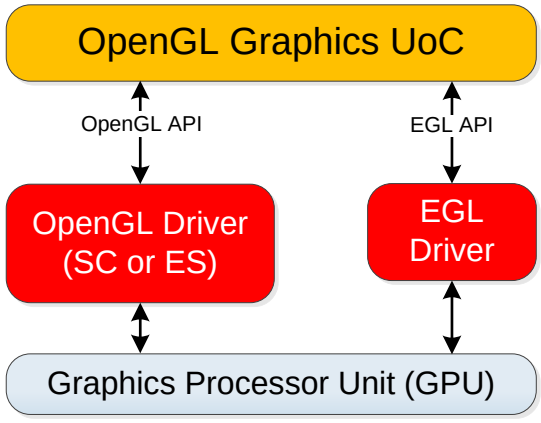

Figure 25: OpenGL Graphics Services UoC

4.12.4.3.1 OpenGL Applicability in FACE Profiles

OpenGL/EGL can be used in the following FACE Profiles:

- Security
- Safety
- General Purpose

4.12.5 Graphics Rendering Services

Graphics Rendering Services are appropriate for platforms where only a single graphics standard is provided and a single service controls access to the display. Graphics Services UoCs rendering to a single screen can be ported between FACE environments with minimal effort.

4.12.6 Graphics Display Management Services

Graphics Display Management Services provide a standard way to share and control graphics resources within a platform. The Graphics Display Management Services reside in the PSSS. It is recommended that platforms provide these services when integrating multiple PCS Graphics Services or UoCs onto a common display.

Display Management is the ability to allocate and control parts of, or the whole display, to a specific Graphic Services UoC. Display Management aims to provide the ability to add Graphics Services UoCs to a system with minimal integration relative to the existing software components. In order to allow future additions of Graphics Services UoCs, the concept of a display manager was created. This method allows most existing Graphics Services UoCs to remain unchanged when adding new Graphics Services UoCs to the system.

ARINC 661 is used for Display Management. This allows OpenGL software components to co-exist with ARINC 661 UA software components. It also provides an enforceable display space partitioning scheme similar to the memory and space partitioning of the operating system. The ARINC 661 CDS provides a standardized interface to control window positioning, focus, and stacking, and a single instance in the FACE Reference Architecture constrains ownership of the display surfaces to the CDS which enforces display space partitioning.

The FACE Technical Standard defines Graphics Display Management Services where an ARINC 661 Display Management UA UoC is responsible for display manager logic. The ARINC 661 Display Management UA is the only software component that needs to be updated to add additional Graphics Services UoCs to a system. An example of a simple display management UA UoC is one which provides the whole screen to a single OpenGL Graphics Services UoC. A complex display management software component may allow several Graphics Services UoCs to share the screen.

4.12.6.1.1 The Display Management Environment

In the Graphics Display Management Services, the ARINC 661 CDS "owns" the display, and is responsible for layout and visibility of everything drawn on the screen, and, in general, is the only software capable of drawing on the screen. ARINC 661 defines the External Source widget as a way to position a video source at a specific location on a display surface. The External Source widget provides a portable means to composite multiple Graphics Services UoCs on a single screen. The External Source widget has defined interfaces to control which Graphics Services UoCs outputs are visible at a given time, the Graphics Services UoC's outputs location and size on the screen, as well as the stacking order of the windows.

The ARINC 661 CDS uses the EGL_EXT_compositor extension to set the size of and allocate off-screen buffers for the OpenGL Graphics Services UoCs. The extension allows for the composition of multiple OpenGL/EGL graphics contexts within a multi-partitioned EGL system. The extension allows a primary *EGLContext* to be created with multiple off-screen windows. The extension provides for asynchronous off-screen window updates and information assurance by the compositor using the primary *EGLContext*. The extension prevents OpenGL Graphics

UoC from interfering with other rendering contexts within the system and from rendering to the primary *EGLContext*.

Therefore, an OpenGL Graphics Services UoC renders using the local OpenGL and EGL APIs. OpenGL Graphics Services UoCs using standard EGL functions are able to query the screen size the ARINC 661 CDS set for the given OpenGL Graphics Service UoC, and have no need to know of any of the other Graphics Services UoCs sharing the same display surface. The OpenGL Graphics Services UoCs render as they normally would; either periodically or event-driven. When an OpenGL Graphics Services UoC finishes rendering a frame, the Graphics Services UoC calls the standard OpenGL and EGL APIs to flush the buffer to the screen. This allows the OpenGL Graphics Services UoC to maintain the highest level of portability.

Figure 26 shows some of the data flows and APIs used to allow OpenGL, ARINC 661, and ARINC 739 to co-exist on the same system. OpenGL Graphics Services UoCs only use OpenGL and EGL APIs to render graphics, which are then placed on the display by the ARINC 661 CDS. ARINC 661 UAs use the ARINC 661 data stream to communicate directly with the ARINC 661 CDS using the TS Interface. The ARINC 661 CDS also communicates window sizing and positioning to the EGL driver such that OpenGL Graphics Services UoCs have correct sizing data using the EGL_EXT_compositor extension.

Figure 26 also shows an ARINC 739 Server Graphics Services UoC using a proprietary interface to the GPU to render its graphic. The ARINC 739-rendered frames are then placed on the display surface by the ARINC 661 CDS according to any display management that may be applied. The ARINC 661 CDS interfaces to the graphics rendering system using any of the available APIs for which a platform may expose to Graphics Services UoCs.

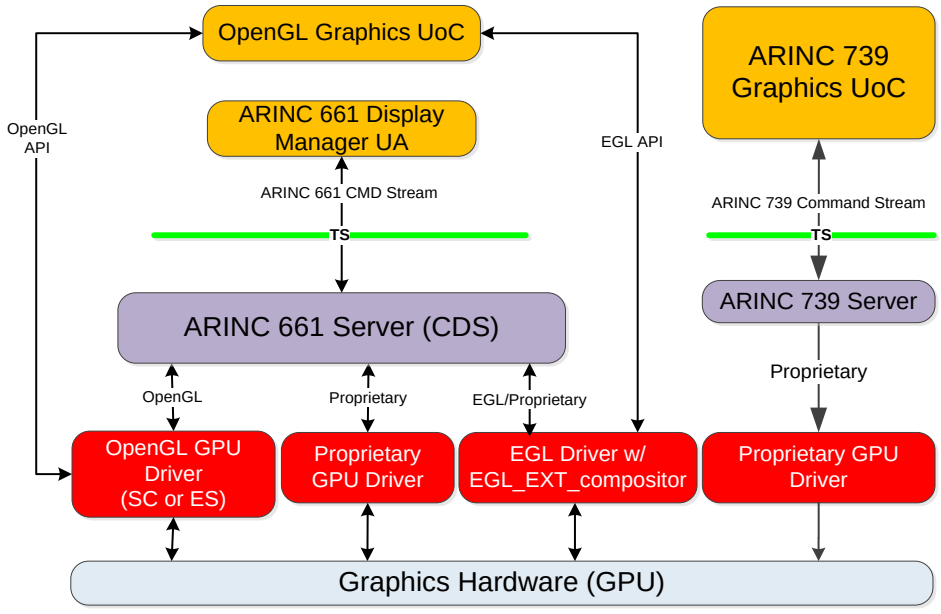

Figure 26: Graphics Services Software Component Relationships

4.12.6.1.2 External Source Widgets

ARINC 661 external source widgets are used to provide window control for rendering objects outside of the other ARINC 661 widgets. Windows are made available to OpenGL via an EGL driver that includes the EGL_EXT_compositor extension. The ARINC 661 CDS uses the same

EGL driver to control the OpenGL rendering areas. In this environment, an ARINC 661 DF file for window management consists of standard ARINC 661 graphics widgets including some number of external source widgets. The windows are then managed by the Display Management UA UoC using the standard ARINC 661 object stacking, location, and visibility rules. The ARINC 661 CDS configures the EGL driver such that OpenGL Graphics Services UoCs can correctly use the screen.

The External Source widget using the EGL_EXT_compositor extension provides a standard interface to specify window location and dimensions for OpenGL frame buffers used by the OpenGL Graphics Services UoCs. This provides a means to control specialized hardware designed to composite external video, or similar software components of non-ARINC 661-generated images or video. The ARINC 661 CDS works with the EGL driver such that OpenGL Graphics Services UoCs can share the screen space. An OpenGL Graphics Services UoC using this EGL API may exist in the PCS or PSSS.

4.12.7 OSS Requirements for Graphics Services

1. When supporting OpenGL, an OSS UoC shall provide OpenGL drivers compatible with OpenGL SC 1.0.1 or OpenGL SC 2.0, or OpenGL ES 2.0.
2. When supporting EGL, an OSS UoC shall provide EGL drivers compatible with EGL 1.4 or EGL 1.4 with the EGL_EXT_compositor extension.
3. When supporting Graphics Display Management Services, an OSS UoC shall provide an EGL driver with the EGL_EXT_compositor extension.

Note: An OpenGL or EGL driver can be provided as part of the OSS. OpenGL and EGL drivers expected to support Graphics UoCs in the PCS or PSSS will need to be able to support OpenGL SC 1.0.1 or OpenGL SC 2.0, or OpenGL ES 2.0 or EGL 1.4, or EGL 1.4 with EGL_EXT_compositor extensions.

4.12.8 PCS Requirements for Graphics Services

4.12.8.1 ARINC 661 Requirements

The FACE General Purpose, Safety, and Security Profile requirements for the ARINC 661 standard are identical and defined here.

4.12.8.1.1 PCS User Application Requirements

1. A Graphics Services UoC implementing ARINC 661 UAs in the PCS shall satisfy the requirements in Section 4.10.
2. A Graphics Services UoC implementing an ARINC 661 UA shall use the TS Interface to communicate ARINC 661 data.
3. A Graphics Services UoC implementing an ARINC 661 UA shall document the set of ARINC 661 widgets it uses.
4. A Graphics Services UoC implementing an ARINC 661 UA shall have an associated ARINC 661 DF.
5. A Graphics Services UoC implementing an ARINC 661 UA shall use the XSD defined in Section H.2 for its style data configuration.

4.12.8.1.2 PCS Cockpit Display System Requirements

1. The ARINC 661 CDS UoC shall provide OpenGL windowing using the external source widget when configured to support Graphics Display Management Services.
2. A Graphics Services UoC implementing an ARINC 661 CDS in the PCS shall support the logic specified in ARINC 661-5 for widgets it provides.
3. A Graphics Services UoC implementing an ARINC 661 CDS in the PCS shall provide the minimum widget subset as specified in Table 10.

   Note: The ARINC 661 CDS may provide additional widgets.
4. A Graphics Services UoC implementing an ARINC 661 CDS in the PCS shall satisfy the requirements in Section 4.10.
5. A Graphics Services UoC implementing an ARINC 661 CDS shall use the TS Interface to communicate ARINC 661 data.
6. A Graphics Services UoC implementing an ARINC 661 CDS shall use the XSD defined in Section H.2 for its style data configuration.
7. A Graphics Services UoC implementing an ARINC 661 CDS shall use the XSD defined in Section H.3 for its display management configuration parameters.
8. A Graphics Services UoC implementing an ARINC 661 CDS shall document the set of ARINC 661 widgets it provides.

4.12.8.2 ARINC 739A Requirements

1. A Graphics Services UoC shall use ARINC 739A messages defined in Section 3.7 of ARINC 739-1 or ARINC 739A-1 when communicating with the ARINC 739A Services.
2. ARINC 739 Client UoCs deployed to the PCS shall satisfy the requirements in Section 4.10.

4.12.8.3 OpenGL Requirements

1. A Graphics Services UoC using OpenGL shall also use EGL, Version 1.4.

   Note: A Graphics Services UoC may only use extended EGL as specified in the OSS graphics requirements.
2. A Graphics Services UoC using OpenGL in the:
   a. General Purpose Profile shall use OpenGL SC 1.0.1, OpenGL SC 2.0, or OpenGL ES 2.0.
   b. Safety Profile shall use OpenGL SC 1.0.1 or OpenGL SC 2.0.
   c. Security Profile shall use OpenGL SC 1.0.1 or OpenGL SC 2.0.

   Note: Graphics Services UoC may not use extended OpenGL and be conformant.
3. A Graphics Services UoC shall be restricted to the core profile for the OpenGL version being used.

   Note: This does not preclude use of dynamic binding to use OpenGL extensions, however the UoC must not rely on the existence of any OpenGL extensions.

4. A Graphics Services UoC using *eglGetDisplay* shall use a configurable parameter for the *display_id* input argument.

   Note: The configurable parameter, for example, could be read from a file or passed in at startup of the software component. See Configuration Services in Section 4.2.5.

5. A Graphics Services UoC using *eglCreateWindowSurface* shall use a configurable parameter for win input argument when using EGL_EXT_compositor as off-screen windows.

   Note: The configurable parameter, for example, could be read from a file or passed in at startup of the software component. See Configuration Services in Section 4.2.5.

6. A Graphics Services UoC using *eglCreateContext* shall use a configurable parameter for the EGL_EXTERNAL_REF_ID_EXT attribute during context creation when not using the EGL_PRIMARY_COMPOSITOR_CONTEXT_EXT attribute.

4.12.9 PSSS Requirements for Graphics Services

This section includes the PSGS sub-segment requirements.

1. When a PSSS Graphics Service UoC renders using a method other than OpenGL, as defined in Section 4.12.7, the Graphics Service UoC shall use the graphics driver API.

   Note: This allows a PSSS Graphics Service UoC to use interfaces not defined in the IOS or OSS APIs for graphics rendering.

2. ARINC 661 Requirements

   The FACE General Purpose, Safety, and Security Profile requirements for the ARINC 661 standard are identical and defined here.

4.12.9.1 PSSS Cockpit Display System Requirements

1. The ARINC 661 CDS UoC shall provide OpenGL windowing using the external source widget when configured to support Graphics Display Management Services.

2. A Graphics Services UoC implementing an ARINC 661 CDS in the PSSS shall support the logic specified in ARINC 661-5 for widgets it provides.

3. A Graphics Services UoC implementing an ARINC 661 CDS in the PSSS shall provide the minimum widget subset as specified in Table 10.

   Note: The ARINC 661 CDS may provide additional widgets.

4. A Graphics Services UoC implementing an ARINC 661 CDS shall use the TS Interface to communicate ARINC 661 data.

5. A Graphics Services UoC implementing an ARINC 661 CDS shall use the XSD defined in Section H.2 for its style data configuration.

6. A Graphics Services UoC implementing an ARINC 661 CDS shall use the XSD defined in Section H.3 for its display management configuration parameters.

   Note: The Graphics Services UoC implementing an ARINC 661 CDS in the PSSS may have direct access to any proprietary graphics hardware drivers and APIs.

7. A Graphics Services UoC implementing an ARINC 661 Display Management UA shall have an associated ARINC 661 DF.

8. A Graphics Services UoC implementing an ARINC 661 CDS shall document the set of ARINC 661 widgets it provides.

4.12.9.2 PSSS User Application Requirements

1. A Graphics Services UoC implementing an ARINC 661 UA shall use the TS Interface to communicate ARINC 661 data.

2. A Graphics Services UoC implementing an ARINC 661 UA shall document the set of ARINC 661 widgets it uses.

3. A Graphics Services UoC implementing an ARINC 661 UA shall use the XSD defined in Section H.2 for its style data configuration.

4.12.9.3 ARINC 739A Requirements

1. A Graphics Services UoC shall use ARINC 739A messages defined in Section 3.7 of ARINC 739-1 or ARINC 739A-1 when communicating with the ARINC 739A Services.

4.12.9.4 OpenGL Requirements

The requirements for OpenGL apply to both PCS and PSSS subsections of the FACE Technical Standard.

1. A Graphics Services UoC using OpenGL shall also use EGL, Version 1.4.

   Note: A Graphics Services UoC may only use extended EGL as specified in the OSS graphics requirements.

2. A Graphics Services UoC using OpenGL in the:

   a. General Purpose Profile shall use OpenGL SC 1.0.1, OpenGL SC 2.0, or OpenGL ES 2.0.

   b. Safety Profile shall use OpenGL SC 1.0.1 or OpenGL SC 2.0.

   c. Security Profile shall use OpenGL SC 1.0.1 or OpenGL SC 2.0.

   Note: Graphics Services UoC may not use extended OpenGL and be conformant.

3. A Graphics Services UoC shall be restricted to the core profile for the OpenGL version being used.

   Note: This does not preclude use of dynamic binding to use OpenGL extensions; however, the UoC must not rely on the existence of any OpenGL extensions.

4. A Graphics Services UoC using *eglGetDisplay* shall use a configurable parameter for the *display_id* input argument.

   Note: The configurable parameter, for example, could be read from a file or passed in at startup of the software component. See Configuration Services.

5. When using EGL_EXT_compositor as off-screen windows, a Graphics Services UoC using *eglCreateWindowSurface* shall use a configurable parameter for the win input argument.

Note: The configurable parameter, for example, could be read from a file or passed in at startup of the software component. See Configuration Services.

6. When calling *eglCreateContext* and not using the EGL_PRIMARY_COMPOSITOR_CONTEXT_EXT attribute, a Graphics Services UoC shall use a configurable parameter for the EGL_EXTERNAL_REF_ID_EXT attribute.

7. A Graphics Services UoC using OpenGL shall document its OpenGL version.

4.13 Life Cycle Management Services

4.13.1 Introduction

The Life Cycle Management Services provide a common and consistent approach for managing the life-cycle of UoCs. Software Suppliers and System Integrators can depend on the common interface to support their Life Cycle Management (LCM) solution implementation.

4.13.1.1 Goals

The LCM Services are defined to provide a consistent approach for software suppliers and system integrators to address several complexities that come with integrating UoCs into a system. Some of the Life Cycle integration complexities include:

- An instance of the UoC is initialized and finalized at execution points that are highly dependent on the system design
- An instance of the UoC is configured at execution points that are highly dependent on the system design
- An instance of the UoC is integrated when a component framework solution is used, where characteristics of that component framework solution are highly dependent on the system design
- An instance of the UoC is integrated with state management algorithms that are highly dependent on the system design
- The system coordinates the state of multiple UoC instances, where the possible states and valid transitions are highly dependent upon each UoC

A UoC that addresses these complexities via the LCM Services provides more flexibility for reuse. Systems that employ the LCM Services to address these complexities may reap cost and schedule benefits from the consistent technical integration of multiple UoCs.

4.13.1.2 Approach

There are four capabilities offered by LCM Services:

- Initializable Capability (Section 4.13.2)
- Configurable Capability (Section 4.13.3)
- Connectable Capability (Section 4.13.4)
- Stateful Capability (Section 4.13.5)

The Capabilities are independent and optional. Each Capability is defined by a corresponding Interface. A UoC that provides one or more of the LCM Capabilities is referred to as a Managed UoC in this context.

LCM Services does not define an interface to create or destroy instances. Section 4.14 addresses the syntax for those operations. LCM Services assumes an existent software object that implements one or more of the interfaces defined.

The time-ordered sequence of execution points supported by LCM Services is as follows:

1. Initialization via the Initializable Capability
2. Configuration via the Configurable Capability
3. Framework startup via the Connectable Capability
4. State transitions via the Stateful Capability
5. Framework teardown via the Connectable Capability
6. Finalization via the Initializable Capability

The Stateful capability uses parameterized data types for state representation, allowing each Managed UoC to define its own valid state values. The FACE Technical Standard need not, therefore, require any specific state representations or transitions. The state of the system becomes a function in part of the states of managed UoCs, and appropriate state transition behavior remains an aspect of system design. The SDM does define a state representation that may be suitable for adoption by a Managed UoC when additional states are not required.

4.13.1.3 Security and Safety Considerations

The capabilities provided by LCM Services coincide with cross-cutting security and safety assurance concerns. The existence of LCM Services in the FACE Technical Standard does not change the continued need for assurance case analysis. Factors involved include, but are not limited to, the assurance level of the system and its constituent UoCs, the criticality requirements of the system and consequent design decisions, and criteria established by the designated assurance authority.

In Real Time Safety-Critical (RTSC) systems, reliable operation and integrity of safeguards during post-startup operation must be assured. Use of LCM Services Interfaces is constrained to preclude changes in executable memory space and execution scheduling for RTSC systems. Use of the Initializable, Configurable, Connectable, and Stateful Capabilities is constrained to system startup/shutdown to ensure that a single known software configuration exists during operation.

In security-critical systems, employing LCM Services needs to be assured as part of the system design to minimize vulnerabilities. Any of the LCM Services Capabilities could be exploited by malicious or unstable software which could compromise the proper behavior of the system.

4.13.2 Initializable Capability Requirements

The Initializable Capability provides an entry point to an instance of a Managed UoC at the initialization and finalization execution points.

1. A UoC that provides the LCM Services Initializable Capability shall provide the LCM Services Initializable Interface per Section D.2.

2. A UoC that uses the LCM Services Initializable Capability shall use the LCM Services Initializable Interface per Section D.2.

4.13.3 Configurable Capability Requirements

The Configurable Capability supports the configuration execution point for an instance of a Managed UoC. This execution point is intended to support parameters specific to the Managed UoC that are not associated with configuration behavior defined by other FACE Interfaces.

1. A UoC that provides the LCM Services Configurable Capability shall provide the LCM Services Configurable Interface per Section D.3.

2. A UoC that uses the LCM Services Configurable Capability shall use the LCM Services Configurable Interface per Section D.3.

4.13.4 Connectable Capability Requirements

The Connectable Capability provides an entry point to an instance of a Managed UoC at the framework startup and framework teardown execution points.

1. A UoC that provides the LCM Services Connectable Capability shall provide the LCM Connectable Services Interface per Section D.4.

2. A UoC that uses the LCM Services Connectable Capability shall use the LCM Services Connectable Interface per Section D.4.

4.13.5 Stateful Capability Requirements

The Stateful Capability supports state transition execution points for an instance of a Managed UoC.

1. A UoC that provides the LCM Services Stateful Capability shall provide the LCM Services Stateful Interface per Section D.5.

2. A UoC that uses the LCM Services Stateful Capability shall use the LCM Services Stateful Interface per Section D.5.

3. The REPORTED_STATE_VALUE_TYPE parameter of the LCM Stateful Interface shall be a UoPModel Template.

4. The REQUESTED_STATE_VALUE_TYPE parameter of the LCM Stateful Interface shall be a UoPModel Template.

5. The fully qualified name of the instantiated LCM Stateful Interface module shall be FACE::LCM::<UOP_MODEL_NAME>::<REQUESTED_DATA_TYPE>_<REPORTED_DATA_TYPE>::StatefulInstance.

   Note: Most of the time, REQUESTED_DATA_TYPE and REPORTED_DATA_TYPE are in the same model. REQUESTED_DATA_TYPE and REPORTED_DATA_TYPE are the short names of a Template or CompositeTemplate, not fully qualified names. UOP_MODEL_NAME is the name of the root UoPModel in which DATATYPE_TYPE or RESPONSE_DATATYPE is a member. The resulting declarations follow programming language mapping rules as if the interface was declared in a directory tree and IDL file as

FACE/LCM/<UOP_MODEL_NAME>/<REQUESTED_DATA_TYPE>_<REPORTED_DATA_TYPE>/Stateful.idl.

4.14 IDL to Programming Language Mappings

Several FACE Standardized Interfaces and the FACE Modeling Language IDL bindings are specified using the OMG Interface Definition Language (IDL), Version 4.1. The portion of the IDL language used in these specifications is a profile consisting of the following building blocks:

- Building Block Core Data Types
- Building Block Interfaces – Basic
- Building Block Interfaces – Full
- Building Block Template Modules

This section describes language mappings from this profile to several programming languages.

Note: Throughout this section, the phrase "IDL compiler" is used to refer to an IDL compiler that supports these Language Mappings.

Note: The FACE Data Architecture and IDL differ in the scoping rules applied to enumerators. In order to accommodate use cases where state/sub-state representations with enumerations result in colliding identifiers in the same IDL scope, the FACE Technical Standard permits IDL enumerators in the same module scope to have the same name. This is an exception to the identifier uniqueness rule in IDL 4.1 Section 7.2.3.1. The programming language mapping rules ensure enumerators are distinct fully scoped identifiers.

4.14.1 Exceptions

IDL exceptions map to nothing in every target language. IDL interface attributes or operations that raise exceptions map to every target language as if they did not raise an exception.

4.14.2 Template Modules

Template modules are similar to C++ templates, in that they allow a module and its contents to be parameterized. A template module is not actually used until it is instantiated with appropriate parameters, so template module definitions and their contents do not map to anything until they are instantiated. Once a template module has been instantiated, it is treated exactly as a classical module would be treated if defined at the point the template module was instantiated.

Template Module Example

```
// IDL
module A<typename T> {
  typedef T MyThing;
};

module B {
  module A<long> MyModuleA;
};
```

```
// (treated as a classical module at the point of instantiation)
module B {
  module MyModuleA {
    typedef long MyThing;
  };
};
```

4.14.3 Constants

These mappings do not support IDL constants whose constant expression evaluates to a string or fixed-point. In such cases, an IDL compiler emits a diagnostic indicating the use of an unsupported construct.

4.14.4 Constant Expressions

A constant expression in IDL maps to every target language either as a literal that is semantically equivalent in value and type to the result of the IDL expression's evaluation, as described in Section 7.4.1.4.3 of the IDL 4.1 Specification, or an expression that mimics the form and semantics of the IDL expression and results in the same value.

Constant Expression Example

```
// IDL
1 + 1

// Permissible mapping to C
2

// Another permissible mapping to C
1 + 1
```

Note: An IDL constant definition is invalid if its expression is outside its type's range. In such cases, an IDL compiler emits a diagnostic indicating the error.

Invalid IDL Constant Definition Example

```
// IDL
const short Foo = 1 + 65535; /* error – result outside
                                      range of IDL short */
```

4.14.5 Preprocessor Directives

The only preprocessor directives supported are:

```
#include <...>
#include "..."
#ifndef identifier
#define identifier
#endif
#pragma FACE include_guard
```

4.14.6 Wide Characters and Wide Strings

These mappings do not support IDL wide characters or wide strings.

4.14.7 IDL to C Mapping

This section describes a mapping from IDL 4.1 to C99 (ISO/IEC 9899:1999: Programming Languages – C). It is based on mappings defined in OMG IDL C Language Mapping, Version 1.0.

4.14.7.1 Names

Names in IDL generally obey C naming conventions, with exceptions described below. Unless otherwise excepted below, unscoped names in IDL appear in the generated source code character-for-character.

In the event that a name that is legal in IDL conflicts with a reserved C keyword, the name for that symbol is constructed by prefixing it with "FACE_". Any language in the remainder of this section indicating that a C construct has the "same name" as its IDL counterpart takes this conflict resolution into account.

4.14.7.1.1 Scoped Names

IDL provides a scoping mechanism similar to C++: modules map roughly to namespaces and interfaces can open new scopes similar to C++ classes. As the C programming language lacks these features, any name that is not in the global scope is constructed to ensure uniqueness. The enclosing scope of a particular name is prepended to that name with an underscore.

Scoped Names Example 1

```
module A {
  module B {
    const long Foo = 0;
    interface Bar {
      void baz();
    };
 };
};
```

The scoped names are as follows:

- Foo: A_B_Foo
- Bar: A_B_Bar
- Baz: A_B_Bar_baz

Because of these constructed names, using underscores in IDL identifiers may result in duplicate symbols in C. In the following example, both "foo_bar" and "bar" typedefs map to "foo_bar" in C, because "bar" is in the "foo" module. An IDL compiler emits an error if such a conflict occurs.

Scoped Names Example 2

```
// IDL
typedef long foo_bar;
module foo {
  typedef short bar; /* Legal IDL, but would cause duplicate symbol in C.
                        Emit error. */
};
```

4.14.7.1.2 File Names

An IDL source file maps to a C header file with the same base name and an ".h" extension. The header file is in a subdirectory tree which reflects the subdirectory tree for the IDL source file. For example, if the input IDL file is Foo/Bar/Sample.idl, then the generated C header file will be Foo/Bar/Sample.h.

Note: A subdirectory tree for an input IDL source file and an output C header file is relative to a base directory that is not specified. The base directory for the source IDL file may be different from the base directory for the output C header file.

4.14.7.2 Preprocessor Directives

A #ifndef include guard is always present in a C header file generated from IDL. The identifier used in the guard is in capital letters. It consists of a leading underscore, the IDL subdirectory tree with the components capitalized, separated by an underscore, followed by the capitalized base name of the resulting C header file, and ending with "_H". For example, Foo/Bar/Sample.idl will have an include guard of _FOO_BAR_SAMPLE_H.

Other IDL preprocessor directives do not map to anything in the C representation.

4.14.7.3 Modules

Modules have no corresponding C language construct and thus do not appear directly as generated code. Instead, modules influence the scoped name generated for any contained elements as described in Section 4.14.7.1. Scoped names are used when resolving definitions from other modules.

Modules Example

```
// IDL
module A {
  typedef long MyLong;
  module B {
    typedef MyLong MyOtherLong;
  };
};

module C {
  typedef A::MyLong MyOtherLong;
};

// C
typedef FACE_long A_MyLong;
typedef A_MyLong A_B_MyOtherLong;
typedef A_MyLong C_MyOtherLong;
```

4.14.7.4 Typedefs

An IDL typedef creates an alias for a type; it maps directly to a C typedef. The C typedef's alias is the scoped name of the IDL typedef, as described in Section 4.14.7.1; the C typedef type's mapping is specified in the IDL type's relevant section below. Multiple declarators in IDL map to the same in C.

Typedef Example 1

```
// IDL
typedef long Foo, Bar;
module A {
  typedef Bar Baz;
};

// C
typedef FACE_long Foo, Bar;
typedef Bar A_Baz;
```

Structures, unions, and enumerations can be declared within a typedef in IDL, which is logically equivalent to a type declaration immediately followed by a typedef.

Typedef Example 2

```
// IDL
typedef struct Foo_struct { long x; } Foo;

// (logically equivalent to the following IDL)
struct Foo_struct { long x; };
typedef Foo_struct Foo;
```

4.14.7.5 Constants

A constant in IDL is mapped to a #define in C. The #define identifier is the fully scoped name of the IDL constant, as specified in Section 4.14.7.1. If the IDL constant's value expression is a scoped name, then the #define replacement is mapped from that scoped name as specified in Section 4.14.7.1. Otherwise, the #define replacement is a C expression mapped from the IDL constant expression as specified in Section 4.14.4. In all cases, the #define replacement includes a cast to the fully-qualified type mapped from the IDL constant's type as specified elsewhere.

Constants Example

```
// IDL
typedef long MyLong;
enum Color {RED, GREEN, BLUE};
module A {
  const long FooLong = 1 + 65535;
  const boolean FooBool = TRUE;
  const MyLong FooMyLong = FooLong;
  const Color clr = RED;
};

// C
typedef FACE_long MyLong;
typedef enum Color {Color_RED, Color_GREEN, Color_BLUE} Color;
#define A_FooLong ((FACE_long)65536)
#define A_FooBool ((FACE_boolean)true)
#define A_FooMyLong ((MyLong)A_FooLong)
#define A_clr ((Color)Color_RED)
```

4.14.7.6 Simple Types

4.14.7.6.1 Basic Types

IDL Basic Types map to C types according to Table 12. Implementations provide definitions for these C types that align with the given size and range requirements. The file containing these definitions is "FACE/types.h", as specified in Section K.1.1.

Table 12: IDL Basic Type C Mapping

IDL Basic Type	C Type	Size (bytes)	Range of Values	Default Value
short	FACE_short	2	-2^15 to (2^15 - 1)	0
long	FACE_long	4	-2^31 to (2^31 - 1)	0
long long	FACE_long_long	8	-2^63 to (2^63 - 1)	0
unsigned short	FACE_unsigned_short	2	0 to (2^16 - 1)	0
unsigned long	FACE_unsigned_long	4	0 to (2^32 - 1)	0
unsigned long long	FACE_unsigned_long_long	8	0 to (2^64 - 1)	0
float	FACE_float	4	IEEE 754-2008 single precision floating point	0.0
double	FACE_double	8	IEEE 754-2008 double precision floating point	0.0
long double	FACE_long_double	10	IEEE 754-2008 extended double precision floating point	0.0
char	FACE_char	1	-2^7 to (2^7 - 1)	0
boolean	FACE_boolean	1	0 to 1	0
octet	FACE_octet	1	0 to (2^8 - 1)	0

4.14.7.6.2 Sequences

Bounded and unbounded sequences map to a typedef FACE_sequence and a set of macros that wrap FACE_sequence functions. There is one macro per FACE_sequence function. The #define identifier for each macro is <derived function name>(<derived parameter list>), and the replacement is <FACE_sequence function name>(<parameter list>), where:

- <FACE_sequence function name> is the name of the FACE_sequence function
- <derived function name> is <FACE_sequence function name>, with "FACE_sequence" replaced by the fully-scoped name of the IDL sequence

- <parameter list> is a comma-separated list of all the FACE_sequence function's parameter names, each prepended with an underscore and enclosed in parentheses

  Any FACE_sequence parameter named sizeof_T is handled differently, becoming sizeof(<element type>) in this list, where <element type> is mapped from the IDL sequence's type as specified elsewhere.

- <derived parameter list> is <parameter list>, with sizeof(<element type>) omitted from the list, and without parenthesis enclosing each parameter name

Full specification for FACE_sequence is in Section K.1.3.

Sequences also map to a #define in C for the sequence's bound, where the #define identifier is the sequence's fully-scoped name appended with "_bound_value", and the #define's replacement is the sequence's maximum size cast to a FACE_unsigned_long. To indicate that a sequence is unbounded, the sentinel value FACE_SEQUENCE_UNBOUNDED_SENTINEL is used as the replacement value.

Unbounded Sequence Mapping

```
// IDL
typedef short TYPE;
typedef sequence<TYPE> Foo;
typedef sequence<TYPE,8> Bar;

// C
typedef FACE_short TYPE;

typedef FACE_sequence Foo;

#define Foo_bound_value FACE_SEQUENCE_UNBOUNDED_SENTINEL

#define Foo_init_managed_unbounded(_seq) \
  FACE_sequence_init_managed_unbounded((_seq), sizeof(TYPE))

#define Foo_init_managed_bounded(_seq, _bound) \
  FACE_sequence_init_managed_bounded((_seq), sizeof(TYPE), (_bound))

#define Foo_init_managed_copy(_seq, _src) \
  FACE_sequence_init_managed_copy((_seq), (_src))

#define Foo_init_managed_data(_seq, _arr, _length) \
  FACE_sequence_init_managed_data((_seq), (_arr), sizeof(TYPE), (_length))

#define Foo_init_unmanaged(_seq, _src, _length, _bound) \
  FACE_sequence_init_unmanaged( \
    (_seq), (_src), sizeof(TYPE), (_length), (_bound))

#define Foo_free(_seq) \
  FACE_sequence_free((_seq))

#define Foo_clear(_seq) \
  FACE_sequence_clear((_seq))

#define Foo_append(_seq, _src) \
  FACE_sequence_append((_seq), (_src))

#define Foo_at(_seq, _index) \
  (TYPE *) FACE_sequence_at((_seq), (_index))
```

```
#define Foo_buffer(_seq) \
  (TYPE *) FACE_sequence_buffer((_seq))

#define Foo_length(_seq, _length) \
  FACE_sequence_length((_seq), (_length))

#define Foo_capacity(_seq, _capacity) \
  FACE_sequence_capacity((_seq), (_capacity))

#define Foo_bound(_seq, _bound) \
  FACE_sequence_bound((_seq), (_bound))

#define Foo_is_managed(_seq, _is_managed) \
  FACE_sequence_is_managed((_seq), (_is_managed))

#define Foo_is_bounded(_seq, _is_bounded) \
  FACE_sequence_is_bounded((_seq), (_is_bounded))

#define Foo_is_valid(_seq, _is_valid) \
  FACE_sequence_is_valid((_seq), (_is_valid))

typedef FACE_sequence Bar;

#define Bar_bound_value ((FACE_unsigned_long)8)

#define Bar_init_managed_unbounded(_seq) \
  FACE_sequence_init_managed_unbounded((_seq), sizeof(TYPE))

#define Bar_init_managed_bounded(_seq, _bound) \
  FACE_sequence_init_managed_bounded((_seq), sizeof(TYPE), (_bound))

#define Bar_init_managed_copy(_seq, _src) \
  FACE_sequence_init_managed_copy((_seq), (_src))

#define Bar_init_managed_data(_seq, _arr, _length) \
  FACE_sequence_init_managed_data((_seq), (_arr), sizeof(TYPE), (_length))

#define Bar_init_unmanaged(_seq, _src, _length, _bound) \
  FACE_sequence_init_unmanaged( \
    (_seq), (_src), sizeof(TYPE), (_length), (_bound))

#define Bar_free(_seq) \
  FACE_sequence_free((_seq))

#define Bar_clear(_seq) \
  FACE_sequence_clear((_seq))

#define Bar_append(_seq, _src) \
  FACE_sequence_append((_seq), (_src))

#define Bar_at(_seq, _index) \
  (TYPE *) FACE_sequence_at((_seq), (_index))

#define Bar_buffer(_seq) \
  (TYPE *) FACE_sequence_buffer((_seq))

#define Bar_length(_seq, _length) \
  FACE_sequence_length((_seq), (_length))

#define Bar_capacity(_seq, _capacity) \
  FACE_sequence_capacity((_seq), (_capacity))

#define Bar_bound(_seq, _bound) \
```

```
  FACE_sequence_bound((_seq), (_bound))

#define Bar_is_managed(_seq, _is_managed) \
  FACE_sequence_is_managed((_seq), (_is_managed))

#define Bar_is_bounded(_seq, _is_bounded) \
  FACE_sequence_is_bounded((_seq), (_is_bounded))

#define Bar_is_valid(_seq, _is_valid) \
  FACE_sequence_is_valid((_seq), (_is_valid))
```

4.14.7.6.3 Strings

Bounded and unbounded strings map to a typedef FACE_string and a #define in C, where the #define identifier is the string's fully-scoped name appended with "_bound_value", and the #define's replacement is the string's maximum size cast to a FACE_unsigned_long. To indicate that a string is unbounded, the sentinel value FACE_STRING_UNBOUNDED_SENTINEL is used as the replacement value. Full specification for FACE_string is in Section K.1.4.

String Example

```
// IDL
typedef string Foo;
typedef string<8> Bar;

// C
typedef FACE_string Foo;

#define Foo_bound_value FACE_STRING_UNBOUNDED_SENTINEL
typedef FACE_string Bar;
#define Bar_bound_value ((FACE_unsigned_long)8)
```

4.14.7.6.4 Fixed

A fixed type maps to a typedef FACE_fixed and two #defines to represent the type's digits and scale. For the digits #define, the identifier is the fixed type's fully-scoped name appended with "_digits", and the replacement is the type's total number of digits cast to a FACE_unsigned_short. For the scale #define, the identifier is the fixed type's fully-scoped name appended with "_scale", and the replacement is the type's scale cast to a FACE_unsigned_short. Implementations are responsible for initializing a fixed type using these constants. Full specification for FACE_fixed is in Section K.1.5.

Fixed Type Example

```
// IDL
typedef fixed<5,2> Foo;

// C
typedef FACE_fixed Foo;
#define Foo_digits ((FACE_unsigned_short) 5)
#define Foo_scale ((FACE_unsigned_short) 2)
```

4.14.7.7 Constructed Types

4.14.7.7.1 Structures

An IDL structure maps to a C structure typedef. The C structure and typedef alias names are the scoped name of the IDL structure, as specified in Section 4.14.7.1. The structure's members

occur in the same order as in IDL; each member's type and identifier are mapped as specified elsewhere.

```
// IDL
typedef long MyLong;

struct A {
  long X;
  MyLong Y;
  char Z;
};

// C
typedef FACE_long MyLong;

typedef struct A {
  FACE_long X;
  MyLong Y;
  FACE_char Z;
} A;
```

4.14.7.7.2 Enumerations

An IDL enum maps to a C enum typedef. The C enum and typedef alias names are the scoped name of the IDL enum, as specified in Section 4.14.7.1. Enum literals map one-to-one from IDL; enum literal names are the same as in IDL, prepended with <fully-scoped enum name>_.

Enumeration Example

```
// IDL
enum Color {RED, GREEN, BLUE};
// C
typedef enum Color {Color_RED, Color_GREEN, Color_BLUE} Color;
```

4.14.7.7.3 Unions

An IDL union maps to a C structure typedef. The C structure and typedef alias names are the IDL union's scoped name, as described in Section 4.14.7.1.1. The structure contains two members: a discriminator named "discriminator" and a union named "values". The type of the discriminator is derived from the type of the IDL union's discriminator as specified elsewhere. Each IDL union member maps to one member in the C union. Each member's type and identifier are mapped as specified elsewhere.

Note: Implementations are responsible for consistently modifying the discriminator and union. It is recommended that comments be used to document which C union member corresponds to which discriminator value.

Union Example

```
// IDL
enum CASES { FOO, BAR, BAZ };

union FooUnion switch (CASES) {
  case FOO: short a;
  case BAR: long b;
  // NOTE: IDL does not require a case for every enum literal
};

// C
```

```
typedef enum CASES { CASES_FOO, CASES_BAR, CASES_BAZ } CASES;

typedef struct FooUnion {
  enum CASES discriminator;
  union {
    FACE_short a; // CASES_FOO
    FACE_long b; // CASES_BAR
  } values;
} FooUnion;
```

4.14.7.8 Arrays

An array in IDL maps to a C-style array of the same dimension. The name of the array is the scoped name of the IDL array as described in Section 4.14.7.1; the C array type's mapping is specified in the IDL type's relevant section.

Array Example

```
// IDL
typedef short Foo[10];

typedef short Bar[4][5];

// C
typedef FACE_short Foo[10];

typedef FACE_short Bar[4][5];
```

4.14.7.9 Interfaces

4.14.7.9.1 Declaration

An IDL interface definition is mapped to a pair of C struct typedefs: one whose name is the fully scoped name of the IDL interface (see Section 4.14.7.1), acting as the main data structure for the interface; another whose name is constructed by appending _ops to the previous name, acting as an operation lookup table for the interface.

An IDL interface definition is also mapped to two typedefs for function pointers – one for initialization of the interface and one for cleanup. The typedef alias for the initialization function pointer is <fully-scoped interface name>_ctor_t; the typedef alias for the cleanup function pointer is <fully-scoped interface name>_dtor_t. Both function pointers return FACE_interface_return and have one parameter named this_obj whose type is a pointer to the interface struct. An interface's behavior is implementation-defined if its initialization function does not return FACE_INTERFACE_NO_ERROR. FACE_interface_return is defined in "FACE/interface.h", as specified in Section K.1.2.

An IDL interface is also mapped to a #define macro for each of these two function pointers. These macros hide the operation table lookup, allowing simpler code in business logic that uses the interface. The #define identifier for the initialization function is <fully-scoped interface name>_ctor(_this_obj), and the replacement is ((_this_obj)->ops.ctor)((_this_obj)). The #define identifier and replacement for the cleanup function is the same, except with dtor instead of ctor.

The main interface struct contains an operations table struct named ops, followed by a void pointer named data for private data. The operations table struct contains an initialization function pointer, followed by a cleanup function pointer – both members have the same name as their respective function pointer type.

An interface may also be declared with a forward declaration, in which case it maps to a forward declaration of a struct in C.

Interface Declaration Example

```
// IDL
interface Bar;
interface Foo {};
interface Bar {};

// C
struct Bar;

struct Foo;
// initialize this_obj->data
typedef FACE_interface_return (*Foo_ctor_t)(struct Foo* this_obj);
// clean up this_obj->data
typedef FACE_interface_return (*Foo_dtor_t)(struct Foo* this_obj);
typedef struct Foo_ops {
  Foo_ctor_t ctor;
  Foo_dtor_t dtor;
} Foo_ops;
typedef struct Foo {
  Foo_ops ops;
  void* data;
} Foo;
#define Foo_ctor(_this_obj) \
  ((_this_obj)->ops.ctor)((_this_obj))
#define Foo_dtor(_this_obj) \
  ((_this_obj)->ops.dtor)((_this_obj))

struct Bar;
// initialize this_obj->data
typedef FACE_interface_return (*Bar_ctor_t)(struct Bar* this_obj);
// clean up this_obj->data
typedef FACE_interface_return (*Bar_dtor_t)(struct Bar* this_obj);
typedef struct Bar_ops {
  Bar_ctor_t ctor;
  Bar_dtor_t dtor;
} Bar_ops;
typedef struct Bar {
  Bar_ops ops;
  void* data;
} Bar;
#define Bar_ctor(_this_obj) \
  ((_this_obj)->ops.ctor)((_this_obj))
#define Bar_dtor(_this_obj) \
  ((_this_obj)->ops.dtor)((_this_obj))
```

4.14.7.9.2 Operations

An interface operation in IDL maps to the following:

- A typedef defining an alias <fully-scoped interface name>_<operation name>_t for a function pointer

  The first parameter in the function pointer's signature is named this_obj, and its type is a pointer to the interface struct. The other parameters map from the IDL parameters in the same order, each with the same name as its corresponding IDL parameter and with a type mapped from its corresponding IDL parameter as specified elsewhere. The return type of

the C function pointer is always FACE_interface_return. If the IDL operation has a non-void return type, then the C function pointer maps as if the IDL operation had an additional "out" parameter named retval whose type is the non-void return type.

- A member in the interface operations table struct

  The name of the member is the same as the IDL operation, and its type is the alias defined by the typedef. These members follow the initialization and cleanup function pointer members, and they are declared in the same order as the operations in IDL.

- A #define macro for the operation

  This macro hides the operation table lookup, allowing simpler code in business logic that uses the interface. The #define identifier for the initialization function is <fully-scoped interface name>_<operation name>(<parameter list>), and the replacement is ((_this_obj)->ops.<operation_name>)(<paren_parameter list>), where <parameter list> is a comma-separated list of parameter names from the corresponding function pointer, each prepended with an underscore, and where <paren parameter_list> is the same, but with parentheses around each underscore-prepended parameter name.

These function pointers, the operations table struct, and the main interface struct are effectively analogous to a C++ abstract class. Keeping the C++ analogy, these functions are effectively "pure virtual".

Interface Operations Example

```
// IDL
interface Foo {
  void go();
  long stop(in short x);
};

// C
struct Foo;

typedef FACE_interface_return (*Foo_ctor_t)(struct Foo* this_obj);
typedef FACE_interface_return (*Foo_dtor_t)(struct Foo* this_obj);

// 1. typedef defining alias for function pointer
//    corresponding to operation
typedef FACE_interface_return (*Foo_go_t)(struct Foo* this_obj);
typedef FACE_interface_return
        (*Foo_stop_t)(struct Foo* this_obj, FACE_short x, FACE_long* retval);

typedef struct Foo_ops {
  Foo_ctor_t ctor;
  Foo_dtor_t dtor;

  // 2. operations table struct members corresponding to the operations
  Foo_go_t go;
  Foo_stop_t stop;
} Foo_ops;

typedef struct Foo {
  Foo_ops ops;
  void* data;
} Foo;

#define Foo_ctor(_this_obj) \
```

```
  ((_this_obj)->ops.ctor)((_this_obj))
#define Foo_dtor(_this_obj) \
  ((_this_obj)->ops.dtor)((_this_obj))

// 3. Macro corresponding to the operation
#define Foo_go(_this_obj) \
  ((_this_obj)->ops.go)((_this_obj))
#define Foo_stop(_this_obj, _x, _retval) \
  ((_this_obj)->ops.stop)((_this_obj), (_x), (_retval))
```

A parameter's directionality in IDL affects the parameter's type in C, as summarized in Table 13. The return type of a C function pointer corresponding to an operation is always FACE_interface_return.

Table 13: IDL Operation Parameter C Mapping

Parameter Type	in	inout	out
Basic Type	T	T *	T *
Enumeration	T	T *	T *
Sequence	const T *	T *	T *
String	const T *	T *	T *
Fixed	const T *	T *	T *
Structure	const T *	T *	T *
Union	const T *	T *	T *
Interface	const T *	T *	T **
Array	const T	T	T

The following outlines the ownership and memory management responsibilities of parameter passing based on an IDL parameter's directionality:

- IDL in parameters – the caller is responsible for providing all storage (either dynamically or statically allocated)
- IDL out parameters – the caller is responsible for providing storage (either dynamically or statically allocated) for the top-level type.

  For strings and sequences (whether as parameters themselves or as a component of a compound type), the callee is permitted to re-size or re-allocate the contained buffer, provided the instance of the object in question is managed. As a consequence of this, the caller may choose to simply initialize a string or sequence and rely on the callee to allocate storage for that object.
- IDL inout parameters – the caller is responsible for providing storage (either dynamically or statically allocated) for the top-level type

For strings and sequences (whether as parameters themselves or as a component of a compound type), the callee is permitted to re-size or re-allocate the contained buffer, provided the instance of the object in question is managed. As a consequence of this, the caller may choose to simply initialize a string or sequence and rely on the callee to allocate storage for that object.

4.14.7.9.3 Attributes

Attributes in IDL logically map to an accessor operation for both mutable and readonly attributes, and a mutator operation for mutable attributes. The accessor operation is named get_<attribute name>, takes no parameters, and returns the same type as the attribute. The mutator operation is named set_<attribute name>, takes an *in* parameter with the same type and identifier as the attribute, and returns void. These operations then map according to Section 4.14.7.9.2. Both operations are declared at the same point the attribute was declared, and the set operation immediately follows its corresponding get operation.

Interface Attributes Example

```
// IDL
interface Foo {
  attribute long x;
  readonly attribute string y;
};

// logically equivalent to the following IDL
interface Foo {
  long get_x();
  void set_x(in long x);
  long get_y();
};
```

4.14.7.9.4 Declarations

Types and constants declared in an interface map to C in the same way they would if declared in the same scope as the interface, except their name is also scoped by the interface in which they are declared.

Interface Declaration Example

```
// IDL
interface Foo {
  typedef char MyChar;
};

// C
/* (constructs for Foo specified in earlier example */
typedef FACE_char Foo_MyChar;
Inheritance
```

It is important to note that inheritance in IDL does not imply anything about the implementation of that interface in a particular programming language. A derived interface in IDL is logically equivalent to an interface that contains the attributes and operations of its base interface. The base interface attributes and operations occur before the derived interface operations and in the same order as in the base interface.

Interface Inheritance Example

```
// IDL
interface Base {
  readonly attribute long x;
  void stop();
};
interface Foo : Base { void go(); };

// logically equivalent to the following IDL
interface Foo {
  readonly attribute long x;
  void stop();
  void go();
};
```

In the case of multiple inheritance, base interfaces are considered in the order they are specified, using depth-first traversal if multiple levels of inheritance exist. (Interface derivation order is semantically insignificant in IDL, but it is significant when mapping to C, because operations generate members in a struct, and the order of members in a struct matters in C.)

Interface Multiple Inheritance Example

```
// IDL
interface A { /* (A members) */ };
interface B : A { /* (B members) */ };
interface C : A { /* (C members) */ };
interface D : B,C { /* (D members) */ };
interface E : C,B { /* (E members) */ };

// C (only operations tables are shown)
typedef struct A_ops {
  // (A member mappings)
} A_ops;

typedef struct B_ops {
  // (A member mappings)
  // (B member mappings)
} B_ops;

typedef struct C_ops {
  // (A member mappings)
  // (C member mappings)
} C_ops;

typedef struct D_ops {
  // (A member mappings)
  // (B member mappings) – B ops declared before C ops
  // (C member mappings)
  // (D member mappings)
} D_ops;

typedef struct E_ops {
  // (A member mappings)
  // (C member mappings) - C ops declared before B ops
  // (B member mappings)
  // (D member mappings)
} E_ops;
```

4.14.7.9.5 Implementation

An implementation of an interface is realized by:

- Defining a structure that contains implementation-specific private data
- Declaring and defining functions that match the operations in the interface

An interface implementation is responsible for initializing the private data in the "ctor" function, cleaning up the private data in the "dtor" function, and setting the primary interface struct's "data" member appropriately.

Interface Implementation Example

```
// IDL
interface Foo {
  void go();
};

// C (Foo interface declaration)
struct Foo;
typedef FACE_interface_return (*Foo_ctor_t)(struct Foo* this_obj);
typedef FACE_interface_return (*Foo_dtor_t)(struct Foo* this_obj);
typedef FACE_interface_return (*Foo_go_t)(struct Foo* this_obj);
typedef struct Foo_ops {
  Foo_ctor_t ctor;
  Foo_dtor_t dtor;
  Foo_go_t go;
} Foo_ops;
typedef struct Foo {
  Foo_ops ops;
  void* data;
} Foo;

#define Foo_ctor(_this_obj) \
  ((_this_obj)->ops.ctor)((_this_obj))
#define Foo_dtor(_this_obj) \
  ((_this_obj)->ops.dtor)((_this_obj))
#define Foo_go(_this_obj) \
  ((_this_obj)->ops.go)((_this_obj))

// C (Foo interface implementation)
// 1. structure defining implementation-specific private data
typedef struct Foo_impl_private_data{
   // members to hold private data...
} Foo_impl_private_data;

// 2. function declarations matching operations in interface
//    (function definitions are implementation-specific)
FACE_interface_return MyFoo_ctor(struct Foo* this_obj); // initialize this_obj-
>data
FACE_interface_return MyFoo_dtor(struct Foo* this_obj); // clean up this_obj-
>data
FACE_interface_return MyFoo_go(struct Foo* this_obj);
```

The example below illustrates the use of the "Foo" interface. A user "instantiates" the implementation "MyFoo" by defining a struct "Foo_Instance" whose first member is the operations table and whose second member is NULL. The address of the "instance" is then passed as the this_obj parameter to the interface's functions. The Foo_ctor function will replace the second member of "Foo_Instance" with a pointer to its private data.

Interface Implementation Use Example

```
// C (use of Foo interface)
Foo Foo_Instance = {
  {  MyFoo_ctor, MyFoo_dtor, MyFoo_go },
  NULL
};

int main(int argc, char* argv[])
{
  FACE_interface_return rc;
  rc = Foo_ctor(&Foo_Instance);
  Foo_go(&Foo_Instance);
  rc = Foo_dtor(&Foo_Instance);

  return 0;
}
```

4.14.7.10 Native Types

There is no general mapping for a native type. Each native type declaration is accompanied by a mapping to C which covers all instances in which the native type might be used, including but not limited to type definitions and operation parameters (in, inout, out).

The following native types are used in FACE Standardized Interfaces:

- FACE::SYSTEM_ADDRESS_TYPE:

  — type: void*

  — in parameter: SYSTEM_ADDRESS_TYPE (pass by value)

  — inout parameter: SYSTEM_ADDRESS_TYPE* (pass by pointer)

  — out parameter: SYSTEM_ADDRESS_TYPE* (pass by pointer)

  — return: SYSTEM_ADDRESS_TYPE (by value)

4.14.8 IDL to C++ Mapping

This section describes a mapping from IDL 4.1 to C++ 2003 (ISO/IEC 14882:2003: Programming Languages – C++). It is based on mappings defined in OMG IDL C++ Language Mapping, Version 1.3. Although this mapping is strictly to C++ 2003, this mapping is also based on general mapping patterns and strategies presented in the OMG IDL C++11 Language Mapping, Version 1.2.

4.14.8.1 Names

Names in IDL generally obey C++ naming conventions, with exceptions described below. Unless otherwise excepted below, unscoped names in IDL appear in the generated source code character-for-character.

In the event that a name that is legal in IDL conflicts with a reserved C++ keyword, the name for that symbol is constructed by prefixing it with “FACE_”. Any language in the remainder of this section indicating that a C++ construct has the “same name” as its IDL counterpart takes this conflict resolution into account.

4.14.8.2 File Names

An IDL source file maps to a C++ header file with the same base name and an ".hpp" extension. The header file is in a subdirectory tree which reflects the subdirectory tree for the IDL source file. For example, if the input IDL file is Foo/Bar/Sample.idl, then the generated C++ header file will be Foo/Bar/Sample.hpp.

Note: A subdirectory tree for an input IDL source file and an output C++ header file is relative to a base directory that is not specified. The base directory for the source IDL file may be different from the base directory for the output C++ header file.

4.14.8.3 Preprocessor Directives

A #ifndef include guard is always present in a C++ header file generated from IDL. The identifier used in the guard is in capital letters. It consists of a leading underscore, the IDL subdirectory tree with the components capitalized, separated by an underscore, followed by the capitalized base name of the resulting C++ header file, and ending with "_HPP". For example, Foo/Bar/Sample.idl will have an include guard of _FOO_BAR_SAMPLE_HPP.

Other IDL preprocessor directives do not map to anything in the C++ representation.

4.14.8.4 Modules

Modules map to C++ namespaces.

Scope resolution operators, used when resolving definitions from other modules, map to the same in C++.

Modules Example

```
// IDL
module A {
  typedef long MyLong;
  module B {
    typedef MyLong MyOtherLong;
  };
};

module C {
  typedef A::MyLong MyOtherLong;
};

// C++
namespace A {
  typedef FACE::Long MyLong;
  namespace B {
    typedef MyLong MyOtherLong;
  }
}

namespace C {
  typedef A::MyLong MyOtherLong;
}
```

4.14.8.5 *Typedefs*

An IDL typedef creates an alias for a type; it maps directly to a C++ typedef. The C++ typedef's alias is the name of the IDL typedef; the C++ typedef type's mapping is specified in the IDL type's relevant section below. Multiple declarators in IDL map to the same in C++.

Typedef Example 1

```
// IDL
typedef long Foo, Bar;
module A {
  typedef Bar Baz;
};

// C++
typedef FACE::Long Foo, Bar;
namespace A {
  typedef Bar Baz;
}
```

Structures, unions, and enumerations can be declared within a typedef in IDL, which is logically equivalent to a type declaration immediately followed by a typedef.

Typedef Example 2

```
// IDL
typedef struct Foo_struct { long x; } Foo;

// (logically equivalent to the following IDL)
struct Foo_struct { long x; };
typedef Foo_struct Foo;
```

4.14.8.6 *Constants*

A constant maps to a #define in C++. The #define identifier is the fully scoped name of the IDL constant, as specified in Section 4.14.7.1.1. If the IDL constant's value expression is a scoped name, then the #define replacement is mapped from that scoped name as specified in Section 4.14.7.1. Otherwise, the #define replacement is a C++ expression mapped from the IDL constant expression as specified in Section 4.14.4. In all cases, the #define replacement includes a cast to the fully-qualified type mapped from the IDL constant's type as specified elsewhere.

Constants Example

```
// IDL
typedef long MyLong;
enum Color {RED, GREEN, BLUE};
module A {
  const long FooLong = 1 + 65535;
  const boolean FooBool = TRUE;
  const MyLong FooMyLong = FooLong;
  const Color clr = RED;
};

// C++
typedef FACE::Long MyLong;
struct Color {
  enum Value {RED, GREEN, BLUE};
private:
```

```
  Color ();
};
#define A_FooLong ((FACE::Long) 65536)
#define A_FooBool ((FACE::Boolean) true)
#define A_FooMyLong ((MyLong)A_FooLong)
#define A_clr ((Color::Value)Color::RED)
```

4.14.8.7 Simple Types

4.14.8.7.1 Basic Types

IDL Basic Types map to C++ types according to Table 14 below. Implementations provide definitions for these C++ types that align with the given size and range requirements. The file containing these definitions is "FACE/types.hpp", specified in Section K.2.1.

Table 14: IDL Basic Type C++ Mapping

IDL Basic Type	C++ Type	Size (bytes)	Range of Values	Default Value
short	FACE::Short	2	-2^15 to (2^15 - 1)	0
long	FACE::Long	4	-2^31 to (2^31 - 1)	0
long long	FACE::LongLong	8	-2^63 to (2^63 - 1)	0
unsigned short	FACE::UnsignedShort	2	0 to (2^16 - 1)	0
unsigned long	FACE::UnsignedLong	4	0 to (2^32 - 1)	0
unsigned long long	FACE::UnsignedLongLong	8	0 to (2^64 - 1)	0
float	FACE::Float	4	IEEE 754-2008 single precision floating point	0.0
double	FACE::Double	8	IEEE 754-2008 double precision floating point	0.0
long double	FACE::LongDouble	10	IEEE 754-2008 extended double precision floating point	0.0
char	FACE::Char	1	-2^7 to (2^7 - 1)	0
boolean	FACE::Boolean	1	true or false	false
octet	FACE::Octet	1	0 to (2^8 - 1)	0

4.14.8.7.2 Sequences

Bounded and unbounded sequences map to a typedef FACE::Sequence specialization with the appropriate element type and a #define in C++, where the #define identifier is the fully-scoped name of the sequence (as specified in Section 4.14.7.1.1) appended with "_bound_value", and the #define's replacement is the sequence's maximum size cast to a FACE::UnsignedLong. To

indicate that a sequence is unbounded, the sentinel value FACE::Sequence<T>::UNBOUNDED_SENTINEL is used as the replacement value. Full specification for FACE::Sequence is in Section K.2.2.

Note: The bound of a sequence is an instance parameter of FACE::Sequence. Implementations are responsible for instantiating a FACE::Sequence with the appropriate maximum size.

Sequence Example

```
// IDL
typedef short TYPE;
typedef sequence<TYPE> Foo;
typedef sequence<TYPE,8> Bar;

// C++
typedef FACE::Short TYPE;
typedef FACE::Sequence<TYPE> Foo;
#define Foo_bound_value FACE::Sequence<TYPE>::UNBOUNDED_SENTINEL
typedef FACE::Sequence<TYPE> Bar;
#define Bar_bound_value ((FACE::UnsignedLong) 8)
```

4.14.8.7.3 Strings

Bounded and unbounded strings map to a typedef FACE::String and a #define in C++, where the #define identifier is the fully-scoped name of the string (as specified in Section 4.14.7.1.1) appended with "_bound", and the #define's replacement is the string's maximum size cast to a FACE::UnsignedLong. To indicate that a string is unbounded, the sentinel value FACE::String::UNBOUNDED_SENTINEL is used as the replacement value. Full specification for FACE::String is in Section K.2.3.

Note: The bound of a string is an instance parameter of FACE::String. Implementations are responsible for instantiating a FACE::String with the appropriate maximum size.

String Example

```
// IDL
typedef string Foo;
typedef string<8> Bar;

// C++
typedef FACE::String Foo;
#define Foo_bound_value FACE::String::UNBOUNDED_SENTINEL
typedef FACE::String Bar;
#define Bar_bound_value ((FACE::UnsignedLong) 8)
```

4.14.8.7.4 Fixed

A fixed type maps to a typedef FACE::Fixed and two #defines to represent the type's digits and scale. For the digits #define, the identifier is the fully-scoped name of the fixed type (as specified in Section 4.14.7.1.1) appended with "_digits", and the replacement is the type's total number of digits cast to a FACE::UnsignedShort. For the scale #define, the identifier is the fully-scoped name of the fixed type (as specified in Section 4.14.7.1.1) appended with "_scale", and the replacement is the type's scale cast to a FACE::UnsignedShort. Implementations are responsible for initializing a fixed type using these constants. Full specification for FACE::Fixed is in Section K.2.4.

Fixed Type Example

```
// IDL
typedef fixed<5,2> Foo;
// C++
typedef FACE::Fixed Foo;
#define Foo_digits ((FACE::UnsignedShort) 5)
#define Foo_scale ((FACE::UnsignedShort) 2)
```

4.14.8.8 Constructed Types

4.14.8.8.1 Structures

An IDL structure maps to a C++ structure with the same name. The structure's members occur in the same order as in IDL; each member's type and identifier are mapped as specified elsewhere.

A forward declaration of a structure in IDL maps to a forward declaration of a structure with the same name in C++.

Structure Example

```
// IDL
typedef long MyLong;

struct A {
  long X;
  MyLong Y;
  char Z;
};

// C++
typedef FACE::Long MyLong;

struct A {
  FACE::Long X;
  MyLong Y;
  FACE::Char Z;
};
```

4.14.8.8.2 Enumerations

An IDL enumeration maps to a C++ enum whose literals are specified with the same name and in the same order as in IDL. The C++ enum is named Value and is wrapped in a struct with the same name as the IDL enum. The struct provides an encapsulating scope in order to reduce the chance for enumerator collisions in scopes with many enumeration declarations, as may occur with TSS message types contained in a USM. The struct has a private constructor declaration with no definition to prohibit construction.

Enumeration Example

```
// IDL
enum Color {RED, GREEN, BLUE};

// C++
struct Color {
  enum Value {RED, GREEN, BLUE};
private:
  Color ();
```

```
};
```

4.14.8.8.3 Unions

An IDL union maps to a C++ class of the same name. Implementations are responsible for supplying the definition of this class, and are permitted to define class members beyond what is specified here. The class contains the following:

- Default constructor – initializes all members to their appropriate default value
- Copy constructor – performs a deep copy
- Assignment operator – performs a deep copy
- Destructor – releases all members

The class contains a public enum RETURN_CODE with two literals in the following order:

- NO_ERROR – indicating no error has occurred
- INVALID_STATE – indicating an operation cannot occur given the union's current state

Each IDL union member maps to public mutator and accessor methods that have the same name as the union member. For a member of type T, where T is mapped from the IDL union member's type as specified elsewhere:

- The first mutator returns void and has a single parameter whose type is T for basic types and enumerations and const T& for all other types
- The second mutator returns T& and takes no parameters
- The accessor is const, returns RETURN_CODE, and has a single parameter whose type is T&

Calling a mutator sets the value of the union member, possibly releasing storage associated with the member's old value, and sets the discriminator to the appropriate value. If multiple cases exist for the same member, the discriminator is set to the value of the first case listed for that member.

Calling a member accessor sets its parameter to the member's value and returns NO_ERROR if the discriminator is set appropriately; otherwise, the parameter is not modified and INVALID_STATE is returned.

The IDL union discriminator maps to a public accessor method, and also maps to a mutator method if any union member has multiple cases. Both methods are named discriminator. For a discriminator of type T, where T is mapped from the IDL discriminator's type as specified elsewhere:

- The mutator (if needed) returns RETURN_CODE and has a single parameter whose type is T
- The accessor is const, takes no parameters, and returns T

The discriminator mutator may only be used to change the discriminator to a different value for the same union member, in which case NO_ERROR is returned; otherwise, the discriminator is not modified and INVALID_STATE is returned.

The discriminator accessor returns the current value of the discriminator. The value returned from the accessor immediately after the class is constructed depends on the union's default case:

- Default case explicitly defined – return explicit default case
- No default case defined – return first case

Note: C++ unions are not used in this mapping, because C++ unions cannot contain certain types as members – specifically, those types mapped to from IDL strings, sequences, and fixed types.

Union Example

```
// IDL
enum CASES { FOO, BAR, BAZ };

union FooUnion switch (CASES) {
  case FOO: short a;
  case BAR: long b;
  // NOTE: IDL does not require a case for every enum literal
};

// C++
struct CASES {
  enum Value {FOO, BAR, BAZ};
private:
  CASES ();
};

class FooUnion {
public:
  enum RETURN_CODE {
    NO_ERROR,
    INVALID_STATE
  };

  FooUnion();
  FooUnion(const FooUnion&);
  FooUnion& operator=(const FooUnion&);
  ~FooUnion();

  // no discriminator mutator defined
  CASES::Value discriminator() const;

  void a(FACE::Short);
  FACE::Short& a();
  RETURN_CODE a(FACE::Short&) const;

  void b(FACE::Long);
  FACE::Long& b();
  RETURN_CODE b(FACE::Long&) const;

  // implementation-specific members
};
```

4.14.8.9 Arrays

An array in IDL maps to a typedef C-style array of the same name and dimension. The array type's mapping is specified in the IDL type's relevant section.

Array Example

```
// IDL
typedef short Foo[10];

typedef short Bar[4][5];

// C++
typedef FACE::Short Foo[10];

typedef FACE::Short Bar[4][5];
```

4.14.8.10 Interfaces

4.14.8.10.1 Declaration

An IDL interface definition is mapped to an abstract class with the same name in C++. The class has a protected default constructor with empty inline definition, a private copy constructor with no definition, a private assignment operator with no definition, and a public pure virtual destructor with empty inline definition. An interface may also be declared with a forward declaration, in which case it maps to a forward declaration of a class in C++.

Interface Declaration Example

```
// IDL
interface Bar;
 interface Foo {};
 interface Bar {};

// C++
 class Bar;

class Foo {
protected:
  Foo() {}
private:
  Foo(const Foo&);
  Foo& operator=(const Foo&);
 public:
  virtual ~Foo() {};= 0;
};

class Bar {
protected:

Bar() {}
private:
  Bar(const Bar&);
  Bar& operator=(const Bar&);
public:
  virtual ~Bar() {};= 0;
};
```

4.14.8.10.2 Operations

An interface operation in IDL maps to a public pure virtual member function declaration with the same name in C++. The member function's parameters map from the IDL parameters in the same order, each with the same name as its corresponding IDL parameter and with a type mapped from its corresponding IDL parameter as specified elsewhere. The return type of the C++ member function is always void. If the IDL operation has a non-void return type, then the

C++ member function maps as if the IDL operation had an additional “out” parameter named retval whose type is the non-void return type.

Interface Operations Example

```
// IDL
interface Foo {
 void go();
 long stop(in short x);
};

// C++
class Foo {
protected:
  Foo() {}
private:
  Foo(const Foo&);
  Foo& operator=(const Foo&);
public:
  virtual ~Foo() {};
  virtual void go() = 0;
  virtual void stop(FACE::Short x, FACE::Long& retval) = 0;
};
```

A parameter’s directionality in IDL affects the parameter’s type in C++. An *in* parameter of type T is passed as T for basic types and const T& for all other types. An *inout* or *out* parameter of type T is passed as T&. An *inout* or *out* parameter representing a FACE Interface of type I is passed as I**. The return type of a C++ member function corresponding to an operation is always void.

The following outlines the ownership and memory management responsibilities of parameter passing based on an IDL parameter’s directionality:

- IDL in parameters – the caller is responsible for providing all storage (either dynamically or statically allocated)
- IDL out parameters – the caller is responsible for providing storage (either dynamically or statically allocated) for the top-level type

  For strings and sequences (whether as parameters themselves or as a component of a compound type), the callee is permitted to re-size or re-allocate the contained buffer, provided the instance of the object in question is managed. As a consequence of this, the caller may choose to simply initialize a string or sequence and rely on the callee to allocate storage for that object.
- IDL inout parameters – the caller is responsible for providing storage (either dynamically or statically allocated) for the top-level type

  For strings and sequences (whether as parameters themselves or as a component of a compound type), the callee is permitted to re-size or re-allocate the contained buffer, provided the instance of the object in question is managed. As a consequence of this, the caller may choose to simply initialize a string or sequence and rely on the callee to allocate storage for that object.

4.14.8.10.3 Attributes

Attributes in an IDL interface map to mutator and accessor methods. The methods are public, pure virtual, and have the same name as their respective IDL attribute. For an attribute of type T, where T is mapped from the IDL attribute's type as specified elsewhere:

- The first mutator returns void and has a single parameter whose type is T for basic types and const T& for all other types
- The second mutator returns T& and takes no parameters
- The accessor is const, takes no parameters, and returns T for basic types and const T& for all other types

Readonly attributes map only to the accessor method.

Interface Attributes Example

```
// IDL
interface Foo {
  attribute long x;
  readonly attribute string y;
};

// C++
class Foo {
protected:
  Foo() {}
private:
  Foo(const Foo&);
  Foo& operator=(const Foo&);
public:
  virtual ~Foo() {};

  virtual void x(FACE::Long) = 0;
  virtual FACE::Long& x() = 0;
  virtual FACE::Long x() const = 0;

  virtual const FACE::String& y() const = 0;
};
```

4.14.8.10.4 Declarations

Types declared in an IDL interface map public typedef declarations in the scope of the class the interface maps to. The typedef maps to C++ according to Section 4.14.8.5.

Interface Declaration Example

```
// IDL
interface Foo {
  typedef char MyChar;
};

// C++
class Foo {
protected:
  Foo() {}
private:
  Foo(const Foo&);
  Foo& operator=(const Foo&);
```

```
public:
  typedef FACE::Char MyChar;
  virtual ~Foo() {};
};
```

4.14.8.10.5 Inheritance

A derived interface maps to a C++ class that inherits (publicly) from its base interfaces' classes.

Interface Inheritance Example

```
// IDL
interface Base {
  readonly attribute long x;
  void stop();
};
interface Foo : Base { void go(); };

// C++
class Base { /* (contents as specified elsewhere) */ };
class Foo : public Base { /* (contents as specified elsewhere */ };
```

In the case of multiple inheritance, base interfaces are considered in the order they are specified, using depth-first traversal if multiple levels of inheritance exist. (Interface derivation order is semantically insignificant in IDL, but it is significant when mapping to C++, because it dictates construction and cleanup ordering.)

Interface Multiple Inheritance Example

```
// IDL
interface A { /* (A members) */ };
interface B : A { /* (B members) */ };
interface C : A { /* (C members) */ };
interface D : B,C { /* (D members) */ };
interface E : C,B { /* (E members) */ };

// C++
class A { /* (contents as specified elsewhere) */ };
class B : public A { /* (contents as specified elsewhere) */ };
class C : public A { /* (contents as specified elsewhere) */ };
class D : public B,C { /* (contents as specified elsewhere) */ };
class E : public C,B { /* (contents as specified elsewhere) */ };
```

4.14.8.10.6 Implementation

An implementation of an interface is realized in C++ by defining and implementing a concrete class that inherits from the abstract interface class.

Interface Implementation Example

```
// IDL
interface Foo {
  void go();
};

// C++ (Foo interface declaration)
class Foo {
protected:
  Foo() {}
private:
```

```
  Foo(const Foo&);
  Foo& operator=(const Foo&);
public:
  virtual ~Foo() {};
  virtual void go() = 0;
};

// C++ (Foo interface implementation)
class MyFooImpl : public Foo {
public:
  ~MyFooImpl(); // implementation elsewhere
  void go(); // implementation elsewhere

// other members as needed by the implementation
```

4.14.8.11 };Native Types

There is no general mapping for a native type. Each native type declaration is accompanied by a mapping to C++ which covers all instances in which the native type might be used, including but not limited to type definitions and operation parameters (in, inout, out).

The following native types are used in FACE Standardized Interfaces:

- FACE::SYSTEM_ADDRESS_TYPE:

  — type: void*

  — in parameter: SYSTEM_ADDRESS_TYPE (pass by value)

  — inout parameter: SYSTEM_ADDRESS_TYPE& (pass by reference)

  — out parameter: SYSTEM_ADDRESS_TYPE& (pass by reference)

  — return: SYSTEM_ADDRESS_TYPE (by value)

4.14.9 IDL to Ada Mapping

This section describes a mapping from IDL 4.1 to Ada. It is based on mappings defined in OMG IDL Language Mapping for Ada, Version 1.3.

This section may be modified in subsequent releases. Prior to implementing, please ensure you are using the latest revision of FACE Technical Standard, Edition 3.x, and you have checked to see if any minor releases, corrigenda, or approved corrections have been published.

This language mapping does not support IDL in which a parent module depends on a child module or two sibling modules are co-dependent.

4.14.9.1 Names

4.14.9.1.1 Identifiers

An identifier in IDL maps to the same identifier in Ada, with the following exceptions (applied in order):

- Leading underscores are prepended with “U”
- Identifiers ending in an underscore are appended with “U”

- Multiple consecutive underscores are all replaced with “U”, except the first one
- Identifiers that conflict with a reserved Ada keyword are prepended with “FACE_”

Any language in the remainder of this section indicating that an Ada construct has the “same name” as its IDL counterpart takes into account these exceptions.

Table 15: Identifier Mapping Example

IDL Identifier	Ada Identifier
_T	U_T
T_	T_U
T___T	T_UUT
package	FACE_package

Using leading, trailing, or consecutive underscores in an IDL identifier may cause a conflict when mapping to Ada. An IDL compiler emits an error if such a conflict occurs.

Identifier Conflict Example

```
// IDL
typedef long U_T;
typedef long __T; /* Legal IDL, but would cause conflict in Ada.
                     Emit error. */
```

4.14.9.1.2 Scoped Names

Scopes in IDL map to the Ada declarative regions as follows:

Table 16: IDL Scope Ada Mapping

IDL Scope	Ada Declarative Region
Global	Library package named by the source IDL file’s name appended with “_IDL_FILE”.
Module or Interface declared in global scope	Library package.
Module or Interface declared in non-global-scope	Child package.

4.14.9.1.3 File Names

There is no associative mapping of an IDL source file into an Ada source file. Each Ada package created by the IDL to Ada mapping rules defined throughout Section 4.14.9 is subject to the following file naming convention:

- Each Ada package specification is to be contained within a single file

The name of that file is determined by the name of the package which the file contains. The name is formed by taking the fully qualified name of the package and replacing the separating dots with hyphens and using lowercase for all letters.

- An exception arises if the file name generated by the above rules starts with one of the characters a, g, i, or s, and the second character is a hyphen

  In this case, the character tilde (~) is used in place of the hyphen. The reason for this special rule is to avoid clashes with the standard names for child units of the system packages commonly provided with the compiler.

- The file extension for all Ada specification files created by the IDL mapping is ".ads"

Table 17 shows some examples of these rules.

Table 17: Example Ada Package File Names

Ada Package (Compilation Unit)	Source File
FACE (specification)	face.ads
Arith_Functions (package specification)	arith_functions.ads
Func.Spec (child package specification)	func-spec.ads

Note: A subdirectory tree for an input IDL source file and output Ada specification file(s) is relative to a base directory that is not specified. The base directory for the source IDL file may be different from the base directory for the output Ada specification file(s).

4.14.9.2 Preprocessor Directives

IDL preprocessor directives do not map to anything in Ada.

4.14.9.3 Modules

A module maps to an Ada package with the same name.

A module declared in a global scope maps to a library package; a nested module (one declared inside another module) maps to a child package of the package mapped from its enclosing module. Similarly, an interface declared in a module maps to a child package of the package mapped from its enclosing module (see Section 4.14.9.9).

Declarations scoped by an IDL module, including those in a "reopened" module, map to declarations in the corresponding Ada package. Thus, a module and all of its subsequent reopenings map to a single package. The declarations in the resulting package may be mapped either directly or indirectly from the corresponding declaration in the source module. Intuitively an indirect mapping goes through a sequence of subtype declarations or constant renaming declarations to obtain an effect equivalent to a direct mapping (types and constants are the only kinds of entities, other than interfaces and nested modules, that can be declared in a module). More formally:

An IDL typedef declaration

```
typedef <old-type> <new-type>;
```

is mapped either directly according to Section 4.14.9.4 or indirectly to a sequence of *n* Ada subtype declarations:

```
subtype T1 is <old-type>;
subtype T2 is T1;
subtype Tn-1 is Tn-2;
subtype <new-type> is Tn-1;
```

The last subtype declaration in the sequence appears in the package mapped by the module; the other subtype declarations appear in auxiliary packages that the IDL-to-Ada compiler may construct.

The subtypes <old-type>, T_1, … T_{n-1}, and <new-type> are said to be *equivalent* since the effect of the application program is independent of which one is referenced.

An IDL enumeration declaration:

```
enum <enum_type> { <enumerator> [, <enumerator>] }
```

is mapped either directly according to Section 4.14.9.7.3 or indirectly as follows:

```
subtype <enum_type> is <package_name>.<enum_type>;
```

where <package_name> is the "with'd" Ada package that declares the base <enum_type>.

In this case each enumeration literal is mapped to a corresponding parameterless Ada function in the "with'ing" package:

```
function <enumerator> return <package_name>.<enum_type>
    renames <package_name>.<enumerator>;
```

An IDL const declaration:

```
const <type> <new-constant> = <const_expr>;
```

is mapped either directly according to Section 4.14.9.5 or indirectly to a sequence of *n* Ada declarations:

```
C1 : constant <ada_type> := <ada_const_expr>;
C2   :  <ada_type> renames C1;
<new-constant> : <ada_type> renames Cn-1;
```

where <ada_type> is equivalent to <type>, and <ada_const_expr> is a mapping of the IDL <const_expr>.

The last declaration in the sequence appears in the package mapped by the module; the other declarations appear in auxiliary packages that the IDL-to-Ada compiler may construct. The expression <ada_const_expr> and the constants C_1, C_2, … <new-constant> are said to be *equivalent* since they all evaluate to the same result. In the mapping of <const_expr> to <ada_const_expr> each name is mapped to the same name or an equivalent name.

Notes

A nested module can refer to entities declared earlier in its enclosing scope, and entities declared in a nested module may be referenced by subsequent declarations in the enclosing scope. The effect of these declarations must be preserved in the parent package and child package to which the modules map. The implementation can generate an auxiliary package structure (not visible to

application code), coupled with indirect mappings to declarations in these packages, to achieve this effect.

Modules may have mutual interdependencies induced by reopenings. Each such module needs to map to an Ada package; the declarations in the module are mapped to declarations in the package. The Ada packages do not have to reflect the same interdependencies as the original modules (and when modules are mutually interdependent their relationships cannot be preserved). As above, the implementation can generate an auxiliary package structure, coupled with indirect mappings, to achieve a correct effect.

To minimize the likelihood of namespace conflicts, an implementation should construct any auxiliary packages in a child package hierarchy rooted at an empty package IDL_Modules.

An indirect mapping comprises a sequence of subtype or constant renaming declarations. The same subsequence may appear in more than one indirect mapping, in which case all of the subtypes (respectively, constants) in all of these mappings are equivalent.

Module Example 1: Direct Mapping

```
// IDL
module A {
  typedef short Foo;
};

module B {
  typedef A.Foo Bar;
};

—- Ada
with FACE;

package A is
  subtype Foo is FACE.Short;
end A;

with A;

package B is
  subtype Bar is A.Foo;
end B;
```

Module Example 2: Indirect Mapping (Continuation of Example 1)

```
// IDLmodule A {  // reopening   const B.Bar k=10;}
-- Ada
with A1, B1, A2;

 -- Auxiliary packages for original A and B and reopened
A package A is
  subtype Foo is A1.Foo;
  K : B1.Bar renames A2.K;
end A;

with B1;
package B is
  subtype Bar is B1.Bar;
end B;
```

The names of the auxiliary packages are not defined by this Technical Standard and are transparent to FACE application code.

4.14.9.4 *Typedefs*

An IDL typedef creates an alias for a type; it maps to an Ada subtype whose name is the same as the IDL alias and whose type is specified in the IDL type's relevant section below. Multiple declarators in IDL are logically equivalent to multiple IDL typedefs.

Typedef Example 1

```
// IDL
module A {
  typedef long Foo, Bar;
  typedef Foo Baz;
};

-- Ada
package A is
  subtype Foo is FACE.Long;
  subtype Bar is FACE.Long;
  subtype Baz is Foo;
end A;
```

Structures, unions, and enumerations can be declared within a typedef in IDL, which is logically equivalent to a type declaration immediately followed by a typedef.

Typedef Example 2

```
// IDL
typedef struct Foo_struct { long x; } Foo;

// (logically equivalent to the following IDL)
struct Foo_struct { long x; };
typedef Foo_struct Foo;
```

4.14.9.5 *Constants*

A constant in IDL maps to a constant of the same name in Ada. The type of the Ada constant is mapped as specified elsewhere. The value of the Ada constant is mapped from the IDL constant expression as specified in Section 4.14.4.

If the IDL expression is an enumeration literal, the Ada constant's value is the appropriate enumeration literal in Ada.

Constants Example

```
// IDL
module A {
  typedef long MyLong;
  enum Color {RED, GREEN, BLUE};
  const long FooLong = 1 + 65535;
  const boolean FooBool = TRUE;
  const MyLong FooMyLong = FooLong;
  const Color clr = RED;
};

-- Ada
```

```
package A is
  subtype MyLong is FACE.Long;
  type Color is (RED, GREEN, BLUE);

  FooLong : constant FACE.Long := 65536;
  FooBool : constant FACE.Boolean := True;
  FooMyLong : constant MyLong := FooLong;
  clr : constant Color := RED;
end A;
```

4.14.9.6 Simple Types

4.14.9.6.1 Basic Types

IDL Basic Types map to Ada (sub)types according to Table 18, for which implementations provide the given definitions.

Table 18: IDL Basic Type Ada Mapping

IDL Basic Type	Ada Type	Subtype/ Derived	Ada Base Type
short	FACE.Short	Subtype	Interfaces.Integer_16
long	FACE.Long	Subtype	Interfaces.Integer_32
long long	FACE.Long_Long	Subtype	Interfaces.Integer_64
unsigned short	FACE.Unsigned_Short	Subtype	Interfaces.Unsigned_16
unsigned long	FACE.Unsigned_Long	Subtype	Interfaces.Unsigned_32
unsigned long long	FACE.Unsigned_Long_Long	Subtype	Interfaces.Unsigned_64
float	FACE.Float	Subtype	Interfaces.IEEE_Float_32
double	FACE.Double	Subtype	Interfaces.IEEE_Float_64
long double	FACE.Long_Double	Subtype	Interfaces.IEEE_Extended_Float
char	FACE.Char	Subtype	Standard.Character
octet	FACE.Octet	Subtype	Interfaces.Unsigned_8
boolean	FACE.Boolean	Subtype	Standard.Boolean

4.14.9.6.2 Sequences

The first use of a bounded sequence in a scope maps to an instantiation (in the same scope) of the generic FACE.Sequences.Bounded package named "FACE_Bounded_Sequence_<type>", where <type> is the name of the sequence's element type. The actual for the formal parameter "Element" is mapped from the sequence's type as specified elsewhere. All references to the bounded sequence within the same scope map to the "Sequence" type defined in this instantiation.

An unbounded sequence maps in the same way, except using an instantiation of FACE.Sequences.Unbounded named FACE_Unbounded_Sequence_<type>.

When constructing a sequence package instantiation's name, if the IDL sequence's element type is a basic type, <type> is the IDL typename; otherwise, <type> is the typename as mapped to Ada.

Full specification for these packages is in Section K.3.1.1, Section K.3.1.2, and Section K.3.1.3.

Sequence Example

```
// IDL
module A {
  typedef sequence< short, 8> Foo;
  typedef sequence< Foo > Bar;
};

-- Ada
with FACE.Sequences.Bounded;
with FACE.Sequences.Unbounded;
package A is

  -- first occurrence of bounded sequence of short
  package FACE_Bounded_Sequence_short is
    new FACE.Sequences.Bounded
    (Element => FACE.Short );

  type Foo is new FACE_Bounded_Sequence_short.Sequence( 8 );

  -- first occurrence of unbounded sequence of Foo
  package FACE_Unbounded_Sequence_Foo is
    new FACE.Sequences.Unbounded
    (Element => Foo );

  type Bar is new FACE_Unbounded_Sequence_Foo.Sequence;
end A;
```

4.14.9.6.3 Strings

An IDL unbounded string maps to Ada as if it were an IDL unbounded sequence of characters. An IDL bounded string maps to Ada as if it were an IDL bounded sequence of characters with the same bound.

String Example

```
// IDL
module A {
  typedef string Foo;
};

// logically equivalent to the following IDL
module A {
  typedef sequence<char> Foo;
};
```

4.14.9.6.4 Fixed

The first use of an IDL fixed type in a scope maps to a definition (in the same scope) of a decimal fixed-point type named Fixed_<digits>_<scale>, where <digits> and <scale> are the

digits and scale of the fixed type as specified in IDL. The decimal fixed-point type definition has the same number of digits as the IDL fixed type, and a delta value equal to 10-<scale>. All references to the fixed type within the same scope map to subtypes of this type definition.

Fixed Example

```
// IDL
module A {
  typedef fixed<5,2> Foo;
};
-- Ada
package A is
  type Fixed_5_2 is delta 0.01 digits 5;
  subtype Foo is Fixed_5_2;
end A;
```

4.14.9.7 Constructed Types

4.14.9.7.1 Structures

An IDL structured type maps to an Ada record. Each structured type member maps to a record component with the same name, in the same order. The type of each component is mapped as specified elsewhere.

Structured Type Example

```
// IDL
module A {
  struct Foo {
    long X;
    char Y;
  };
};

-- Ada
package A is
  type Foo is record
    X : FACE.Long;
    Y : FACE.Char;
  end record;
end A;
```

4.14.9.7.2 Unions

An IDL union maps to an Ada discriminated (variant) record with the same name. The record's discriminant is named "Switch", its type is mapped from the IDL union's discriminator type as specified elsewhere, and its default value is the first value of its type. Each union member maps to a variant in the discriminated record, as specified in Section 4.14.9.7.1. The discrete choice list for each variant is mapped from the constant expression(s) of the corresponding IDL case label, as specified in Section 4.14.4, with expressions "or"ed together if a member has multiple case labels. The discrete choice for the "default" case is "others". If there is no "default" case in the IDL, the Ada variant record declaration includes a default choice "others => null".

Union Example 1

```
// IDL
module A {
  enum Direction {UP, LEFT, RIGHT, DOWN};
```

```
  union UnionExample switch (Direction) {
    case UP: long myUp;
    case LEFT:
    case RIGHT: short myWay;
    default: boolean myDefault;
  };
};

-- Ada
package A is
  type Direction is (UP, LEFT, RIGHT, DOWN);
  type UnionExample (Switch : A.Direction := A.Direction'First) is record
    case Switch is
      when UP =>
        myUp : FACE.Long;
      when LEFT | RIGHT =>
        myWay : FACE.Short;
      when others =>
        myDefault : FACE.Boolean;
    end case;
  end record;
end A;
```

Union Example 2

```
// IDL (No default specified)
module B {
  enum Direction {UP, LEFT, RIGHT, DOWN};
  union UnionExample switch (Direction) {
    case UP: long myUp;
    case LEFT:  short otherWay;
    case RIGHT: short myWay;
      };
};

-- Ada
package B is
  type Direction is (UP, LEFT, RIGHT, DOWN);
  type UnionExample (Switch : Direction := Direction'First) is record
    case Switch is
      when UP =>
        myUp : FACE.Long;
      when LEFT =>
        otherWay : FACE.Short;
      when RIGHT =>
        myWay : FACE.Short;
      when others =>
        null;
    end case;
  end record;
end B;
```

4.14.9.7.3 Enumerations

An IDL enumeration maps to an Ada enumerated type with the same name and with literals specified with the same name and in the same order as in IDL.

Enumeration Example

```
// IDL
module A {
```

```
  enum Color {RED, GREEN, BLUE};
};

-- Ada
package A is
  type Color is (RED, GREEN, BLUE);
end A;
```

4.14.9.8 Arrays

The first use of an IDL array in a scope maps to a definition (in the same scope) of an array type. The type's name is Array_<dimensionality>_<type> where <type> is the name of the sequence's type, and <dimensionality> is the underscore-separated sequence of dimensions as specified in IDL. The array's element type is mapped as specified elsewhere; each of its dimensions is over the range from 0 to 1 less than its corresponding dimension in IDL. All references to the array within the same scope map to subtypes of this type definition.

When constructing the array type definition's name, if the IDL array's element type is a basic type, <type> is the IDL typename; otherwise, <type> is the typename as mapped to Ada.

Array Example

```
// IDL
module A {
  typedef short Foo[10];
  typedef short Bar[4][5];
};

-- Ada
package A is
  type Array_10_short is array(0 .. 9) of FACE.Short;
 subtype Foo is Array_10_short;
  type Array_4_5_short is array(0 .. 3, 0 .. 4) of FACE.Short;
  subtype Bar is Array_4_5_short;
end A;
```

4.14.9.9 Interfaces

4.14.9.9.1 Declaration

An IDL interface maps to a package in Ada (an "interface package") with the same name. If the IDL interface is declared inside a module, the interface package is a child package of the package mapped from the module; otherwise the interface package is a root library package.

The interface package contains either an abstract tagged null record type (Ada 95) or an interface type (Ada 2012) named Interface_T (the "interface type") and a class-wide general access type for Interface_T named Interface_T_Ptr. Any reference to the interface (e.g., as an operation parameter or structure member) maps to this type Interface_T (or Interface_T'Class in some cases).

Interface Declaration Example

```
// IDL
interface Foo {};

-- Ada 95
package Foo is
```

```
  type Interface_T is abstract tagged null record;
  type Interface_T_Ptr is access all Interface_T'Class;
end Foo;

-- Ada 2012
package Foo is
   type Interface_T is interface;
   type Interface_T_Ptr is access all Interface T'Class;
end Foo;
```

An IDL interface declared with a forward declaration maps to a package with the same name, appended with "_Forward". The package contains either an abstract tagged null record type (Ada 95) or an interface type (Ada 2012) named Interface_T (the "interface forward type"). Any reference to the forward declared interface before its full declaration maps to this type Interface_T (or Interface_T'Class in some cases).

If an interface is forward declared, the interface package for the full declaration also contains a nested package named "Convert" for converting between the interface type and the interface forward type. This package contains two abstract functions – the first is named From_Forward, takes a single *in* parameter of the interface forward type named "Forward", and returns the interface type; the second is named To_Forward, takes a single *in* parameter of the interface type named "Full", and returns the interface forward type.

Interface Forward Declaration Example

```
// IDL
interface Foo;
 interface Foo {};

-- Ada 95
package Foo_Forward is
  type Interface_T is abstract tagged null record;
 end Foo_Forward;

package Foo is
  type Interface_T is abstract tagged null record;
  type Interface_T_Ptr is access all Interface_T'Class;
  package Convert is
    function From_Forward (Forward : in Foo_Forward.Interface_T)
      return Foo.Interface_T is abstract;
    function Interface_To_Forward (Full : in Foo.Interface_T)
      return Foo_Forward.Interface_T is abstract;
  end Convert;
 end Foo;

-- Ada 2012
package Foo_Forward is
  type Interface_T is interface;
end Foo_Forward;

package Foo is
  type Interface_T is interface;
  type Interface_T_Ptr is access all Interface_T'Class;
  package Convert is
    function From_Forward (Forward : in Foo_Forward.Interface_T)
      return Foo.Interface_T is abstract;
    function Interface_To_Forward (Full : in Foo.Interface_T)
      return Foo_Forward.Interface_T is abstract;
  end Convert;
end Foo;
```

4.14.9.9.2 Operations

An interface operation in IDL maps to an abstract primitive subprogram of the interface type with the same name as the operation. The first parameter of the subprogram is an *access* parameter named "Self" of the interface type.

Each parameter in the operation maps to a parameter in the subprogram with the same name and mode, in the same order, following the "Self" parameter.

If the operation has a non-void return type and only *in* parameters, it maps to an abstract function whose return type is mapped from the operation's return type. Otherwise, the operation maps to an abstract procedure. If the operation has a non-void return type, but maps to an abstract procedure, then the return type maps to an *out* parameter named "Returns" that follows all other parameters.

Except for the "Self" parameter, the type of each subprogram parameter or return type is mapped from the operation parameter's type or return type, respectively, except when that type is the interface itself, in which case it maps to the class of the interface type (i.e., Interface_T'Class). (This prevents multiple controlling parameters, making it clear that the "Self" parameter controls dispatching.)

Interface Operations Example

```
// IDL
interface Foo {
  void op1();
  short op2();
  short op3(out long A);
};

-- Ada 95
package Foo is
  type Interface_T is abstract tagged null record;
  type Interface_T_Ptr is access all Interface_T'Class;
  procedure op1 (Self : access Interface_T) is abstract;
  function  op2 (Self : access Interface_T)
    return FACE.Short is abstract;
  procedure op3 (Self   : access Interface_T;
A      : out FACE.Long;
Result : out FACE.Short) is abstract;
end Foo;

-- Ada 2012
package Foo is
  type Interface_T is interface;
  type Interface_T_Ptr is access all Interface_T'Class;
  procedure op1 (Self : access Interface_T) is abstract;
  function  op2 (Self : access Interface_T)
    return FACE.Short is abstract;
  procedure op3 (Self   : access Interface_T;
A      : out FACE.Long;
Result : out FACE.Short) is abstract;
end Foo;
```

4.14.9.9.3 Attributes

Attributes in IDL logically map to an accessor operation, for both mutable and readonly attributes, and a mutator operation for mutable attributes. The accessor operation is named

Get_<attribute name>, takes no parameters, and returns the same type as the attribute. The mutator operation is named Set_<attribute name>, takes an *in* parameter with the same type and identifier as the attribute, and returns void. These operations then map according to Section 4.14.9.9.2.

Interface Attributes Example

```
// IDL
interface Foo {
  attribute long x;
  readonly attribute string y;
};

// logically equivalent to the following IDL
interface Foo {
  long Get_x();
  void Set_x(in long x);
  string Get_y();
};
```

4.14.9.9.4 Inheritance

In the IDL mapping to Ada 95, the interface type of a derived interface is derived from the interface type of the first listed base interface. (The operations and attributes of that base interface are then inherited through tagged type inheritance.) The operations and attributes of all other direct and indirect base interfaces map explicitly to primitive subprograms of the derived interface's interface type.

In the IDL mapping to Ada 2012, the interface type of a derived interface is derived from each of the base interfaces listed in the interface_inheritance_spec. The operations and attributes of the direct and indirect base interfaces are then inherited through interface inheritance and are implicitly mapped to corresponding primitive subprograms of the derived interface's interface type.

Interface Inheritance Example

```
// IDL
interface Base1 { void stop(); };
interface Base2 { void slow(); };
interface Foo : Base1, Base2 { void go(); };

-- Ada 95
-- mapping for Base1 and Base2 as specified elsewhere
package Foo is
  type Interface_T is abstract new Base1.Interface_T with null record;
  type Interface_T_Ptr is access all Interface_T'Class;
  procedure slow(Self : access Interface_T) is abstract;
  procedure go(Self : access Interface_T) is abstract;
end Foo;

-- Ada 2012
-- mapping for Base1 and Base2 as specified elsewhere
package Foo is
  type Interface_T is new Base1.Interface_T and Base2.Interface_T;
  type Interface_T_Ptr is access all Interface_T'Class;
  -- stop, slow inherited
  procedure go(Self : access Interface_T) is abstract;
end Foo;
```

4.14.9.9.5 Implementation

An implementation of an interface is realized in Ada by defining and implementing a package containing a type that derives from the interface's interface type.

Interface Implementation Example

```
// IDL
interface Foo {
 void go();

};

-- Ada 95 (interface declaration)
package Foo is
  type Interface_T is abstract tagged null record;
  type Interface_T_Ptr is access all Interface_T'Class;
  procedure go(Self : access Interface_T) is abstract;
end Foo;

-- Ada 2012 (interface declaration)
package Foo is
  type Interface_T is interface;
  type Interface_T_Ptr is access all Interface_T'Class;
  procedure go(Self : access Interface_T) is abstract;
end Foo;

-- Ada 95 and Ada 2012 (interface implementation)
with foo;
package MyFooImpl is
  type Interface_T is new foo.Interface_T with private;
  procedure go(Self : access Interface_T);
private
  type Interface_T is new foo.Interface_T with record
    null; -- or any record extension needed by implementation
  end record;
 end MyFooImpl;
```

4.14.9.10 Native Types

There is no general mapping for a native type. Each native type declaration is accompanied by a mapping to Ada which covers all instances in which the native type might be used, including but not limited to type definitions and operation parameters (in, inout, out).

The following native types are used in FACE Standardized Interfaces:

- FACE::SYSTEM_ADDRESS_TYPE

  In all cases, this type maps to a System.Address in Ada.

4.14.10 IDL to Java Mapping

This section defines a mapping from a subset of IDL 4.1 to Java. It is based on mappings defined in OMG IDL to Java Language Mapping, Version 1.3. Table 19 is a summary of the mappings elaborated upon in this section.

This section may be modified in subsequent releases. Prior to implementing, please ensure you are using the latest revision of FACE Technical Standard, Edition 3.x, and you have checked to see if any minor releases, corrigenda, or approved corrections have been published.

Table 19: Summary of IDL to Java Mapping

IDL Construct	Java Construct
Source file	No mapping
Preprocessor directive	No mapping
Module	Package
Interface	Interface
Operation	Method (Declared in the Interface)
Attribute	Accessor and Mutator Methods in Interface
Inheritance	Extending Interfaces
Data types	Classes
Constants	Encapsulated by Interfaces
Exception	No mapping

4.14.10.1 Names

Names in IDL generally obey Java naming conventions, with exceptions described below. Unless otherwise excepted below, unscoped names in IDL appear in the generated source code character-for-character.

In addition, cases exist where a name that is legal in IDL may conflict with the corresponding Java source. When this occurs, the name for that symbol is constructed by prefixing it with "FACE_". The following is a list of potential naming conflicts:

- Keywords in the Java language (e.g., abstract, boolean, if, etc.)
- Java literals (e.g., true, false, null)
- IDL declarations that collide with methods on java.lang.Object (e.g., clone, equals, hashCode, etc.)
- IDL declarations that collide with class, interface, and method names as defined by this section

Any language in the remainder of this section indicating that a Java construct has the "same name" as its IDL counterpart takes this conflict resolution into account.

4.14.10.2 File Names

There is no associative mapping of an IDL source file into a Java source file. Each Java package and public class created by the IDL to Java mapping rules defined throughout Section 4.14.10 is subject to the following file naming conventions:

- The source for each public class, interface, enumeration, or annotation type is to be defined in a text file whose name is the simple name of the type and whose extension is ".java"
- The file extension for all Java class files created by the IDL mapping is ".java"
- The fully qualified name of the package member and the relative path name to the file are to be parallel; for example, package org.example.MyPackage is in file org/example/MyPackage.java

Note: A subdirectory tree for an input IDL source file and output Java source file(s) is relative to a base directory that is not specified. The base directory for the source IDL file may be different from the base directory for the output Java source file(s).

4.14.10.3 Preprocessor Directives

IDL preprocessor directives do not map to anything in Java.

4.14.10.4 Modules

Modules map to Java packages with the same name, lower-cased. Scope resolution operators, used when resolving definitions from other modules, map to the same package in Java and are replaced by a Java scope resolution operator.

Modules Example

```
// IDL
module A {
  module B {
    struct Foo{ long i;};
  };
};

// Java
package a.b;
public interface Foo {
  int geti();
  void seti(int i);
}
```

4.14.10.5 Typedefs

An IDL typedef does not have an equivalent in Java. An IDL typedef aliases a type. When a typedef alias is used, it is equivalent to using the original type.

Typedef Equivalence Example

```
// IDL
typedef long Foo;
typedef Foo AnotherFoo;
struct Foo_struct{ AnotherFoo x; };
typedef Foo_struct AnotherFoo_struct;
struct Bar_struct {
  Foo myFoo;
  AnotherFoo myAnotherFoo;
  AnotherFoo_struct myAFStruct;
};
```

```
// Equivalent IDL
struct Foo_struct{long x;};
struct Bar_struct{
  long myFoo;
  long myAnotherFoo;
  Foo_struct myAFStruct;
};
```

4.14.10.6 Constants

Constants map differently depending on scope. In all cases, the value of the Java constant is mapped from the IDL constant expression as specified in Section 4.14.4, and its type is mapped as specified elsewhere. If a constant appears within an interface, the IDL constant is mapped to a final qualified member of the same name in the resulting Java interface.

Constant Example 1

```
// IDL
module Example {
  interface FooInterface {
    const long aFooConstant = 32;
  };
};

// Java
package Example;
public interface FooInterface{
  final int aFooConstant = 32;
}
```

If a constant is declared outside of an interface scope, it maps to a Java interface with the same name as the constant with a final qualified member named “value” for assigning the constant’s value.

Constants Example 2

```
// IDL
module Example {
  const long aBarConstant = -42;
};

// Java
package Example;
public interface aBarConstant {
  final int value = -42;
}
```

4.14.10.7 Simple Types

4.14.10.7.1 Basic Types

Table 20 shows the mapping of IDL basic types to Java. Care should be taken when using unsigned types in the IDL as Java does not support usage of unsigned types.

Table 20: IDL Basic Type Java Mapping

IDL Basic Type	Java Data Type
short	short
long	int
long long	long
unsigned short	int
unsigned long	long
unsigned long long	java.math.BigInteger
float	float
double	double
long double	java.math.BigDecimal
char	char
octet	byte
boolean	boolean

4.14.10.7.2 Sequences

An IDL sequence maps to a java.util.List for both bounded and unbounded cases. Bounds are enforced by the implementation of a method that uses the type, as specified in Section 4.14.10.10.5.

All values specifying the length of a sequence get removed when the IDL maps to Java. If the value of the sequence length is desired to be represented in the Java representation, then the value is represented as a constant value in IDL.

Sequence Example

```
// IDL
struct Foo_struct{ long i;};
typedef Foo_struct AnotherFoo_struct;
struct Bar_struct {
  long myLong;
  sequence<Foo_struct, 32> myFoo;
  sequence<AnotherFoo_struct> myAnotherFoo;
};

// Java
public interface Foo_struct {
  int geti();
  void seti(int i);
}

public interface Bar_struct {
```

```
  int getmyLong();
  void setmyLong(int myLong);

  java.util.List<Foo_struct> getmyFoo();
  void setmyFoo(java.util.List<Foo_struct> myFoo);

  java.util.List<Foo_struct> getmyAnotherFoo();
  void setmyAnotherFoo(java.util.List<Foo_struct> myAnotherFoo);
}
```

4.14.10.7.3 Strings

IDL strings map to a Java String for both bounded and unbounded cases. Bounds are enforced by the implementation of a method that uses the type, as specified in Section 4.14.10.10.5.

Any values restricting the length of an IDL string are removed when mapping to Java. If a value restricting the length of a string is desired to be represented in the Java representation, then the value should be represented as a constant value in IDL.

4.14.10.7.4 Fixed

IDL fixed types map to java.math.BigDecimal.

Any value restrictions are removed when IDL maps to Java and no limiting of the fixed type is done in Java. If a value restricting the IDL fixed type is desired to be represented in the Java representation, then the value should be represented as a constant value in IDL.

4.14.10.8 Constructed Types

4.14.10.8.1 Structured Types

An IDL struct maps to a Java interface with the same name that contains mutator and accessor methods for each IDL member. The accessor method for an IDL member is named “get” followed by the name of the IDL member, takes no parameters, and has a return type mapped from the IDL member’s type as specified elsewhere. The mutator method for an IDL member is named “set” followed by the name of the IDL member, returns void, and has a single parameter whose name is the same as the IDL member and whose type is mapped from the IDL member’s type as specified elsewhere. An IDL struct defined in an IDL interface maps to a nested Java interface.

Structure Example

```
// IDL
typedef long long Foo;
typedef Foo AnotherFoo;
struct Foo_struct { AnotherFoo x; };
typedef Foo_struct AnotherFoo_struct;
struct Bar_struct {
  Foo myFoo;
  AnotherFoo myAnotherFoo;
  AnotherFoo_struct myAFstruct;
};

// Java
public interface Foo_struct {
  long getx();
  void setx(long x);
}
```

```
public interface Bar_struct {
  long getmyFoo();
  void setmyFoo(long myFoo);

  long getmyAnotherFoo();
  void setmyAnotherFoo(long myAnotherFoo);

  Foo_struct getmyAFStruct();
  void setmyAFStruct(Foo_struct myAFStruct);
}
```

4.14.10.8.2 Unions

An IDL union is mapped to a Java interface with the same name that contains the following:

- An accessor method for the discriminator that is named "getDiscriminator", takes no parameters, and has a return type mapped from the IDL discriminator's type as specified elsewhere
- An accessor method for each member that is named "get" followed by the name of the IDL member, takes no parameters, and has a return type mapped from the IDL member's type as specified elsewhere
- A mutator method for each member that is named "set" followed by the name of the IDL member, returns void, and has a single parameter whose name is the same as the IDL member and whose type is mapped from the IDL member's type as specified elsewhere

If the IDL union has multiple cases for the same union member, then a mutator is also defined for the discriminator. The mutator is named "setDiscriminator", returns void, and has a single parameter named "Discriminator" whose type is mapped from the IDL discriminator's type as specified elsewhere. The mutator may only change the discriminator to a different value for the same union member; any other use results in UnsupportedOperationException being raised.

The value returned from the discriminator's accessor immediately after the class is constructed depends on the union's default case:

- Default case explicitly defined – return explicit default case
- No default case defined – return first case

Calling a member mutator sets the value of the member and sets the discriminator to the appropriate value. If multiple cases exist for the same member, the discriminator is set to the first case listed for that member.

Calling a member accessor returns the member's value if the discriminator is set appropriately; any other use results in UnsupportedOperationException being raised.

If the IDL union is defined within an interface, then the resulting Java interface maps to a nested Java interface.

Union Example

```
// IDL
enum Direction{up,down,left,right};
union UnionExample switch (Direction) {
  case up: long myUp;
```

```
  case left:
  case right: short myWay;
  default: boolean myDefault;
};

// Java
public enum Direction {
  up, down, left, right
}

public interface UnionExample {
  Direction getDiscriminator();

  int getmyUp();
  void setmyUp(int val);

  short getmyWay();
  void setmyWay(short val);

  boolean getmyDefault();
  void setmyDefault(boolean myDefault);
}
```

4.14.10.8.3 Enumerations

An IDL enum is mapped to a public enum with the same name in Java.

If the IDL enum is defined within an interface, then the resulting Java class resides in the corresponding Java interface.

Enumeration Example

```
// IDL
module Example{
  enum Direction{up, down, left, right};
};

// Java
package Example;
public enum Direction {
  up, down, left, right
}
```

4.14.10.8.4 Constructed Recursive Types and Forward Declarations

Forward declarations do not map to Java.

4.14.10.9 Arrays

An IDL array is mapped to a Java array. Bounds are enforced by the implementation of a method that uses the type, as specified in Section 4.14.10.10.5.

Multi-dimensional IDL arrays map to multi-dimensional Java arrays.

Interface Declaration Example

```
// IDL
module Example{
  typedef short Foo[10];
  typedef short Bar[4][5];
```

```
  struct X {
    Foo foo;
    Bar bar;
  };
};

// Java
package example;
public interface X {
  short[] getfoo();
  void setfoo(short[] foo);

  short[][] getbar();
  void setbar(short[][] bar);
}
```

4.14.10.10 Interfaces

4.14.10.10.1 Declaration

An IDL interface definition maps to an interface with the same name in Java.

An IDL interface may also be declared with a forward declaration. This syntax does not map to Java.

Interface Declaration Example

```
// IDL
interface Foo {};

// Java
public interface Foo {}
```

4.14.10.10.2 Operations

An interface operation in IDL maps to a public member method declaration with the same name in Java.

An operation parameter's directionality in IDL affects the parameter's type in Java. An IDL *in* parameter of type T is passed as a T for all types. An IDL *inout* or *out* parameter of type T whose corresponding Java type is mutable is passed as a T. An IDL *inout* or *out* parameter of type T whose corresponding Java type is immutable is passed using the us.opengroup.FACE.Holder class (specified in Section K.4.1) parameterized with the corresponding Java type. Primitive wrapper classes are used to parameterize the Holder class when the corresponding Java type is a Java primitive.

The object parameter passing mechanism in Java requires care on both the caller and callee to abide by the expectation implied in IDL. The following outlines the ownership responsibilities of object passing based on an IDL parameter's directionality:

- IDL *in* parameters – Java objects passed as IDL in parameters are created and owned by the caller

  The callee does not modify or retain a reference to this object beyond the duration of the call. Violation of these rules can result in unpredictable behavior.

- IDL *out* and *return* parameters – Java objects returned as IDL *out* parameters are created and owned by the callee

The callee does not modify or retain a reference to this object beyond the duration of the call. Violation of these rules can result in unpredictable behavior.

- IDL *inout* parameters – Java objects passed as IDL *inout* parameters follow the guidelines for *in* parameters for the *in* value, and follow the guidelines for *out* parameters for the *out* value

Interface Operations Example

```
// IDL
interface Foo{
  void go(in long arg1, inout short arg2);
};

// Java
public interface Foo{
  void go (
    /*in*/    int arg1,
    /*inout*/ us.opengroup.FACE.Holder<Short> arg2
  );
}
```

4.14.10.10.3 Attributes

Attributes in an IDL interface map to a pair of methods for each attribute: one mutator method and one accessor method. The exceptional case is readonly attributes map only to an accessor method.

The methods have the same name as the attribute. Mutator methods have a void return type and accept a parameter of the corresponding attribute type. Accessor methods take no parameters and have a return type matching the attribute type.

Interface Attributes Example

```
// IDL
interface Foo{
  attribute long x;
  readonly attribute long y;
};

// Java
public interface Foo{
  int getx();
  void setx(int x);

  int gety();
}
```

4.14.10.10.4 Inheritance

An IDL derived interface maps to a Java interface that extends from the base interface. In the case of multiple inheritance, the resulting Java interface extends multiple base interfaces.

Interface Inheritance Example

```
// IDL
interface Base { void stop(); };
interface Foo : Base { void go(); };
```

```
// Java
public interface Base {
  void stop();
}

public interface Foo extends Base{
  void go();
}
```

4.14.10.10.5 Implementation

An implementation of an interface is realized in Java by defining and implementing a concrete class that implements the resulting interface.

Interface Implementation Example

```
// IDL
interface Foo {
  void go();
};

// Java
public interface Foo {
  void go();
}

// Implementation of Foo interface
public class FooImpl implements Foo {
  public void go() {/*some implementation here*/}
}
```

IDL definitions and accompanying documentation are used to specify language-agnostic structural and behavioral requirements for interface operations. Some IDL requirements are implied by a parameter's type but cannot be enforced by the Java language. For example, an IDL unsigned short maps to a Java short, which has a wider range of values than dictated by IDL. Because the Java language does not enforce these requirements, interface implementations are responsible for enforcing them at run-time.

The us.opengroup.FACE.BAD_PARAM exception is provided for use in the following cases:

- The length of a Java String falls outside the range specified in the IDL for the string
- The length of a Java List falls outside the range specified in the IDL
- The value of a Java short, int, or long is negative where the IDL specifies an unsigned value
- The value of a java.math.BigDecimal is outside the range specified in the IDL for a long double or fixed type

The us.opengroup.FACE.DATA_CONVERSION exception is provided for use in the following cases:

- A Java character (16-bit Unicode) cannot be represented per the IDL definition of a character (8-bit)

- A character (16-bit Unicode) in a Java String cannot be represented per the IDL definition of a character (8-bit)

The full specification of BAD_PARAM is in Section K.4.2. The full specification of DATA_CONVERSION is in Section K.4.3.

4.14.10.11 Native Types

There is no general mapping for a native type. Each native type declaration is accompanied by a mapping to Java which covers all instances in which the native type might be used, including but not limited to type definitions and operation parameters (in, inout, out).

The following native types are used in FACE Standardized Interfaces:

- FACE::SYSTEM_ADDRESS_TYPE

  In all cases, this type maps to a java.nio.ByteBuffer in Java. Note that every use of FACE::SYSTEM_ADDRESS_TYPE in a FACE Standardized Interface is accompanied by a length. This length is meaningless in Java; the length of the data can obtained using ByteBuffer methods.

5 Security

Military and civilian airborne systems with security considerations are engineered to meet rigorous security standards specified by the authorizing official and evaluation authority. The FACE Technical Standard defines the Security Profile to support security-relevant UoCs. While the Security Profile is targeted for UoCs that actively enforce or contribute to security requirements, other OSS Profiles may work together with the Security Profile to provide protection against specific threats and vulnerabilities. The FACE Reference Architecture allows for evaluation of UoCs providing secure capabilities. The following sections provide basic security considerations expected of a UoC.

5.1 Scope

Security in the context of the FACE Technical Standard:

- Recognizes that procuring agencies and their designated approval authorities define the requirements for addressing systems security and processes to achieve Authority to Operate
- Provides context for processes that capture best principles to be used in developing "high-assurance" security software
- Specifies Security Profile API sets to facilitate security assessment
- Supports partitioning to address isolation of security functions
- Is agnostic to security processes (e.g., DIARMF, NIST RMF)
- Does not assert a UoC is automatically assessed to some evaluation assurance level
- Recognizes UoCs are a part of the platform security architecture and are evaluated with the larger system

5.2 Guiding Concepts

The sections below provide guidance and recommendations on developing FACE security-relevant software components. FACE security-relevant software executes within or supports the Security Profile. It differs from other OSS Profiles in that it also contributes or influences the security policy of the system designed to the FACE Reference Architecture. More specifically, a UoC that executes within the Security Profile but has no mechanism to alter or otherwise impact the security policy is not considered security-relevant. The following sections include discussions on isolation of security functions, design constraints, and assessment considerations. These guiding concepts are critical to reducing the time and cost to field secure solutions to the warfighter and are applicable to the entire software life-cycle and platform stakeholders.

5.2.1 Isolation of Security Functions

The concept of isolation of security-relevant functions from non-security-relevant functions provides a more robust and threat-resilient solution. Additional benefits include modular, less complex designs that can minimize the recurring cost and schedule impacts of assessment and authorization (A&A) required for Authority to Operate. This can help expedite the fielding of new capabilities by leveraging similarity and visibility of information architecture and documentation.

The Security Profile enables isolation by:

- Allowing separate ARINC and/or POSIX partitions that can securely separate relevant functionality into unique time and space execution environments
- Aligning with the FACE Architectural Segments and key interfaces which constrain APIs used for functional operations
- Securely allowing interoperability with other profiles that do not contain security-relevant functions

5.2.2 Security Transformations

Transformations are one type of a security control to protect the confidentiality and integrity of data in transit and at rest within a secure system. Examples include authentication, encryption, and labeling. A Security Transformation may also be used to protect Critical Program Information (CPI). This approach supports isolation of security-enforcing functions which may ease a security evaluation by limiting the elements that would necessitate an assessment. Security Transformations are intended exclusively for security capabilities as designed to meet system-level security controls. Information on the Security Transformation needs to be captured to ensure portability of PCS, PSSS, and TSS UoCs. Given the sensitivity of both the data and transformation mechanism, there may be restrictions on availability and distribution of this information.

Requirements for Security Transformations are included in each applicable segment (Sections 4.6.1.3, 4.7.9, and 4.10.1.4, and use of the TS Interface).

5.2.3 Security Guidance and Design Constraints

The Security Profile also addresses functional and performance attributes that define a reference architecture that can support multiple levels of security processing requirements. To be more specific, security considerations exhibit the following key operational environment attributes for PCS, PSSS, and TSS UoCs:

- Trusted Information Flow between software components
- Trusted Information Flow between software components and external interfaces
- Protection of Data in Processing
- Protection of Data in Transit
- Protection of Data at Rest

It is important to reiterate that the Security Profile is defined for all the FACE segments: OSS, IOSS, PSSS, TSS, and the PCS. For each of the segments, the Security Profile limits the set of APIs to those that are robust and necessary to develop security-related software components.

There are security engineering practices that are important but are outside the domain of the FACE Technical Standard. The specific engineering practices and design constraints vary depending on your platform security requirements but an overview of some accepted security design constraints, general guidance, and examples of security considerations associated with implementing the FACE Technical Standard are included in the FACE Reference Implementation Guide.

6 Safety Considerations

Military and civilian airborne systems with safety considerations are engineered to meet rigorous airworthiness standards specified by the respective airworthiness authorities. The FACE Technical Standard defines Safety Profile to support UoCs with safety implications.

Safety considerations in the context of the FACE Reference Architecture:

- Recognize procuring agencies and their designated airworthiness authorities define the requirements for addressing system safety and processes to achieve airworthiness
- Allow application of software safety design assurance principles
- Specify Safety Profile API sets appropriate for use by UoCs
- The Software Supplier is responsible for the determination of when unbounded sequence and string data types are appropriate for their given implementation
- Support partitioning to address separation of concerns
- Are agnostic to safety processes (e.g., DO-178, ARP 4754A, ARP 4761)
- Does not assert that a UoC automatically achieves any particular safety certification

There are safety engineering practices that are important but are outside the domain of the FACE Technical Standard. The specific engineering practices and design constraints vary depending on your platform safety requirements but an overview of some accepted safety design constraints, general guidance, and examples of safety considerations associated with implementing the FACE Technical Standard are included in the FACE Reference Implementation Guide.

A OSS Profile Details

A.1 OSS Profiles for the POSIX Interface

This normative appendix describes the OSS Profiles based on the IEEE Std 1003.1-2008. The OSS and OS Interface are described in Section 4.1 and Section 4.2, respectively. Table 21 shows FACE OS APIs and their associated OSS Profiles. Each API name is a hyperlink to an IEEE Std 1003.1-2008 description of the API.

Explanation for POSIX Call Table Format for the OSS Profile Columns

INCL Included in the Profile indicated at top of column

Blank Excluded from the Profile indicated at top of column

MP Optional based on multi-process support

Note: Blank items may be included in future editions of the FACE Technical Standard.

The Inter-UoC column includes a "YES" for those APIs whose Inter-UoC usages are restricted to Transport Services and I/O Services.

Table 21: FACE OSS Profile APIs

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
tzname		INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
stderr				INCL		POSIX_DEVICE_IO
stdin				INCL		POSIX_DEVICE_IO
stdout				INCL		POSIX_DEVICE_IO
optarg						POSIX_C_LIB_EXT
opterr						POSIX_C_LIB_EXT
optind						POSIX_C_LIB_EXT
optopt						POSIX_C_LIB_EXT
environ			INCL	INCL		POSIX_SINGLE_PROCESS

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
errno	INCL	INCL	INCL	INCL		POSIX_SINGLE_PROCESS
signgam						XSI_C_LANG_SUPPORT
timezone						XSI_C_LANG_SUPPORT
daylight						XSI_C_LANG_SUPPORT
posix_fadvise()						_POSIX_ADVISORY_INFO
posix_fallocate()						_POSIX_ADVISORY_INFO
posix_memalign()						_POSIX_ADVISORY_INFO
aio_cancel()				INCL		POSIX_ASYNCHRONOUS_IO
aio_error()				INCL		POSIX_ASYNCHRONOUS_IO
aio_fsync()				INCL		POSIX_ASYNCHRONOUS_IO
aio_read()				INCL		POSIX_ASYNCHRONOUS_IO
aio_return()				INCL		POSIX_ASYNCHRONOUS_IO
aio_suspend()				INCL		POSIX_ASYNCHRONOUS_IO
aio_write()				INCL		POSIX_ASYNCHRONOUS_IO
lio_listio()				INCL		POSIX_ASYNCHRONOUS_IO
pthread_barrier_destroy()				INCL		POSIX_BARRIERS
pthread_barrier_init()				INCL		POSIX_BARRIERS
pthread_barrier_wait()				INCL		POSIX_BARRIERS
pthread_barrierattr_destroy()				INCL		POSIX_BARRIERS
pthread_barrierattr_init()				INCL		POSIX_BARRIERS
pthread_barrierattr_ getpshared()						POSIX_BARRIERS
pthread_barrierattr_setpshared()						POSIX_BARRIERS
clock_nanosleep()	INCL	INCL	INCL	INCL		POSIX_CLOCK_SELECTION
pthread_condattr_getclock()		INCL	INCL	INCL		POSIX_CLOCK_SELECTION
pthread_condattr_setclock()		INCL	INCL	INCL		POSIX_CLOCK_SELECTION
clock_getcpuclockid()				INCL		_POSIX_CPUTIME

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
fsync()		INCL	INCL	INCL		_POSIX_FSYNC
msync()				INCL		_POSIX_SYNCHRONIZED_IO
mmap()	INCL	INCL	INCL	INCL		POSIX_MAPPED_FILES
munmap()				INCL		POSIX_MAPPED_FILES
mlockall()				INCL		_POSIX_MEMLOCK
munlockall()				INCL		_POSIX_MEMLOCK
mlock()				INCL		_POSIX_MEMLOCK_RANGE
munlock()				INCL		_POSIX_MEMLOCK_RANGE
mprotect()				INCL		POSIX_MEMORY_PROTECTION
mq_close()			INCL	INCL	YES	_POSIX_MESSAGE_PASSING
mq_getattr()		INCL	INCL	INCL	YES	_POSIX_MESSAGE_PASSING
mq_notify()		INCL	INCL	INCL	YES	_POSIX_MESSAGE_PASSING
mq_open()		INCL	INCL	INCL	YES	_POSIX_MESSAGE_PASSING
mq_receive()		INCL	INCL	INCL	YES	_POSIX_MESSAGE_PASSING
mq_send()		INCL	INCL	INCL	YES	_POSIX_MESSAGE_PASSING
mq_setattr()		INCL	INCL	INCL	YES	_POSIX_MESSAGE_PASSING
mq_unlink()			INCL	INCL	YES	_POSIX_MESSAGE_PASSING
mq_timedreceive()		INCL	INCL	INCL	YES	_POSIX_MESSAGE_PASSING
mq_timedsend()		INCL	INCL	INCL	YES	_POSIX_MESSAGE_PASSING
sched_getparam()			INCL	INCL		_POSIX_PRIORITY_SCHEDULING
sched_getscheduler()			INCL	INCL		_POSIX_PRIORITY_SCHEDULING
sched_setparam()			INCL	INCL		_POSIX_PRIORITY_SCHEDULING
sched_setscheduler()			INCL	INCL		_POSIX_PRIORITY_SCHEDULING
posix_spawnattr_ getschedparam()				MP		_POSIX_PRIORITY_SCHEDULING and _POSIX_SPAWN

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
posix_spawnattr_ getschedpolicy()				MP		_POSIX_PRIORITY_SCHEDULING and _POSIX_SPAWN
posix_spawnattr_ setschedparam()				MP		_POSIX_PRIORITY_SCHEDULING and _POSIX_SPAWN
posix_spawnattr_ setschedpolicy()				MP		_POSIX_PRIORITY_SCHEDULING and _POSIX_SPAWN
sched_yield()	INCL	INCL	INCL	INCL		_POSIX_PRIORITY_SCHEDULING
sched_get_priority_max()	INCL	INCL	INCL	INCL		_POSIX_PRIORITY_SCHEDULING and _POSIX_THREAD_PRIORITY_SCHEDULING
sched_get_priority_min()	INCL	INCL	INCL	INCL		_POSIX_PRIORITY_SCHEDULING and _POSIX_THREAD_PRIORITY_SCHEDULING
sched_rr_get_interval()			INCL	INCL		_POSIX_PRIORITY_SCHEDULING and _POSIX_THREAD_PRIORITY_SCHEDULING
sigqueue()	INCL	INCL	INCL	INCL		POSIX_REALTIME_SIGNALS
sigtimedwait()	INCL	INCL	INCL	INCL		POSIX_REALTIME_SIGNALS
sigwaitinfo()	INCL	INCL	INCL	INCL		POSIX_REALTIME_SIGNALS
sem_close()	INCL	INCL	INCL	INCL		POSIX_SEMAPHORES
sem_destroy()			INCL	INCL		POSIX_SEMAPHORES
sem_getvalue()	INCL	INCL	INCL	INCL		POSIX_SEMAPHORES
sem_init()	INCL	INCL	INCL	INCL		POSIX_SEMAPHORES
sem_open()	INCL	INCL	INCL	INCL		POSIX_SEMAPHORES
sem_post()	INCL	INCL	INCL	INCL		POSIX_SEMAPHORES
sem_trywait()	INCL	INCL	INCL	INCL		POSIX_SEMAPHORES
sem_unlink()			INCL	INCL		POSIX_SEMAPHORES
sem_wait()	INCL	INCL	INCL	INCL		POSIX_SEMAPHORES

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
sem_timedwait()	INCL	INCL	INCL	INCL		POSIX_SEMAPHORES
shm_open()	INCL	INCL	INCL	INCL	YES	_POSIX_SHARED_MEMORY_ OBJECTS
shm_unlink()				INCL	YES	_POSIX_SHARED_MEMORY_ OBJECTS
posix_madvise()						_POSIX_ADVISORY_INFO
posix_spawn()			MP	MP		_POSIX_SPAWN
posix_spawn_file_actions_ addclose()				MP		_POSIX_SPAWN
posix_spawn_file_actions_ adddup2()				MP		_POSIX_SPAWN
posix_spawn_file_actions_ addopen()				MP		_POSIX_SPAWN
posix_spawn_file_actions_ destroy()				MP		_POSIX_SPAWN
posix_spawn_file_actions_init()				MP		_POSIX_SPAWN
posix_spawnattr_destroy()			MP	MP		_POSIX_SPAWN
posix_spawnattr_getflags()			MP	MP		_POSIX_SPAWN
posix_spawnattr_getpgroup()				MP		_POSIX_SPAWN
posix_spawnattr_getsigdefault()			MP	MP		_POSIX_SPAWN
posix_spawnattr_getsigmask()			MP	MP		_POSIX_SPAWN
posix_spawnattr_init()			MP	MP		_POSIX_SPAWN
posix_spawnattr_setflags()			MP	MP		_POSIX_SPAWN
posix_spawnattr_setpgroup()				MP		_POSIX_SPAWN
posix_spawnattr_setsigdefault()			MP	MP		_POSIX_SPAWN
posix_spawnattr_setsigmask()			MP	MP		_POSIX_SPAWN
posix_spawnp()				MP		_POSIX_SPAWN
pthread_spin_destroy()						POSIX_SPIN_LOCKS
pthread_spin_init()						POSIX_SPIN_LOCKS
pthread_spin_lock()						_POSIX_SPIN_LOCKS
pthread_spin_trylock()						_POSIX_SPIN_LOCKS

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
pthread_spin_unlock()						_POSIX_SPIN_LOCKS
fdatasync()				INCL		_POSIX_SYNCHRONIZED_IO
pthread_attr_getstacksize()		INCL	INCL	INCL		_POSIX_THREAD_ATTR_ STACKSIZE
pthread_attr_setstacksize()		INCL	INCL	INCL		_POSIX_THREAD_ATTR_ STACKSIZE
pthread_attr_getstack()	INCL	INCL	INCL	INCL		XSI_THREADS_EXT
pthread_attr_setstack()	INCL	INCL	INCL	INCL		XSI_THREADS_EXT
pthread_getcpuclockid()	INCL	INCL	INCL	INCL		_POSIX_THREAD_CPUTIME
pthread_mutex_getprioceiling()				INCL		_POSIX_THREAD_PRIO_PROTE CT
pthread_mutex_setprioceiling()				INCL		_POSIX_THREAD_PRIO_PROTE CT
pthread_mutexattr_ getprioceiling()	INCL	INCL	INCL	INCL		_POSIX_THREAD_PRIO_PROTE CT
pthread_mutexattr_ setprioceiling()	INCL	INCL	INCL	INCL		_POSIX_THREAD_PRIO_PROTE CT
pthread_attr_getinheritsched()	INCL	INCL	INCL	INCL		_POSIX_THREAD_PRIORITY_ SCHEDULING
pthread_attr_getschedpolicy()	INCL	INCL	INCL	INCL		_POSIX_THREAD_PRIORITY_ SCHEDULING
pthread_attr_getscope()	INCL	INCL	INCL	INCL		_POSIX_THREAD_PRIORITY_ SCHEDULING
pthread_attr_setinheritsched()	INCL	INCL	INCL	INCL		_POSIX_THREAD_PRIORITY_ SCHEDULING
pthread_attr_setschedpolicy()	INCL	INCL	INCL	INCL		_POSIX_THREAD_PRIORITY_ SCHEDULING
pthread_attr_setscope()	INCL	INCL	INCL	INCL		_POSIX_THREAD_PRIORITY_ SCHEDULING
pthread_getschedparam()	INCL	INCL	INCL	INCL		_POSIX_THREAD_PRIORITY_ SCHEDULING
pthread_setschedparam()	INCL	INCL	INCL	INCL		_POSIX_THREAD_PRIORITY_ SCHEDULING

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
pthread_setschedprio()	INCL	INCL	INCL	INCL		_POSIX_THREAD_PRIORITY_ SCHEDULING
pthread_condattr_getpshared()				INCL		_POSIX_THREAD_PROCESS_ SHARED
pthread_condattr_setpshared()				INCL		_POSIX_THREAD_PROCESS_ SHARED
pthread_mutexattr_getpshared()				INCL		_POSIX_THREAD_PROCESS_ SHARED
pthread_mutexattr_setpshared()				INCL		_POSIX_THREAD_PROCESS_ SHARED
pthread_mutexattr_ getprotocol()	INCL	INCL	INCL	INCL		_POSIX_THREAD_PRIO_INHERIT or _POSIX_THREAD_PRIO_PROTECT
pthread_mutexattr_setprotocol()	INCL	INCL	INCL	INCL		_POSIX_THREAD_PRIO_INHERIT or _POSIX_THREAD_PRIO_PROTECT
clock_getres()	INCL	INCL	INCL	INCL		POSIX_TIMERS
clock_gettime()	INCL	INCL	INCL	INCL		POSIX_TIMERS
clock_settime()	INCL	INCL	INCL	INCL		POSIX_TIMERS
nanosleep()	INCL	INCL	INCL	INCL		POSIX_TIMERS
timer_create()	INCL	INCL	INCL	INCL		POSIX_TIMERS
timer_delete()			INCL	INCL		POSIX_TIMERS
timer_getoverrun()	INCL	INCL	INCL	INCL		POSIX_TIMERS
timer_gettime()	INCL	INCL	INCL	INCL		POSIX_TIMERS
timer_settime()	INCL	INCL	INCL	INCL		POSIX_TIMERS
posix_trace_attr_destroy()						_POSIX_TRACE
posix_trace_attr_getclockres()						_POSIX_TRACE
posix_trace_attr_ getcreatetime()						_POSIX_TRACE
posix_trace_attr_ getgenversion()						_POSIX_TRACE

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
posix_trace_attr_ getmaxdatasize()						_POSIX_TRACE
posix_trace_attr_ getmaxsystemeventsize()						_POSIX_TRACE
posix_trace_attr_ getmaxusereventsize()						_POSIX_TRACE
posix_trace_attr_getname()						_POSIX_TRACE
posix_trace_attr_ getstreamfullpolicy()						_POSIX_TRACE
posix_trace_attr_ getstreamsize()						_POSIX_TRACE
posix_trace_attr_init()						_POSIX_TRACE
posix_trace_attr_ setmaxdatasize()						_POSIX_TRACE
posix_trace_attr_setname()						_POSIX_TRACE
posix_trace_attr_ setstreamfullpolicy()						_POSIX_TRACE
posix_trace_attr_setstreamsize()						_POSIX_TRACE
posix_trace_clear()						_POSIX_TRACE
posix_trace_create()						_POSIX_TRACE
posix_trace_event()						_POSIX_TRACE
posix_trace_eventid_equal()						_POSIX_TRACE
posix_trace_eventid_get_ name()						_POSIX_TRACE
posix_trace_eventid_open()						_POSIX_TRACE
posix_trace_eventtypelist_ getnext_id()						_POSIX_TRACE
posix_trace_eventtypelist_ rewind()						_POSIX_TRACE
posix_trace_get_attr()						_POSIX_TRACE
posix_trace_get_status()						_POSIX_TRACE
posix_trace_getnext_event()						_POSIX_TRACE
posix_trace_shutdown()						_POSIX_TRACE
posix_trace_start()						_POSIX_TRACE
posix_trace_stop()						_POSIX_TRACE

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
posix_trace_timedgetnext_ event()						_POSIX_TRACE
posix_trace_trygetnext_event()						_POSIX_TRACE
posix_trace_eventset_add()						_POSIX_TRACE and _POSIX_TRACE_EVENT_FILTER
posix_trace_eventset_del()						_POSIX_TRACE and _POSIX_TRACE_EVENT_FILTER
posix_trace_eventset_empty()						_POSIX_TRACE and _POSIX_TRACE_EVENT_FILTER
posix_trace_eventset_fill()						_POSIX_TRACE and _POSIX_TRACE_EVENT_FILTER
posix_trace_eventset_ ismember()						_POSIX_TRACE and _POSIX_TRACE_EVENT_FILTER
posix_trace_get_filter()						_POSIX_TRACE and _POSIX_TRACE_EVENT_FILTER
posix_trace_set_filter()						_POSIX_TRACE and _POSIX_TRACE_EVENT_FILTER
posix_trace_trid_eventid_ open()						_POSIX_TRACE and _POSIX_TRACE_EVENT_FILTER
posix_trace_attr_getinherited()						_POSIX_TRACE and _POSIX_TRACE_INHERIT
posix_trace_attr_setinherited()						_POSIX_TRACE and _POSIX_TRACE_INHERIT
posix_trace_attr_ getlogfullpolicy()						_POSIX_TRACE and _POSIX_TRACE_LOG
posix_trace_attr_getlogsize()						_POSIX_TRACE and _POSIX_TRACE_LOG
posix_trace_attr_ setlogfullpolicy()						_POSIX_TRACE and _POSIX_TRACE_LOG
posix_trace_attr_setlogsize()						_POSIX_TRACE and _POSIX_TRACE_LOG
posix_trace_close()						_POSIX_TRACE and _POSIX_TRACE_LOG
posix_trace_create_withlog()						_POSIX_TRACE and _POSIX_TRACE_LOG

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
posix_trace_flush()						_POSIX_TRACE and _POSIX_TRACE_LOG
posix_trace_open()						_POSIX_TRACE and _POSIX_TRACE_LOG
posix_trace_rewind()						_POSIX_TRACE and _POSIX_TRACE_LOG
posix_mem_offset()						_POSIX_TYPED_MEMORY_ OBJECTS
posix_typed_mem_get_info()						_POSIX_TYPED_MEMORY_ OBJECTS
posix_typed_mem_open()						_POSIX_TYPED_MEMORY_ OBJECTS
crypt()						_XOPEN_CRYPT
encrypt()						_XOPEN_CRYPT
setkey()						_XOPEN_CRYPT
fattach()						_XOPEN_STREAMS
fdetach()						_XOPEN_STREAMS
getmsg()						_XOPEN_STREAMS
getpmsg()						_XOPEN_STREAMS
ioctl()						_XOPEN_STREAMS
isastream()						_XOPEN_STREAMS
putmsg()						_XOPEN_STREAMS
putpmsg()						_XOPEN_STREAMS
posix_devctl()	INCL	INCL	INCL	INCL		IEEE Std 1003.26, device control
getdate_err						XSI_C_LANG_SUPPORT
longjmp()				INCL		POSIX_C_LANG_JUMP
setjmp()				INCL		POSIX_C_LANG_JUMP
acos()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
acosf()				INCL		POSIX_C_LANG_MATH

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
acosh()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
acoshf()				INCL		POSIX_C_LANG_MATH
acoshl()				INCL		POSIX_C_LANG_MATH
acosl()				INCL		POSIX_C_LANG_MATH
asin()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
asinf()				INCL		POSIX_C_LANG_MATH
asinh()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
asinhf()				INCL		POSIX_C_LANG_MATH
asinhl()				INCL		POSIX_C_LANG_MATH
asinl()				INCL		POSIX_C_LANG_MATH
atan()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
atan2()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
atan2f()				INCL		POSIX_C_LANG_MATH
atan2l()				INCL		POSIX_C_LANG_MATH
atanf()				INCL		POSIX_C_LANG_MATH
atanh()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
atanhf()				INCL		POSIX_C_LANG_MATH
atanhl()				INCL		POSIX_C_LANG_MATH
atanl()				INCL		POSIX_C_LANG_MATH
cabs()				INCL		POSIX_C_LANG_MATH
cabsf()				INCL		POSIX_C_LANG_MATH
cabsl()				INCL		POSIX_C_LANG_MATH
cacos()				INCL		POSIX_C_LANG_MATH
cacosf()				INCL		POSIX_C_LANG_MATH
cacosh()				INCL		POSIX_C_LANG_MATH
cacoshf()				INCL		POSIX_C_LANG_MATH

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
cacoshl()				INCL		POSIX_C_LANG_MATH
cacosl()				INCL		POSIX_C_LANG_MATH
carg()				INCL		POSIX_C_LANG_MATH
cargf()				INCL		POSIX_C_LANG_MATH
cargl()				INCL		POSIX_C_LANG_MATH
casin()				INCL		POSIX_C_LANG_MATH
casinf()				INCL		POSIX_C_LANG_MATH
casinh()				INCL		POSIX_C_LANG_MATH
casinhf()				INCL		POSIX_C_LANG_MATH
casinhl()				INCL		POSIX_C_LANG_MATH
casinl()				INCL		POSIX_C_LANG_MATH
catan()				INCL		POSIX_C_LANG_MATH
catanf()				INCL		POSIX_C_LANG_MATH
catanh()				INCL		POSIX_C_LANG_MATH
catanhf()				INCL		POSIX_C_LANG_MATH
catanhl()				INCL		POSIX_C_LANG_MATH
catanl()				INCL		POSIX_C_LANG_MATH
cbrt()				INCL		POSIX_C_LANG_MATH
cbrtf()				INCL		POSIX_C_LANG_MATH
cbrtl()				INCL		POSIX_C_LANG_MATH
ccos()				INCL		POSIX_C_LANG_MATH
ccosf()				INCL		POSIX_C_LANG_MATH
ccosh()				INCL		POSIX_C_LANG_MATH
ccoshf()				INCL		POSIX_C_LANG_MATH
ccoshl()				INCL		POSIX_C_LANG_MATH
ccosl()				INCL		POSIX_C_LANG_MATH

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
ceil()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
ceilf()				INCL		POSIX_C_LANG_MATH
ceill()				INCL		POSIX_C_LANG_MATH
cexp()				INCL		POSIX_C_LANG_MATH
cexpf()				INCL		POSIX_C_LANG_MATH
cexpl()				INCL		POSIX_C_LANG_MATH
cimag()				INCL		POSIX_C_LANG_MATH
cimagf()				INCL		POSIX_C_LANG_MATH
cimagl()				INCL		POSIX_C_LANG_MATH
clog()				INCL		POSIX_C_LANG_MATH
clogf()				INCL		POSIX_C_LANG_MATH
clogl()				INCL		POSIX_C_LANG_MATH
conj()				INCL		POSIX_C_LANG_MATH
conjf()				INCL		POSIX_C_LANG_MATH
conjl()				INCL		POSIX_C_LANG_MATH
copysign()				INCL		POSIX_C_LANG_MATH
copysignf()				INCL		POSIX_C_LANG_MATH
copysignl()				INCL		POSIX_C_LANG_MATH
cos()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
cosf()				INCL		POSIX_C_LANG_MATH
cosh()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
coshf()				INCL		POSIX_C_LANG_MATH
coshl()				INCL		POSIX_C_LANG_MATH
cosl()				INCL		POSIX_C_LANG_MATH
cpow()				INCL		POSIX_C_LANG_MATH
cpowf()				INCL		POSIX_C_LANG_MATH

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
cpowl()				INCL		POSIX_C_LANG_MATH
cproj()				INCL		POSIX_C_LANG_MATH
cprojf()				INCL		POSIX_C_LANG_MATH
cprojl()				INCL		POSIX_C_LANG_MATH
creal()				INCL		POSIX_C_LANG_MATH
crealf()				INCL		POSIX_C_LANG_MATH
creall()				INCL		POSIX_C_LANG_MATH
csin()				INCL		POSIX_C_LANG_MATH
csinf()				INCL		POSIX_C_LANG_MATH
csinh()				INCL		POSIX_C_LANG_MATH
csinhf()				INCL		POSIX_C_LANG_MATH
csinhl()				INCL		POSIX_C_LANG_MATH
csinl()				INCL		POSIX_C_LANG_MATH
csqrt()				INCL		POSIX_C_LANG_MATH
csqrtf()				INCL		POSIX_C_LANG_MATH
csqrtl()				INCL		POSIX_C_LANG_MATH
ctan()				INCL		POSIX_C_LANG_MATH
ctanf()				INCL		POSIX_C_LANG_MATH
ctanh()				INCL		POSIX_C_LANG_MATH
ctanhf()				INCL		POSIX_C_LANG_MATH
ctanhl()				INCL		POSIX_C_LANG_MATH
ctanl()				INCL		POSIX_C_LANG_MATH
erf()				INCL		POSIX_C_LANG_MATH
erfc()				INCL		POSIX_C_LANG_MATH
erfcf()				INCL		POSIX_C_LANG_MATH
erfcl()				INCL		POSIX_C_LANG_MATH

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
erff()				INCL		POSIX_C_LANG_MATH
erfl()				INCL		POSIX_C_LANG_MATH
exp()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
exp2()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
exp2f()				INCL		POSIX_C_LANG_MATH
exp2l()				INCL		POSIX_C_LANG_MATH
expf()				INCL		POSIX_C_LANG_MATH
expl()				INCL		POSIX_C_LANG_MATH
expm1()				INCL		POSIX_C_LANG_MATH
expm1f()				INCL		POSIX_C_LANG_MATH
expm1l()				INCL		POSIX_C_LANG_MATH
fabs()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
fabsf()				INCL		POSIX_C_LANG_MATH
fabsl()				INCL		POSIX_C_LANG_MATH
fdim()				INCL		POSIX_C_LANG_MATH
fdimf()				INCL		POSIX_C_LANG_MATH
fdiml()				INCL		POSIX_C_LANG_MATH
floor()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
floorf()				INCL		POSIX_C_LANG_MATH
floorl()				INCL		POSIX_C_LANG_MATH
fma()				INCL		POSIX_C_LANG_MATH
fmaf()				INCL		POSIX_C_LANG_MATH
fmal()				INCL		POSIX_C_LANG_MATH
fmax()				INCL		POSIX_C_LANG_MATH
fmaxf()				INCL		POSIX_C_LANG_MATH
fmaxl()				INCL		POSIX_C_LANG_MATH

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
fmin()				INCL		POSIX_C_LANG_MATH
fminf()				INCL		POSIX_C_LANG_MATH
fminl()				INCL		POSIX_C_LANG_MATH
fmod()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
fmodf()				INCL		POSIX_C_LANG_MATH
fmodl()				INCL		POSIX_C_LANG_MATH
fpclassify()				INCL		POSIX_C_LANG_MATH
frexp()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
frexpf()				INCL		POSIX_C_LANG_MATH
frexpl()				INCL		POSIX_C_LANG_MATH
hypot()				INCL		POSIX_C_LANG_MATH
hypotf()				INCL		POSIX_C_LANG_MATH
hypotl()				INCL		POSIX_C_LANG_MATH
ilogb()				INCL		POSIX_C_LANG_MATH
ilogbf()				INCL		POSIX_C_LANG_MATH
ilogbl()				INCL		POSIX_C_LANG_MATH
isfinite()				INCL		POSIX_C_LANG_MATH
isgreater()				INCL		POSIX_C_LANG_MATH
isgreaterequal()				INCL		POSIX_C_LANG_MATH
isinf()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
isless()				INCL		POSIX_C_LANG_MATH
islessequal()				INCL		POSIX_C_LANG_MATH
islessgreater()				INCL		POSIX_C_LANG_MATH
isnan()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
isnormal()				INCL		POSIX_C_LANG_MATH
isunordered()				INCL		POSIX_C_LANG_MATH

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
ldexp()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
ldexpf()				INCL		POSIX_C_LANG_MATH
ldexpl()				INCL		POSIX_C_LANG_MATH
lgamma()				INCL		POSIX_C_LANG_MATH
lgammaf()				INCL		POSIX_C_LANG_MATH
lgammal()				INCL		POSIX_C_LANG_MATH
llrint()				INCL		POSIX_C_LANG_MATH
llrintf()				INCL		POSIX_C_LANG_MATH
llrintl()				INCL		POSIX_C_LANG_MATH
llround()				INCL		POSIX_C_LANG_MATH
llroundf()				INCL		POSIX_C_LANG_MATH
llroundl()				INCL		POSIX_C_LANG_MATH
log()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
log10()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
log10f()				INCL		POSIX_C_LANG_MATH
log10l()				INCL		POSIX_C_LANG_MATH
log1p()				INCL		POSIX_C_LANG_MATH
log1pf()				INCL		POSIX_C_LANG_MATH
log1pl()				INCL		POSIX_C_LANG_MATH
log2()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
log2f()				INCL		POSIX_C_LANG_MATH
log2l()				INCL		POSIX_C_LANG_MATH
logb()				INCL		POSIX_C_LANG_MATH
logbf()				INCL		POSIX_C_LANG_MATH
logbl()				INCL		POSIX_C_LANG_MATH
logf()				INCL		POSIX_C_LANG_MATH

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
logl()				INCL		POSIX_C_LANG_MATH
lrint()				INCL		POSIX_C_LANG_MATH
lrintf()				INCL		POSIX_C_LANG_MATH
lrintl()				INCL		POSIX_C_LANG_MATH
lround()				INCL		POSIX_C_LANG_MATH
lroundf()				INCL		POSIX_C_LANG_MATH
lroundl()				INCL		POSIX_C_LANG_MATH
modf()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
modff()				INCL		POSIX_C_LANG_MATH
modfl()				INCL		POSIX_C_LANG_MATH
nan()				INCL		POSIX_C_LANG_MATH
nanf()				INCL		POSIX_C_LANG_MATH
nanl()				INCL		POSIX_C_LANG_MATH
nearbyint()				INCL		POSIX_C_LANG_MATH
nearbyintf()				INCL		POSIX_C_LANG_MATH
nearbyintl()				INCL		POSIX_C_LANG_MATH
nextafter()				INCL		POSIX_C_LANG_MATH
nextafterf()				INCL		POSIX_C_LANG_MATH
nextafterl()				INCL		POSIX_C_LANG_MATH
nexttoward()				INCL		POSIX_C_LANG_MATH
nexttowardf()				INCL		POSIX_C_LANG_MATH
nexttowardl()				INCL		POSIX_C_LANG_MATH
pow()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
powf()				INCL		POSIX_C_LANG_MATH
powl()				INCL		POSIX_C_LANG_MATH
remainder()				INCL		POSIX_C_LANG_MATH

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
remainderf()				INCL		POSIX_C_LANG_MATH
remainderl()				INCL		POSIX_C_LANG_MATH
remquo()				INCL		POSIX_C_LANG_MATH
remquof()				INCL		POSIX_C_LANG_MATH
remquol()				INCL		POSIX_C_LANG_MATH
rint()				INCL		POSIX_C_LANG_MATH
rintf()				INCL		POSIX_C_LANG_MATH
rintl()				INCL		POSIX_C_LANG_MATH
round()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
roundf()				INCL		POSIX_C_LANG_MATH
roundl()				INCL		POSIX_C_LANG_MATH
scalbln()				INCL		POSIX_C_LANG_MATH
scalblnf()				INCL		POSIX_C_LANG_MATH
scalblnl()				INCL		POSIX_C_LANG_MATH
scalbn()				INCL		POSIX_C_LANG_MATH
scalbnf()				INCL		POSIX_C_LANG_MATH
scalbnl()				INCL		POSIX_C_LANG_MATH
signbit()				INCL		POSIX_C_LANG_MATH
sin()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
sinf()				INCL		POSIX_C_LANG_MATH
sinh()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
sinhf()				INCL		POSIX_C_LANG_MATH
sinhl()				INCL		POSIX_C_LANG_MATH
sinl()				INCL		POSIX_C_LANG_MATH
sqrt()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
sqrtf()				INCL		POSIX_C_LANG_MATH

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
sqrtl()				INCL		POSIX_C_LANG_MATH
tan()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
tanf()				INCL		POSIX_C_LANG_MATH
tanh()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
tanhf()				INCL		POSIX_C_LANG_MATH
tanhl()				INCL		POSIX_C_LANG_MATH
tanl()				INCL		POSIX_C_LANG_MATH
tgamma()				INCL		POSIX_C_LANG_MATH
tgammaf()				INCL		POSIX_C_LANG_MATH
tgammal()				INCL		POSIX_C_LANG_MATH
trunc()	INCL	INCL	INCL	INCL		POSIX_C_LANG_MATH
truncf()				INCL		POSIX_C_LANG_MATH
truncl()				INCL		POSIX_C_LANG_MATH
abs()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
atof()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
atoi()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
atol()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
atoll()				INCL		POSIX_C_LANG_SUPPORT
bsearch()			INCL	INCL		POSIX_C_LANG_SUPPORT
calloc()		INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
ctime()						POSIX_C_LANG_SUPPORT
difftime()		INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
div()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
feclearexcept()				INCL		POSIX_C_LANG_SUPPORT
fegetenv()				INCL		POSIX_C_LANG_SUPPORT
fegetexceptflag()				INCL		POSIX_C_LANG_SUPPORT

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
fegetround()				INCL		POSIX_C_LANG_SUPPORT
feholdexcept()				INCL		POSIX_C_LANG_SUPPORT
feraiseexcept()				INCL		POSIX_C_LANG_SUPPORT
fesetenv()				INCL		POSIX_C_LANG_SUPPORT
fesetexceptflag()				INCL		POSIX_C_LANG_SUPPORT
fesetround()				INCL		POSIX_C_LANG_SUPPORT
fetestexcept()				INCL		POSIX_C_LANG_SUPPORT
feupdateenv()				INCL		POSIX_C_LANG_SUPPORT
free()			INCL	INCL		POSIX_C_LANG_SUPPORT
gmtime()				INCL		POSIX_C_LANG_SUPPORT
imaxabs()				INCL		POSIX_C_LANG_SUPPORT
imaxdiv()				INCL		POSIX_C_LANG_SUPPORT
isalnum()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
isalpha()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
isblank()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
iscntrl()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
isdigit()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
isgraph()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
islower()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
isprint()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
ispunct()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
isspace()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
isupper()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
isxdigit()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
labs()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
ldiv()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
llabs()				INCL		POSIX_C_LANG_SUPPORT
lldiv()				INCL		POSIX_C_LANG_SUPPORT
localeconv()				INCL		POSIX_C_LANG_SUPPORT
localtime()				INCL		POSIX_C_LANG_SUPPORT
malloc()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
memchr()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
memcmp()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
memcpy()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
memmove()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
memset()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
mktime()		INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
qsort()			INCL	INCL		POSIX_C_LANG_SUPPORT
rand()				INCL		POSIX_C_LANG_SUPPORT
realloc()			INCL	INCL		POSIX_C_LANG_SUPPORT
setlocale()				INCL		POSIX_C_LANG_SUPPORT
snprintf()		INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
sprintf()				INCL		POSIX_C_LANG_SUPPORT
srand()				INCL		POSIX_C_LANG_SUPPORT
sscanf()			INCL	INCL		POSIX_C_LANG_SUPPORT
strcat()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
strchr()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
strcmp()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
strcoll()				INCL		POSIX_C_LANG_SUPPORT
strcpy()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
strcspn()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
strerror()				INCL		POSIX_C_LANG_SUPPORT

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
strftime()			INCL	INCL		POSIX_C_LANG_SUPPORT
strlen()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
strncat()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
strncmp()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
strncpy()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
strpbrk()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
strrchr()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
strspn()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
strstr()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
strtod()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
strtof()				INCL		POSIX_C_LANG_SUPPORT
strtoimax()				INCL		POSIX_C_LANG_SUPPORT
strtok()				INCL		POSIX_C_LANG_SUPPORT
strtol()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
strtold()				INCL		POSIX_C_LANG_SUPPORT
strtoll()				INCL		POSIX_C_LANG_SUPPORT
strtoul()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
strtoull()				INCL		POSIX_C_LANG_SUPPORT
strtoumax()				INCL		POSIX_C_LANG_SUPPORT
strxfrm()				INCL		POSIX_C_LANG_SUPPORT
time()		INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
tolower()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
toupper()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
tzset()		INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
va_arg()			INCL	INCL		POSIX_C_LANG_SUPPORT
va_copy()			INCL	INCL		POSIX_C_LANG_SUPPORT

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
va_end()			INCL	INCL		POSIX_C_LANG_SUPPORT
va_start()			INCL	INCL		POSIX_C_LANG_SUPPORT
vsnprintf()			INCL	INCL		POSIX_C_LANG_SUPPORT
vsprintf()				INCL		POSIX_C_LANG_SUPPORT
vsscanf()				INCL		POSIX_C_LANG_SUPPORT
asctime()						POSIX_C_LANG_SUPPORT
asctime_r()		INCL	INCL	INCL		POSIX_C_LANG_SUPPORT_R
ctime_r()		INCL	INCL	INCL		POSIX_C_LANG_SUPPORT_R
gmtime_r()		INCL	INCL	INCL		POSIX_C_LANG_SUPPORT_R
localtime_r()		INCL	INCL	INCL		POSIX_C_LANG_SUPPORT_R
rand_r()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT_R
strerror_r()		INCL	INCL	INCL		POSIX_C_LANG_SUPPORT_R
strtok_r()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT_R
btowc()						POSIX_C_LANG_WIDE_CHAR
iswalnum()						POSIX_C_LANG_WIDE_CHAR
iswalpha()						POSIX_C_LANG_WIDE_CHAR
iswblank()						POSIX_C_LANG_WIDE_CHAR
iswcntrl()						POSIX_C_LANG_WIDE_CHAR
iswctype()						POSIX_C_LANG_WIDE_CHAR
iswdigit()						POSIX_C_LANG_WIDE_CHAR
iswgraph()						POSIX_C_LANG_WIDE_CHAR
iswlower()						POSIX_C_LANG_WIDE_CHAR
iswprint()						POSIX_C_LANG_WIDE_CHAR
iswpunct()						POSIX_C_LANG_WIDE_CHAR
iswspace()						POSIX_C_LANG_WIDE_CHAR
iswupper()						POSIX_C_LANG_WIDE_CHAR

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
iswxdigit()						POSIX_C_LANG_WIDE_CHAR
mblen()						POSIX_C_LANG_WIDE_CHAR
mbrlen()						POSIX_C_LANG_WIDE_CHAR
mbrtowc()						POSIX_C_LANG_WIDE_CHAR
mbsinit()						POSIX_C_LANG_WIDE_CHAR
mbsrtowcs()						POSIX_C_LANG_WIDE_CHAR
mbstowcs()						POSIX_C_LANG_WIDE_CHAR
mbtowc()						POSIX_C_LANG_WIDE_CHAR
swprintf()						POSIX_C_LANG_WIDE_CHAR
swscanf()						POSIX_C_LANG_WIDE_CHAR
towctrans()						POSIX_C_LANG_WIDE_CHAR
towlower()						POSIX_C_LANG_WIDE_CHAR
towupper()						POSIX_C_LANG_WIDE_CHAR
vswprintf()						POSIX_C_LANG_WIDE_CHAR
vswscanf()						POSIX_C_LANG_WIDE_CHAR
wcrtomb()						POSIX_C_LANG_WIDE_CHAR
wcscat()						POSIX_C_LANG_WIDE_CHAR
wcschr()						POSIX_C_LANG_WIDE_CHAR
wcscmp()						POSIX_C_LANG_WIDE_CHAR
wcscoll()						POSIX_C_LANG_WIDE_CHAR
wcscpy()						POSIX_C_LANG_WIDE_CHAR
wcscspn()						POSIX_C_LANG_WIDE_CHAR
wcsftime()						POSIX_C_LANG_WIDE_CHAR
wcslen()						POSIX_C_LANG_WIDE_CHAR
wcsncat()						POSIX_C_LANG_WIDE_CHAR
wcsncmp()						POSIX_C_LANG_WIDE_CHAR

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
wcsncpy()						POSIX_C_LANG_WIDE_CHAR
wcspbrk()						POSIX_C_LANG_WIDE_CHAR
wcsrchr()						POSIX_C_LANG_WIDE_CHAR
wcsrtombs()						POSIX_C_LANG_WIDE_CHAR
wcsspn()						POSIX_C_LANG_WIDE_CHAR
wcsstr()						POSIX_C_LANG_WIDE_CHAR
wcstod()						POSIX_C_LANG_WIDE_CHAR
wcstof()						POSIX_C_LANG_WIDE_CHAR
wcstoimax()						POSIX_C_LANG_WIDE_CHAR
wcstok()						POSIX_C_LANG_WIDE_CHAR
wcstol()						POSIX_C_LANG_WIDE_CHAR
wcstold()						POSIX_C_LANG_WIDE_CHAR
wcstoll()						POSIX_C_LANG_WIDE_CHAR
wcstombs()						POSIX_C_LANG_WIDE_CHAR
wcstoul()						POSIX_C_LANG_WIDE_CHAR
wcstoull()						POSIX_C_LANG_WIDE_CHAR
wcstoumax()						POSIX_C_LANG_WIDE_CHAR
wcsxfrm()						POSIX_C_LANG_WIDE_CHAR
wctob()						POSIX_C_LANG_WIDE_CHAR
wctomb()						POSIX_C_LANG_WIDE_CHAR
wctrans()						POSIX_C_LANG_WIDE_CHAR
wctype()						POSIX_C_LANG_WIDE_CHAR
wmemchr()						POSIX_C_LANG_WIDE_CHAR
wmemcmp()						POSIX_C_LANG_WIDE_CHAR
wmemcpy()						POSIX_C_LANG_WIDE_CHAR
wmemmove()						POSIX_C_LANG_WIDE_CHAR

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
wmemset()						POSIX_C_LANG_WIDE_CHAR
clearerr()		INCL	INCL	INCL		POSIX_DEVICE_IO
close()		INCL	INCL	INCL		POSIX_DEVICE_IO
fclose()		INCL	INCL	INCL		POSIX_DEVICE_IO
fdopen()			INCL	INCL		POSIX_DEVICE_IO
feof()		INCL	INCL	INCL		POSIX_DEVICE_IO
ferror()		INCL	INCL	INCL		POSIX_DEVICE_IO
fflush()		INCL	INCL	INCL		POSIX_DEVICE_IO
fgetc()		INCL	INCL	INCL		POSIX_DEVICE_IO
fgets()		INCL	INCL	INCL		POSIX_DEVICE_IO
fileno()		INCL	INCL	INCL		POSIX_DEVICE_IO
fopen()		INCL	INCL	INCL		POSIX_DEVICE_IO
fprintf()		INCL	INCL	INCL		POSIX_DEVICE_IO
fputc()				INCL		POSIX_DEVICE_IO
fputs()				INCL		POSIX_DEVICE_IO
fread()		INCL	INCL	INCL		POSIX_DEVICE_IO
freopen()		INCL	INCL	INCL		POSIX_DEVICE_IO
fscanf()				INCL		POSIX_DEVICE_IO
fwrite()		INCL	INCL	INCL		POSIX_DEVICE_IO
getc()				INCL		POSIX_DEVICE_IO
getchar()				INCL		POSIX_DEVICE_IO
gets()						POSIX_DEVICE_IO
open()		INCL	INCL	INCL		POSIX_DEVICE_IO
perror()				INCL		POSIX_DEVICE_IO
printf()				INCL		POSIX_DEVICE_IO
putc()				INCL		POSIX_DEVICE_IO

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
putchar()				INCL		POSIX_DEVICE_IO
puts()				INCL		POSIX_DEVICE_IO
read()		INCL	INCL	INCL		POSIX_DEVICE_IO
scanf()				INCL		POSIX_DEVICE_IO
setbuf()						POSIX_DEVICE_IO
setvbuf()				INCL		POSIX_DEVICE_IO
ungetc()				INCL		POSIX_DEVICE_IO
vfprintf()			INCL	INCL		POSIX_DEVICE_IO
vfscanf()				INCL		POSIX_DEVICE_IO
vprintf()				INCL		POSIX_DEVICE_IO
vscanf()				INCL		POSIX_DEVICE_IO
write()		INCL	INCL	INCL		POSIX_DEVICE_IO
cfgetispeed()						POSIX_DEVICE_SPECIFIC
cfgetospeed()						POSIX_DEVICE_SPECIFIC
cfsetispeed()						POSIX_DEVICE_SPECIFIC
cfsetospeed()						POSIX_DEVICE_SPECIFIC
ctermid()						POSIX_DEVICE_SPECIFIC
isatty()						POSIX_DEVICE_SPECIFIC
tcdrain()						POSIX_DEVICE_SPECIFIC
tcflow()						POSIX_DEVICE_SPECIFIC
tcflush()						POSIX_DEVICE_SPECIFIC
tcgetattr()						POSIX_DEVICE_SPECIFIC
tcsendbreak()						POSIX_DEVICE_SPECIFIC
tcsetattr()						POSIX_DEVICE_SPECIFIC
ttyname()						POSIX_DEVICE_SPECIFIC
ttyname_r()						POSIX_DEVICE_SPECIFIC_R

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
FD_CLR()		INCL	INCL	INCL	YES	POSIX_DEVICE_IO
FD_ISSET()		INCL	INCL	INCL	YES	POSIX_DEVICE_IO
FD_SET()		INCL	INCL	INCL	YES	POSIX_DEVICE_IO
FD_ZERO()		INCL	INCL	INCL	YES	POSIX_DEVICE_IO
pselect()				INCL	YES	POSIX_DEVICE_IO
select()		INCL	INCL	INCL	YES	POSIX_DEVICE_IO
dup()				INCL		POSIX_FD_MGMT
dup2()			INCL	INCL		POSIX_FD_MGMT
fcntl()			INCL	INCL		POSIX_FD_MGMT
fgetpos()				INCL		POSIX_FD_MGMT
fseek()		INCL	INCL	INCL		POSIX_FD_MGMT
fseeko()		INCL	INCL	INCL		POSIX_FD_MGMT
fsetpos()				INCL		POSIX_FD_MGMT
ftell()		INCL	INCL	INCL		POSIX_FD_MGMT
ftello()		INCL	INCL	INCL		POSIX_FD_MGMT
ftruncate()	INCL	INCL	INCL	INCL		POSIX_FD_MGMT
lseek()		INCL	INCL	INCL		POSIX_FD_MGMT
rewind()				INCL		POSIX_FD_MGMT
mkfifo()			INCL	INCL		POSIX_FIFO
chmod()			INCL	INCL		POSIX_FILE_ATTRIBUTES
chown()			INCL	INCL		POSIX_FILE_ATTRIBUTES
fchmod()				INCL		POSIX_FILE_ATTRIBUTES
fchown()				INCL		POSIX_FILE_ATTRIBUTES
umask()		INCL	INCL	INCL		POSIX_FILE_ATTRIBUTES
flockfile()			INCL	INCL		POSIX_FILE_LOCKING
ftrylockfile()			INCL	INCL		POSIX_FILE_LOCKING

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
funlockfile()			INCL	INCL		POSIX_FILE_LOCKING
getc_unlocked()				INCL		POSIX_FILE_LOCKING
getchar_unlocked()				INCL		POSIX_FILE_LOCKING
putc_unlocked()				INCL		POSIX_FILE_LOCKING
putchar_unlocked()				INCL		POSIX_FILE_LOCKING
access()		INCL	INCL	INCL		POSIX_FILE_SYSTEM
chdir()		INCL	INCL	INCL		POSIX_FILE_SYSTEM
closedir()		INCL	INCL	INCL		POSIX_FILE_SYSTEM
creat()		INCL	INCL	INCL		POSIX_FILE_SYSTEM
fchdir()						POSIX_FILE_SYSTEM
fpathconf()				INCL		POSIX_FILE_SYSTEM
fstat()		INCL	INCL	INCL		POSIX_FILE_SYSTEM
fstatvfs()						POSIX_FILE_SYSTEM
getcwd()		INCL	INCL	INCL		POSIX_FILE_SYSTEM
link()		INCL	INCL	INCL		POSIX_FILE_SYSTEM
mkdir()		INCL	INCL	INCL		POSIX_FILE_SYSTEM
mkstemp()						POSIX_FILE_SYSTEM
opendir()		INCL	INCL	INCL		POSIX_FILE_SYSTEM
pathconf()				INCL		POSIX_FILE_SYSTEM
readdir()		INCL	INCL	INCL		POSIX_FILE_SYSTEM
remove()		INCL	INCL	INCL		POSIX_FILE_SYSTEM
rename()		INCL	INCL	INCL		POSIX_FILE_SYSTEM
rewinddir()		INCL	INCL	INCL		POSIX_FILE_SYSTEM
rmdir()		INCL	INCL	INCL		POSIX_FILE_SYSTEM
stat()	INCL	INCL	INCL	INCL		POSIX_FILE_SYSTEM
statvfs()						POSIX_FILE_SYSTEM

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
S_ISBLK()		INCL	INCL	INCL		POSIX_FILE_SYSTEM
S_ISCHR()		INCL	INCL	INCL		POSIX_FILE_SYSTEM
S_ISDIR()		INCL	INCL	INCL		POSIX_FILE_SYSTEM
S_ISFIFO()		INCL	INCL	INCL		POSIX_FILE_SYSTEM
S_ISREG()		INCL	INCL	INCL		POSIX_FILE_SYSTEM
S_ISLNK()		INCL	INCL	INCL		POSIX_FILE_SYSTEM
S_ISSOCK()				INCL		POSIX_FILE_SYSTEM
S_TYPEISMQ()						POSIX_FILE_SYSTEM
S_TYPEISSEM()						POSIX_FILE_SYSTEM
S_TYPEISSHM()						POSIX_FILE_SYSTEM
S_TYPEISTMO()						POSIX_FILE_SYSTEM
tmpfile()				INCL		POSIX_FILE_SYSTEM
tmpnam()						POSIX_FILE_SYSTEM
truncate()						POSIX_FILE_SYSTEM
unlink()		INCL	INCL	INCL		POSIX_FILE_SYSTEM
utime()						POSIX_FILE_SYSTEM
utimes()						
readdir_r()		INCL	INCL	INCL		POSIX_FILE_SYSTEM_R
glob()						POSIX_FILE_SYSTEM_GLOB
globfree()						POSIX_FILE_SYSTEM_GLOB
setpgid()						POSIX_JOB_CONTROL
tcgetpgrp()						POSIX_JOB_CONTROL
tcsetpgrp()						POSIX_JOB_CONTROL
_Exit()			MP	MP		POSIX_MULTI_PROCESS
_exit()			MP	MP		POSIX_MULTI_PROCESS
assert()				MP		POSIX_MULTI_PROCESS

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
atexit()			MP	MP		POSIX_MULTI_PROCESS
clock()			MP	MP		POSIX_MULTI_PROCESS
execl()			MP	MP		POSIX_MULTI_PROCESS
execle()			MP	MP		POSIX_MULTI_PROCESS
execlp()						POSIX_MULTI_PROCESS
execv()			MP	MP		POSIX_MULTI_PROCESS
execve()			MP	MP		POSIX_MULTI_PROCESS
execvp()						POSIX_MULTI_PROCESS
exit()			MP	MP		POSIX_MULTI_PROCESS
fork()			MP	MP		POSIX_MULTI_PROCESS
jgetpgid()						POSIX_MULTI_PROCESS
getpgrp()			MP	MP		POSIX_MULTI_PROCESS
getpid()			MP	MP		POSIX_MULTI_PROCESS
getppid()			MP	MP		POSIX_MULTI_PROCESS
getsid()						POSIX_MULTI_PROCESS
setsid()				MP		POSIX_MULTI_PROCESS
sleep()			MP	MP		POSIX_MULTI_PROCESS
times()			MP	MP		POSIX_MULTI_PROCESS
wait()				MP		POSIX_MULTI_PROCESS
waitid()						POSIX_MULTI_PROCESS
waitpid()			MP	MP		POSIX_MULTI_PROCESS
accept()			INCL	INCL	YES	POSIX_NETWORKING
bind()	INCL	INCL	INCL	INCL	YES	POSIX_NETWORKING
connect()	INCL	INCL	INCL	INCL	YES	POSIX_NETWORKING
endhostent()				INCL	YES	POSIX_NETWORKING
endnetent()				INCL	YES	POSIX_NETWORKING

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
endprotoent()				INCL	YES	POSIX_NETWORKING
endservent()				INCL	YES	POSIX_NETWORKING
freeaddrinfo()	INCL	INCL	INCL	INCL	YES	POSIX_NETWORKING
gai_strerror()				INCL	YES	POSIX_NETWORKING
getaddrinfo()	INCL	INCL	INCL	INCL	YES	POSIX_NETWORKING
gethostent()				INCL	YES	POSIX_NETWORKING
gethostname()		INCL	INCL	INCL		POSIX_NETWORKING
getnameinfo()	INCL	INCL	INCL	INCL	YES	POSIX_NETWORKING
getnetbyaddr()				INCL	YES	POSIX_NETWORKING
getnetbyname()				INCL	YES	POSIX_NETWORKING
getnetent()				INCL	YES	POSIX_NETWORKING
getpeername()	INCL	INCL	INCL	INCL	YES	POSIX_NETWORKING
getprotobyname()				INCL	YES	POSIX_NETWORKING
getprotobynumber()				INCL	YES	POSIX_NETWORKING
getprotoent()				INCL	YES	POSIX_NETWORKING
getservbyname()				INCL	YES	POSIX_NETWORKING
getservbyport()				INCL	YES	POSIX_NETWORKING
getservent()				INCL	YES	POSIX_NETWORKING
getsockname()	INCL	INCL	INCL	INCL	YES	POSIX_NETWORKING
getsockopt()	INCL	INCL	INCL	INCL	YES	POSIX_NETWORKING
htonl()	INCL	INCL	INCL	INCL		POSIX_NETWORKING
htons()	INCL	INCL	INCL	INCL		POSIX_NETWORKING
if_freenameindex()				INCL	YES	POSIX_NETWORKING
if_indextoname()				INCL	YES	POSIX_NETWORKING
if_nameindex()				INCL	YES	POSIX_NETWORKING
if_nametoindex()				INCL	YES	POSIX_NETWORKING

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
inet_addr()				INCL	YES	POSIX_NETWORKING
inet_ntoa()				INCL	YES	POSIX_NETWORKING
inet_ntop()	INCL	INCL	INCL	INCL	YES	POSIX_NETWORKING
inet_pton()	INCL	INCL	INCL	INCL	YES	POSIX_NETWORKING
listen()			INCL	INCL	YES	POSIX_NETWORKING
ntohl()	INCL	INCL	INCL	INCL		POSIX_NETWORKING
ntohs()	INCL	INCL	INCL	INCL		POSIX_NETWORKING
recv()	INCL	INCL	INCL	INCL	YES	POSIX_NETWORKING
recvfrom()	INCL	INCL	INCL	INCL	YES	POSIX_NETWORKING
recvmsg()				INCL	YES	POSIX_NETWORKING
send()	INCL	INCL	INCL	INCL	YES	POSIX_NETWORKING
sendmsg()				INCL	YES	POSIX_NETWORKING
sendto()	INCL	INCL	INCL	INCL	YES	POSIX_NETWORKING
sethostent()				INCL	YES	POSIX_NETWORKING
setnetent()				INCL	YES	POSIX_NETWORKING
setprotoent()				INCL	YES	POSIX_NETWORKING
setservent()				INCL	YES	POSIX_NETWORKING
setsockopt()	INCL	INCL	INCL	INCL	YES	POSIX_NETWORKING
shutdown()	INCL	INCL	INCL	INCL	YES	POSIX_NETWORKING
sockatmark()				INCL	YES	POSIX_NETWORKING
socket()	INCL	INCL	INCL	INCL	YES	POSIX_NETWORKING
socketpair()				INCL	YES	POSIX_NETWORKING
pipe()			INCL	INCL		POSIX_PIPE
regerror()						POSIX_REGEXP
regexec()						POSIX_REGEXP
regfree()						POSIX_REGEXP

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
pthread_rwlock_destroy()				INCL		POSIX_RW_LOCKS
pthread_rwlock_init()				INCL		POSIX_RW_LOCKS
pthread_rwlock_rdlock()				INCL		POSIX_RW_LOCKS
pthread_rwlock_tryrdlock()				INCL		POSIX_RW_LOCKS
pthread_rwlock_trywrlock()				INCL		POSIX_RW_LOCKS
pthread_rwlock_unlock()				INCL		POSIX_RW_LOCKS
pthread_rwlock_wrlock()				INCL		POSIX_RW_LOCKS
pthread_rwlockattr_destroy()				INCL		POSIX_RW_LOCKS
pthread_rwlockattr_init()				INCL		POSIX_RW_LOCKS
pthread_rwlockattr_ getpshared()						POSIX_RW_LOCKS
pthread_rwlockattr_ setpshared()						POSIX_RW_LOCKS
pthread_rwlock_timedrdlock()				INCL		POSIX_RW_LOCKS
pthread_rwlock_timedwrlock()				INCL		POSIX_RW_LOCKS
pclose()						POSIX_SHELL_FUNC
popen()						POSIX_SHELL_FUNC
system()						POSIX_SHELL_FUNC
wordexp()						POSIX_SHELL_FUNC
wordfree()						POSIX_SHELL_FUNC
siglongjmp()			INCL	INCL		POSIX_SIGNAL_JUMP
sigsetjmp()			INCL	INCL		POSIX_SIGNAL_JUMP
abort()			INCL	INCL		POSIX_SIGNALS
alarm()	INCL	INCL	INCL	INCL		POSIX_SIGNALS
kill()			INCL	INCL		POSIX_SIGNALS
pause()	INCL	INCL	INCL	INCL		POSIX_SIGNALS
raise()			INCL	INCL		POSIX_SIGNALS
sigaction()	INCL	INCL	INCL	INCL		POSIX_SIGNALS

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
sigaddset()	INCL	INCL	INCL	INCL		POSIX_SIGNALS
sigdelset()	INCL	INCL	INCL	INCL		POSIX_SIGNALS
sigemptyset()	INCL	INCL	INCL	INCL		POSIX_SIGNALS
sigfillset()	INCL	INCL	INCL	INCL		POSIX_SIGNALS
sigismember()	INCL	INCL	INCL	INCL		POSIX_SIGNALS
signal()				INCL		POSIX_SIGNALS
sigpending()	INCL	INCL	INCL	INCL		POSIX_SIGNALS
sigprocmask()				INCL		POSIX_SIGNALS
sigsuspend()	INCL	INCL	INCL	INCL		POSIX_SIGNALS
sigwait()	INCL	INCL	INCL	INCL		POSIX_SIGNALS
confstr()				INCL		POSIX_SINGLE_PROCESS
getenv()			INCL	INCL		POSIX_SINGLE_PROCESS
setenv()				INCL		POSIX_SINGLE_PROCESS
sysconf()			INCL	INCL		POSIX_SINGLE_PROCESS
uname()			INCL	INCL		POSIX_SINGLE_PROCESS
unsetenv()				INCL		POSIX_SINGLE_PROCESS
fnmatch()						POSIX_C_LIB_EXT
getopt()						POSIX_C_LIB_EXT
lstat()			INCL	INCL		POSIX_SYMBOLIC_LINKS
readlink()						POSIX_SYMBOLIC_LINKS
symlink()						POSIX_SYMBOLIC_LINKS
getgrgid()						POSIX_SYSTEM_DATABASE
getgrnam()						POSIX_SYSTEM_DATABASE
getpwnam()						POSIX_SYSTEM_DATABASE
getpwuid()						POSIX_SYSTEM_DATABASE
getgrgid_r()						POSIX_SYSTEM_DATABASE_R

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
getgrnam_r()						POSIX_SYSTEM_DATABASE_R
getpwnam_r()						POSIX_SYSTEM_DATABASE_R
getpwuid_r()						POSIX_SYSTEM_DATABASE_R
pthread_atfork()			INCL	INCL		POSIX_THREADS_BASE
pthread_attr_destroy()	INCL	INCL	INCL	INCL		POSIX_THREADS_BASE
pthread_attr_getdetachstate()			INCL	INCL		POSIX_THREADS_BASE
pthread_attr_getschedparam()	INCL	INCL	INCL	INCL		POSIX_THREADS_BASE
pthread_attr_init()	INCL	INCL	INCL	INCL		POSIX_THREADS_BASE
pthread_attr_setdetachstate()			INCL	INCL		POSIX_THREADS_BASE
pthread_attr_setschedparam()	INCL	INCL	INCL	INCL		POSIX_THREADS_BASE
pthread_cancel()			INCL	INCL		POSIX_THREADS_BASE
pthread_cleanup_pop()			INCL	INCL		POSIX_THREADS_BASE
pthread_cleanup_push()			INCL	INCL		POSIX_THREADS_BASE
pthread_cond_broadcast()		INCL	INCL	INCL		POSIX_THREADS_BASE
pthread_cond_destroy()		INCL	INCL	INCL		POSIX_THREADS_BASE
pthread_cond_init()		INCL	INCL	INCL		POSIX_THREADS_BASE
pthread_cond_signal()		INCL	INCL	INCL		POSIX_THREADS_BASE
pthread_cond_timedwait()		INCL	INCL	INCL		POSIX_THREADS_BASE
pthread_cond_wait()		INCL	INCL	INCL		POSIX_THREADS_BASE
pthread_condattr_destroy()		INCL	INCL	INCL		POSIX_THREADS_BASE
pthread_condattr_init()		INCL	INCL	INCL		POSIX_THREADS_BASE
pthread_create()	INCL	INCL	INCL	INCL		POSIX_THREADS_BASE
pthread_detach()			INCL	INCL		POSIX_THREADS_BASE
pthread_equal()	INCL	INCL	INCL	INCL		POSIX_THREADS_BASE
pthread_exit()			INCL	INCL		POSIX_THREADS_BASE
pthread_getspecific()		INCL	INCL	INCL		POSIX_THREADS_BASE

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
pthread_join()			INCL	INCL		POSIX_THREADS_BASE
pthread_key_create()		INCL	INCL	INCL		POSIX_THREADS_BASE
pthread_key_delete()			INCL	INCL		POSIX_THREADS_BASE
pthread_kill()			INCL	INCL		POSIX_THREADS_BASE
pthread_mutex_destroy()			INCL	INCL		POSIX_THREADS_BASE
pthread_mutex_init()	INCL	INCL	INCL	INCL		POSIX_THREADS_BASE
pthread_mutex_lock()	INCL	INCL	INCL	INCL		POSIX_THREADS_BASE
pthread_mutex_timedlock()	INCL	INCL	INCL	INCL		POSIX_THREADS_BASE
pthread_mutex_trylock()	INCL	INCL	INCL	INCL		POSIX_THREADS_BASE
pthread_mutex_unlock()	INCL	INCL	INCL	INCL		POSIX_THREADS_BASE
pthread_mutexattr_destroy()	INCL	INCL	INCL	INCL		POSIX_THREADS_BASE
pthread_mutexattr_init()	INCL	INCL	INCL	INCL		POSIX_THREADS_BASE
pthread_once()	INCL	INCL	INCL	INCL		POSIX_THREADS_BASE
pthread_self()	INCL	INCL	INCL	INCL		POSIX_THREADS_BASE
pthread_setcancelstate()			INCL	INCL		POSIX_THREADS_BASE
pthread_setcanceltype()			INCL	INCL		POSIX_THREADS_BASE
pthread_setspecific()		INCL	INCL	INCL		POSIX_THREADS_BASE
pthread_sigmask()	INCL	INCL	INCL	INCL		POSIX_THREADS_BASE
pthread_testcancel()			INCL	INCL		POSIX_THREADS_BASE
getegid()			INCL	INCL		POSIX_USER_GROUPS
geteuid()			INCL	INCL		POSIX_USER_GROUPS
getgid()			INCL	INCL		POSIX_USER_GROUPS
getgroups()			INCL	INCL		POSIX_USER_GROUPS
getlogin()						POSIX_USER_GROUPS
getuid()			INCL	INCL		POSIX_USER_GROUPS
setegid()			INCL	INCL		POSIX_USER_GROUPS

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
seteuid()			INCL	INCL		POSIX_USER_GROUPS
setgid()			INCL	INCL		POSIX_USER_GROUPS
setuid()			INCL	INCL		POSIX_USER_GROUPS
getlogin_r()				INCL		POSIX_USER_GROUPS_R
fgetwc()						POSIX_WIDE_CHAR_ DEVICE_IO
fgetws()						POSIX_WIDE_CHAR_ DEVICE_IO
fputwc()						POSIX_WIDE_CHAR_ DEVICE_IO
fputws()						POSIX_WIDE_CHAR_ DEVICE_IO
fwide()						POSIX_WIDE_CHAR_ DEVICE_IO
fwprintf()						POSIX_WIDE_CHAR_ DEVICE_IO
fwscanf()						POSIX_WIDE_CHAR_ DEVICE_IO
getwc()						POSIX_WIDE_CHAR_ DEVICE_IO
getwchar()						POSIX_WIDE_CHAR_ DEVICE_IO
putwc()						POSIX_WIDE_CHAR_ DEVICE_IO
putwchar()						POSIX_WIDE_CHAR_ DEVICE_IO
ungetwc()						POSIX_WIDE_CHAR_ DEVICE_IO
vfwprintf()						POSIX_WIDE_CHAR_ DEVICE_IO
vfwscanf()						POSIX_WIDE_CHAR_ DEVICE_IO

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
vwprintf()						POSIX_WIDE_CHAR_ DEVICE_IO
vwscanf()						POSIX_WIDE_CHAR_ DEVICE_IO
wprintf()						POSIX_WIDE_CHAR_ DEVICE_IO
wscanf()						POSIX_WIDE_CHAR_ DEVICE_IO
_tolower()						XSI_C_LANG_SUPPORT
_toupper()						XSI_C_LANG_SUPPORT
a64l()						XSI_C_LANG_SUPPORT
drand48()						XSI_C_LANG_SUPPORT
erand48()						XSI_C_LANG_SUPPORT
ffs()						XSI_C_LANG_SUPPORT
getdate()						XSI_C_LANG_SUPPORT
getsubopt()						POSIX_C_LIB_EXT
hcreate()						XSI_C_LANG_SUPPORT
hdestroy()						XSI_C_LANG_SUPPORT
hsearch()						XSI_C_LANG_SUPPORT
iconv()						POSIX_I18N
iconv_close()						POSIX_I18N
iconv_open()						POSIX_I18N
initstate()						XSI_C_LANG_SUPPORT
insque()						XSI_C_LANG_SUPPORT
isascii()						XSI_C_LANG_SUPPORT
jrand48()						XSI_C_LANG_SUPPORT
l64a()						XSI_C_LANG_SUPPORT
lcong48()						XSI_C_LANG_SUPPORT

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
lfind()						XSI_C_LANG_SUPPORT
lrand48()						XSI_C_LANG_SUPPORT
lsearch()						XSI_C_LANG_SUPPORT
memccpy()						XSI_C_LANG_SUPPORT
mrand48()						XSI_C_LANG_SUPPORT
nrand48()						XSI_C_LANG_SUPPORT
random()						XSI_C_LANG_SUPPORT
remque()						XSI_C_LANG_SUPPORT
seed48()						XSI_C_LANG_SUPPORT
setstate()						XSI_C_LANG_SUPPORT
srand48()						XSI_C_LANG_SUPPORT
srandom()						XSI_C_LANG_SUPPORT
strcasecmp()						POSIX_C_LIB_EXT
strdup()						POSIX_C_LIB_EXT
strfmon()						POSIX_C_LIB_EXT
strncasecmp()						POSIX_C_LIB_EXT
strptime()						XSI_C_LANG_SUPPORT
swab()						XSI_C_LANG_SUPPORT
tdelete()						XSI_C_LANG_SUPPORT
tfind()						XSI_C_LANG_SUPPORT
toascii()						XSI_C_LANG_SUPPORT
tsearch()						XSI_C_LANG_SUPPORT
twalk()						XSI_C_LANG_SUPPORT
dbm_clearerr()						XSI_DBM
dbm_close()						XSI_DBM
dbm_delete()						XSI_DBM

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
dbm_error()						XSI_DBM
dbm_fetch()						XSI_DBM
dbm_firstkey()						XSI_DBM
dbm_nextkey()						XSI_DBM
dbm_open()						XSI_DBM
dbm_store()						XSI_DBM
fmtmsg()						XSI_DEVICE_IO
poll()						POSIX_DEVICE_IO
pread()						POSIX_DEVICE_IO
pwrite()						POSIX_DEVICE_IO
readv()						XSI_DEVICE_IO
writev()						XSI_DEVICE_IO
grantpt()						XSI_DEVICE_SPECIFIC
posix_openpt()						XSI_DEVICE_SPECIFIC
ptsname()						XSI_DEVICE_SPECIFIC
unlockpt()						XSI_DEVICE_SPECIFIC
dlclose()						POSIX_DYNAMIC_LINKING
dlerror()						POSIX_DYNAMIC_LINKING
dlopen()						POSIX_DYNAMIC_LINKING
dlsym()						POSIX_DYNAMIC_LINKING
basename()						XSI_FILE_SYSTEM
dirname()						XSI_FILE_SYSTEM
ftw()						XSI_FILE_SYSTEM
lchown()						POSIX_SYMBOLIC_LINKS
lockf()						XSI_FILE_SYSTEM
mknod()						XSI_FILE_SYSTEM

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
nftw()						XSI_FILE_SYSTEM
realpath()						XSI_FILE_SYSTEM
seekdir()						XSI_FILE_SYSTEM
sync()						XSI_FILE_SYSTEM
telldir()						XSI_FILE_SYSTEM
tempnam()						XSI_FILE_SYSTEM
catclose()						POSIX_I18N
catgets()						POSIX_I18N
catopen()						POSIX_I18N
msgctl()						XSI_IPC
msgget()						XSI_IPC
msgrcv()						XSI_IPC
msgsnd()						XSI_IPC
nl_langinfo()						POSIX_I18N
semctl()						XSI_IPC
semget()						XSI_IPC
semop()						XSI_IPC
shmat()						XSI_IPC
shmctl()						XSI_IPC
shmdt()						XSI_IPC
shmget()						XSI_IPC
ftok()						XSI_IPC
tcgetsid()						POSIX_JOB_CONTROL
_longjmp()						XSI_JUMP
_setjmp()						XSI_JUMP
j0()						XSI_MATH

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
j1()						XSI_MATH
jn()						XSI_MATH
y0()						XSI_MATH
y1()						XSI_MATH
yn()						XSI_MATH
getpriority()						XSI_MULTI_PROCESS
getrlimit()						XSI_MULTI_PROCESS
getrusage()						XSI_MULTI_PROCESS
nice()						XSI_MULTI_PROCESS
setpgrp()						XSI_MULTI_PROCESS
setpriority()						XSI_MULTI_PROCESS
setrlimit()						XSI_MULTI_PROCESS
ulimit()						XSI_MULTI_PROCESS
killpg()						XSI_SIGNALS
sigaltstack()						XSI_SIGNALS
sighold()						XSI_SIGNALS
sigignore()						XSI_SIGNALS
siginterrupt()						XSI_SIGNALS
sigpause()						XSI_SIGNALS
sigrelse()						XSI_SIGNALS
sigset()						XSI_SIGNALS
gethostid()						XSI_SINGLE_PROCESS
gettimeofday()						XSI_SINGLE_PROCESS
putenv()						XSI_SINGLE_PROCESS
endpwent()						XSI_SYSTEM_DATABASE
getpwent()						XSI_SYSTEM_DATABASE

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
setpwent()						XSI_SYSTEM_DATABASE
closelog()						XSI_SYSTEM_LOGGING
openlog()						XSI_SYSTEM_LOGGING
setlogmask()						XSI_SYSTEM_LOGGING
syslog()						XSI_SYSTEM_LOGGING
LOG_MASK()						XSI_SYSTEM_LOGGING
pthread_mutexattr_gettype()				INCL		POSIX_THREADS_EXT
pthread_mutexattr_settype()				INCL		POSIX_THREADS_EXT
pthread_attr_getguardsize()			INCL	INCL		POSIX_THREADS_EXT
pthread_attr_setguardsize()			INCL	INCL		POSIX_THREADS_EXT
pthread_getconcurrency()		INCL	INCL	INCL		XSI_THREADS_EXT
pthread_setconcurrency()		INCL	INCL	INCL		XSI_THREADS_EXT
getitimer()						XSI_TIMERS
setitimer()						XSI_TIMERS
endgrent()						XSI_USER_GROUPS
endutxent()						XSI_USER_GROUPS
getgrent()						XSI_USER_GROUPS
getutxent()						XSI_USER_GROUPS
getutxid()						XSI_USER_GROUPS
getutxline()						XSI_USER_GROUPS
pututxline()						XSI_USER_GROUPS
setgrent()						XSI_USER_GROUPS
setregid()						XSI_USER_GROUPS
setreuid()						XSI_USER_GROUPS
setutxent()						XSI_USER_GROUPS
wcswidth()						XSI_WIDE_CHAR

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
wcwidth()						XSI_WIDE_CHAR
alphasort()						POSIX_FILE_SYSTEM_EXT
dirfd()						POSIX_FILE_SYSTEM_EXT
dprintf()						POSIX_DEVICE_IO_EXT
duplocale()						POSIX_MULTI_CONCURRENT_LOCALES
faccessat()						POSIX_FILE_SYSTEM_FD
fchmodat()						POSIX_FILE_ATTRIBUTES_FD
fchownat()						POSIX_FILE_ATTRIBUTES_FD
fdopendir()						POSIX_FILE_SYSTEM_FD
fexecve()						POSIX_MULTI_PROCESS_FD
fmemopen()						POSIX_DEVICE_IO_EXT
freelocale()						POSIX_MULTI_CONCURRENT_LOCALES
fstatat()						POSIX_FILE_SYSTEM_FD
futimens()						POSIX_FILE_SYSTEM_FD
getdelim()						POSIX_FILE_SYSTEM_EXT
getline()						POSIX_FILE_SYSTEM_EXT
isalnum_l()						POSIX_MULTI_CONCURRENT_LOCALES
isalpha_l()						POSIX_MULTI_CONCURRENT_LOCALES
isblank_l()						POSIX_MULTI_CONCURRENT_LOCALES
iscntrl_l()						POSIX_MULTI_CONCURRENT_LOCALES
isdigit_l()						POSIX_MULTI_CONCURRENT_LOCALES
isgraph_l()						POSIX_MULTI_CONCURRENT_LOCALES

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
islower_l()						POSIX_MULTI_CONCURRENT_LOCALES
isprint_l()						POSIX_MULTI_CONCURRENT_LOCALES
ispunct_l()						POSIX_MULTI_CONCURRENT_LOCALES
isspace_l()						POSIX_MULTI_CONCURRENT_LOCALES
isupper_l()						POSIX_MULTI_CONCURRENT_LOCALES
iswalnum_l()						POSIX_MULTI_CONCURRENT_LOCALES
iswalpha_l()						POSIX_MULTI_CONCURRENT_LOCALES
iswblank_l()						POSIX_MULTI_CONCURRENT_LOCALES
iswcntrl_l()						POSIX_MULTI_CONCURRENT_LOCALES
iswctype_l()						POSIX_MULTI_CONCURRENT_LOCALES
iswdigit_l()						POSIX_MULTI_CONCURRENT_LOCALES
iswgraph_l()						POSIX_MULTI_CONCURRENT_LOCALES
iswlower_l()						POSIX_MULTI_CONCURRENT_LOCALES
iswprint_l()						POSIX_MULTI_CONCURRENT_LOCALES
iswpunct_l()						POSIX_MULTI_CONCURRENT_LOCALES
iswspace_l()						POSIX_MULTI_CONCURRENT_LOCALES
iswupper_l()						POSIX_MULTI_CONCURRENT_LOCALES

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
iswxdigit_l()						POSIX_MULTI_CONCURRENT_LOCALES
isxdigit_l()						POSIX_MULTI_CONCURRENT_LOCALES
linkat()						POSIX_FILE_SYSTEM_FD
mbsnrtowcs()						POSIX_C_LANG_WIDE_CHAR_EXT
mkdirat()						POSIX_FILE_SYSTEM_FD
mkdtemp()						POSIX_FILE_SYSTEM_EXT
mkfifoat()						POSIX_FIFO_FD
mknodat()						POSIX_FIFO_FD
newlocale()						POSIX_MULTI_CONCURRENT_LOCALES
nl_langinfo_l()						POSIX_I18N
open_memstream()						POSIX_DEVICE_IO_EXT
open_wmemstream()						POSIX_C_LANG_WIDE_CHAR
openat()						POSIX_FILE_SYSTEM_FD
psiginfo()						POSIX_SIGNALS_EXT
psignal()						POSIX_SIGNALS_EXT
pthread_mutex_consistent()						_POSIX_ROBUST_MUTEXES
pthread_mutexattr_getrobust()						_POSIX_ROBUST_MUTEXES
pthread_mutexattr_setrobust()						_POSIX_ROBUST_MUTEXES
readlinkat()						POSIX_SYMBOLIC_LINKS_FD
renameat()						POSIX_FILE_SYSTEM_FD
scandir()						POSIX_FILE_SYSTEM_EXT
stpcpy()						POSIX_C_LIB_EXT
stpncpy()						POSIX_C_LIB_EXT
strcasecmp_l()						POSIX_MULTI_CONCURRENT_LOCALES

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
strcoll_l()						POSIX_MULTI_CONCURRENT_LOCALES
strerror_l()						POSIX_MULTI_CONCURRENT_LOCALES
strfmon_l()						POSIX_MULTI_CONCURRENT_LOCALES
strftime_l()						POSIX_MULTI_CONCURRENT_LOCALES
strncasecmp_l()						POSIX_MULTI_CONCURRENT_LOCALES
strndup()						POSIX_C_LIB_EXT
strnlen()						POSIX_C_LIB_EXT
strsignal()						POSIX_SIGNALS_EXT
strxfrm_l()						POSIX_MULTI_CONCURRENT_LOCALES
symlinkat()						POSIX_SYMBOLIC_LINKS_FD
tolower_l()						POSIX_MULTI_CONCURRENT_LOCALES
toupper_l()						POSIX_MULTI_CONCURRENT_LOCALES
towctrans_l()						POSIX_MULTI_CONCURRENT_LOCALES
towlower_l()						POSIX_MULTI_CONCURRENT_LOCALES
towupper_l()						POSIX_MULTI_CONCURRENT_LOCALES
unlinkat()						POSIX_FILE_SYSTEM_FD
uselocale()						POSIX_MULTI_CONCURRENT_LOCALES
utimensat()						POSIX_FILE_SYSTEM_FD
vdprintf()						POSIX_DEVICE_IO_EXT

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
wcpcpy()						POSIX_C_LANG_WIDE_CHAR_EXT
wcpncpy()						POSIX_C_LANG_WIDE_CHAR_EXT
wcscasecmp()						POSIX_C_LANG_WIDE_CHAR_EXT
wcscasecmp_l()						POSIX_MULTI_CONCURRENT_LOCALES
wcscoll_l()						POSIX_MULTI_CONCURRENT_LOCALES
wcsdup()						POSIX_C_LANG_WIDE_CHAR_EXT
wcsncasecmp()						POSIX_C_LANG_WIDE_CHAR_EXT
wcsncasemcp_l()						POSIX_MULTI_CONCURRENT_LOCALES
wcsnlen()						POSIX_C_LANG_WIDE_CHAR_EXT
wcsnrtombs()						POSIX_C_LANG_WIDE_CHAR_EXT
wcsxfrm_l()						POSIX_MULTI_CONCURRENT_LOCALES
wctrans_l()						POSIX_MULTI_CONCURRENT_LOCALES
wctype_l()						POSIX_MULTI_CONCURRENT_LOCALES
offsetof()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
INT8_C()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
INT16_C()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
INT32_C()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
UINT8_C()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
UINT16_C()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT

IEEE Std 1003.1-2008 API	Security	Safety Base	Safety Extended	General Purpose	Inter-UoC	POSIX Functionality Categories
UINT32_C()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
INTMAX_C()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT
UINTMAX_C()	INCL	INCL	INCL	INCL		POSIX_C_LANG_SUPPORT

A.2 POSIX API Rules

Safety Profile for POSIX Substitutions

In order to limit the amount of APIs certified within the Safety Profile API, substitutions were made. The substituted APIs perform an equivalent operation. The following list details which APIs were substituted:

- *ctime*() was substituted by *ctime_r*()
- *gmtime*() was substituted by *gmtime_r*()
- *asctime*() was substituted by *asctime_r*()
- *localtime*() was substituted by *localtime_r*()
- *rand*() was substituted by *rand_r*()
- *srand*() was substituted by *rand_r*()
- *sprintf*() was substituted by *snprintf*()
- *strerror*() was substituted by *strerror_r*()
- *rewind*() was substituted by *fseek*()
- *strtok*() was substituted by *strtok_r*()
- *setjmp*() was substituted by *sigsetjmp*()
- *longjmp*() was substituted by *siglongjmp*()
- *sigprocmask*() was substituted by *pthread_sigmask*()
- *fdatasync*() was substituted by *fsync*()
- *vsprintf*() was substituted by *vsnprintf*()
- *printf*() was substituted by *fprintf*()
- *signal*() was substituted by *sigaction*()

A.3 POSIX Enumeration Rules

Some POSIX APIs include enumeration constants that are used to configure API and operating environment-related behaviors. This section identifies the availability of these constants on a per FACE OSS Profile basis. In many cases, support for the constant is based on corresponding APIs being included in a FACE OSS Profile or not. Enumeration constants identified in the POSIX API definitions and included in but not defined in are also included.

Explanation for POSIX Enumeration Rules Table Format

INCL Included in the Profile indicated at top of column

Blank Excluded from the Profile indicated at top of column

Note: Blank items may be included in future editions of the FACE Technical Standard.

The enumerations for the POSIX thread detach state enumeration constants are supported in the following FACE Profiles:

Table 22: POSIX Thread Detach State Values

Enumeration	Security	Safety Base	Safety Extended	General Purpose
PTHREAD_CREATE_JOINABLE			INCL	INCL
PTHREAD_CREATE_DETACHED			INCL	INCL

The enumerations for the POSIX thread inherit scheduler enumeration constants are supported in the following FACE Profiles:

Table 23: POSIX Thread Inherit Scheduler Values

Enumeration	Security	Safety Base	Safety Extended	General Purpose
PTHREAD_INHERIT_SCHED	INCL	INCL	INCL	INCL
PTHREAD_EXPLICIT_SCHED	INCL	INCL	INCL	INCL

The enumerations for the POSIX thread inherit scheduler enumeration constants are supported in the following FACE Profiles:

Table 24: POSIX Thread Scheduler Policy Values

Enumeration	Security	Safety Base	Safety Extended	General Purpose
SCHED_FIFO	INCL	INCL	INCL	INCL
SCHED_RR	INCL	INCL	INCL	INCL

Enumeration	Security	Safety Base	Safety Extended	General Purpose
SCHED_SPORADIC				INCL
SCHED_OTHER				

The enumerations for the POSIX thread inherit scope enumeration constants are supported in the following FACE Profiles:

Table 25: POSIX Thread Scope Values

Attribute	Security	Safety Base	Safety Extended	General Purpose
PTHREAD_SCOPE_SYSTEM			MP	MP
PTHREAD_SCOPE_PROCESS	INCL	INCL	INCL	INCL

The enumerations for POSIX synchronization object visibility constants are supported in the following FACE Profiles:

Table 26: POSIX Mutex Scope Values

Attribute	Security	Safety Base	Safety Extended	General Purpose
PTHREAD_PROCESS_SHARED				MP
PTHREAD_PROCESS_PRIVATE				MP

The enumerations for POSIX mutex type constants are supported in the following FACE Profiles:

Table 27: POSIX Mutex Type Attribute Values

Attribute	Security	Safety Base	Safety Extended	General Purpose
PTHREAD_MUTEX_DEFAULT				INCL
PTHREAD_MUTEX_ERRORCHECK				INCL
PTHREAD_MUTEX_NORMAL				INCL
PTHREAD_MUTEX_RECURSIVE				INCL
PTHREAD_MUTEX_ROBUST				
PTHREAD_MUTEX_STALLED				

The enumerations for POSIX mutex protocol constants are supported in the following FACE Profiles:

Table 28: POSIX Mutex Protocol Values

Attribute	Security	Safety Base	Safety Extended	General Purpose
PTHREAD_PRIO_INHERIT				INCL
PTHREAD_PRIO_NONE				INCL
PTHREAD_PRIO_PROTECT	INCL	INCL	INCL	INCL

The enumerations for POSIX robust mutex constants are supported in the following FACE Profiles:

Table 29: POSIX Mutex Robustness Values

Attribute	Security	Safety Base	Safety Extended	General Purpose
PTHREAD_MUTEX_STALLED				
PTHREAD_MUTEX_ROBUST				

The enumerations for POSIX clock and timer source constants are supported in the following FACE Profiles:

Table 30: POSIX Clock and Timer Source Values and FACE Profiles

POSIX.1-2013 ConstantConstant	Security	Safety Base	Safety Extended	General Purpose
CLOCK_MONOTONIC	INCL	INCL	INCL	INCL
CLOCK_PROCESS_CPUTIME_ID				MP
CLOCK_REALTIME	INCL	INCL	INCL	INCL
CLOCK_THREAD_CPUTIME_ID	INCL	INCL	INCL	INCL
TIMER_ABSTIME	INCL	INCL	INCL	INCL

Note: Only CLOCK_MONOTONIC and CLOCK_REALTIME are available for use with POSIX condition variables.

The enumerations for setting and obtaining socket options are supported in the following FACE Profiles:

Table 31: POSIX Set Socket (Socket-Level) Option Values

POSIX.1-2013 Option Value	Security	Safety Base	Safety Extended	General Purpose
SO_ACCEPTCONN				
SO_BROADCAST				
SO_DEBUG				
SO_DONTROUTE				
SO_ERROR				
SO_KEEPALIVE				
SO_LINGER				
SO_OOBINLINE				
SO_RCVBUF	INCL	INCL	INCL	INCL
SO_RCVLOWAT				
SO_RCVTIMEO				
SO_REUSEADDR	INCL	INCL	INCL	INCL
SO_SNDBUF	INCL	INCL	INCL	INCL
SO_SNDLOWAT				
SO_SNDTIMEO				
SO_TYPE				
IP_MULTICAST_IF	INCL	INCL	INCL	INCL
IP_MULTICAST_TTL	INCL	INCL	INCL	INCL
IP_MULTICAST_LOOP	INCL	INCL	INCL	INCL
IP_ADD_MEMBERSHIP	INCL	INCL	INCL	INCL
IP_DROP_MEMBERSHIP	INCL	INCL	INCL	INCL

Note: The IP_xxx constants are not defined by POSIX but commonly supported and explicitly required by the FACE Technical Standard.

The following IPv6-related set socket options are supported in the following FACE Profiles:

Table 32: POSIX Set Socket (Use over IPv6 Internet Protocols) Option Values

POSIX.1-2013 Option Value	Security	Safety Base	Safety Extended	General Purpose
IPV6_JOIN_GROUP				
IPV6_LEAVE_GROUP				
IPV6_MULTICAST_HOPS				
IPV6_MULTICAST_IF	INCL	INCL	INCL	INCL
IPV6_MULTICAST_LOOP	INCL	INCL	INCL	INCL
IPV6_UNICAST_HOPS				
IPV6_V6ONLY				

The POSIX method *sigaction*() is included all FACE Profiles but the associated constants are supported in the FACE OSS Profiles as follows:

Table 33: POSIX Set Socket (Use over Internet Protocols) Option Values

POSIX.1-2013 Option Value	Security	Safety Base	Safety Extended	General Purpose
SOCK_STREAM			INCL	INCL
SOCK_DGRAM	INCL	INCL	INCL	INCL
SOCK_RAW				INCL
SOCK_SEQPACKET				INCL

The enumerations for POSIX *mmap*() constants are supported in the following FACE Profiles:

Table 34: POSIX mmap() Constant Values and FACE Profiles

POSIX.1-2013 Constant	Security	Safety Base	Safety Extended	General Purpose
MAP_FIXED				INCL
MAP_PRIVATE				INCL
MAP_SHARED	INCL	INCL	INCL	INCL

Table 35: POSIX sigaction() Flags

POSIX.1-2013 Option Value	Security	Safety Base	Safety Extended	General Purpose
SA_NOCLDSTOP				MP
SA_ONSTACK				
SA_RESETHAND	INCL	INCL	INCL	INCL
SA_RESTART				MP
SA_SIGINFO	INCL	INCL	INCL	INCL
SA_NOCLDWAIT				
SA_NODEFER				

The following constants are associated with the POSIX process spawn attribute set:

Table 36: POSIX Spawn Attribute Flags

Attribute	Security	Safety Base	Safety Extended	General Purpose
POSIX_SPAWN_RESETIDS			MP	MP
POSIX_SPAWN_SETPGROUP				MP
POSIX_SPAWN_SETSIGDEF			MP	MP
POSIX_SPAWN_SETSIGMASK			MP	MP
POSIX_SPAWN_SETSCHEDPARAM			MP	MP
POSIX_SPAWN_SETSCHEDULER			MP	MP

The following constants are associated with the POSIX process trace attribute set:

Table 37: POSIX Trace Attribute Flags

Attribute	Security	Safety Base	Safety Extended	General Purpose
POSIX_TRACE_ALL_EVENTS				
POSIX_TRACE_APPEND				
POSIX_TRACE_CLOSE_FOR_CHILD				
POSIX_TRACE_FILTER				

Attribute	Security	Safety Base	Safety Extended	General Purpose
POSIX_TRACE_FLUSH				
POSIX_TRACE_FLUSH_START				
POSIX_TRACE_FLUSH_STOP				
POSIX_TRACE_FLUSHING				
POSIX_TRACE_FULL				
POSIX_TRACE_LOOP				
POSIX_TRACE_NO_OVERRUN				
POSIX_TRACE_NOT_FLUSHING				
POSIX_TRACE_NOT_FULL				
POSIX_TRACE_INHERITED				
POSIX_TRACE_NOT_TRUNCATED				
POSIX_TRACE_OVERFLOW				
POSIX_TRACE_OVERRUN				
POSIX_TRACE_RESUME				
POSIX_TRACE_RUNNING				
POSIX_TRACE_START				
POSIX_TRACE_STOP				
POSIX_TRACE_SUSPENDED				
POSIX_TRACE_SYSTEM_EVENTS				
POSIX_TRACE_TRUNCATED_READ				
POSIX_TRACE_TRUNCATED_RECORD				
POSIX_TRACE_UNNAMED_USER_EVENT				
POSIX_TRACE_UNTIL_FULL				
POSIX_TRACE_WOPID_EVENTS				

A.4 Internet Networking Standards

This section identifies the IP-based networking standards and profiles required to promote interoperability between IP-based network software components. The goal of this section is to identify a minimum set of IETF RFC networking standards required for interoperability between UoCs.

FACE Reference Architecture-based implementations can include IP-based network interfaces connected to external environments. Interoperability with the external environments can require additional RFCs for interoperability. Such interoperability can include consideration of:

- DoD Joint Technical Architecture
- DoD IPv6 Standard Profiles for IPv6-capable Products
- A Profile for IPv6 in the U.S. Government
- ARINC Report 664: Aircraft Data Network

Note: The DoD standards above define overlapping and somewhat inconsistent groups of mandatory requirements for network standards. The defined standards divide into IPv4 requirements (which is the basis for ARINC 664), and IPv6 requirements providing a natural separation of the requirements into two independent network profiles. IPv4 networks represent the majority of the existing embedded network platforms currently in service, while IPv6 networks represent the emerging standards for future platforms.

The decision to support an IPv4 only or an IPv6 only network is an implementation decision.

Table 38: Basic Internetwork Capabilities

Standard	Description
RFC 0791	Internet Protocol (IP)
RFC 0768	User Datagram Protocol
RFC 1112	Host Extensions for IP Multicasting

Table 39: TCP Capabilities

Standard	Description
RFC 0793	Transmission Control Protocol
RFC 3390	Increasing TCP’s Initial Window

Table 40: IPv6 Capabilities

Standard	Description
RFC 2460	IPv6 Specification

Table 41: IPv4/IPv6 Transition Mechanisms

Standard	Description
RFC 4213	Basic Transition Mechanisms for IPv6 Hosts and Routers

A.5 Obsolete or Deprecated POSIX APIs

IEEE Std 1003.1-2008 describes some of the APIs listed in Table 21 as obsolete or deprecated. Such APIs are candidates for removal from Table 21 in future major revisions of the FACE Technical Standard. The following obsolete or deprecated methods are included in one or more FACE OSS Profiles:

- *pthread_getconcurrency*()
- *pthread_setconcurrency*()

A.6 ARINC 653 Inter-Partition Capabilities

The ARINC 653 standard defines the following capabilities for use for inter-partition (i.e., inter-UoC) communications:

- Sampling ports
- Queuing ports

These capabilities can also be utilized for intra-partition communications (along with other capabilities restricted to intra-partition use only). The inter-partition use of these capabilities is restricted for use only by the Transport Services and I/O Services.

A.7 POSIX API Usage Restrictions

The following restrictions apply to UoCs using the POSIX APIs as defined in the FACE Profiles:

- Section A.3 details restrictions on use of POSIX API constants per FACE Profile
- In the Security and Safety Profiles, the *mmap*() API is restricted to use with shared memory objects
- In the Security and Safety Profiles, mutex protocol use is limited to PTHREAD_PRIO_PROTECT (e.g., *pthread_mutexattr_setprotocol*() API)
- Use of *posix_devctl*() is restricted to use with sockets
- Section A.1 details the POSIX APIs whose inter-UoC usages are restricted to Transport Services and I/O Services
- Any call to *fork*() will be consecutively followed in the child process by a call to one of the permitted *exec**() interfaces

B FACE API Common Elements

B.1 Introduction

The FACE Technical Standard defines multiple APIs and those APIs are designed to have a common appearance. Elements such as the return status codes from FACE services used by more than one API are described in this appendix.

Note: The code in this document is formatted to align with the formatting constraints of the printed document.

B.2 FACE API Common Elements Type Definitions

FACE/common.idl

```
//! Source file: FACE/Common.idl

#ifndef __FACE_COMMON
#define __FACE_COMMON

module FACE {

 //! This enumeration defines the possible set of status codes which may be
 //! returned by a method defined in the FACE API.
 //! The first set of codes (through TIMED_OUT) is constrained to match the
 //! set defined in the ARINC 653 standard.
 enum RETURN_CODE_TYPE {
  NO_ERROR,          //! request valid and operation performed
  NO_ACTION,         //! status of system unaffected by request
  NOT_AVAILABLE,     //! no message was available
  INVALID_PARAM,     //! invalid parameter specified in request
  INVALID_CONFIG,    //! parameter incompatible with configuration
  INVALID_MODE,      //! request incompatible with current mode
  TIMED_OUT,         //! the time expired before the request could be filled
  ADDR_IN_USE,       //! address currently in use
  PERMISSION_DENIED, //! no permission to send or connecting to wrong
                     //! partition
  MESSAGE_STALE,     //! current time - timestamp exceeds configured limits
  IN_PROGRESS,       //! asynchronous connection in progress
  CONNECTION_CLOSED, //! connection was closed
  DATA_BUFFER_TOO_SMALL, //! Data Buffer was too small for message
  DATA_OVERFLOW      //! A loss of messages due to data buffer overflow
 };

 //! This type is used to represent 64-bit signed integer with a
 //! 1 nanosecond resolution.
 typedef long long SYSTEM_TIME_TYPE;

 //! This type is used to represent an infinitely long time value.
 //! It is often used to specify that the caller is willing to wait
 //! forever for an operation to complete and does not wish to timeout.
 const SYSTEM_TIME_TYPE INF_TIME_VALUE = -1;
```

```
 //! This type is used to represent an unbounded string of characters
 typedef string UNBOUNDED_STRING_TYPE;

 //! This type is used to represent a bounded string of characters.
 typedef string<256> STRING_TYPE;

 //! This string is used to specify the location of the configuration
 //! resource in both the IOSS and TSS initialize functions. This may be local,
 //! a file name reference, or a reference to a configuration service.
 typedef STRING_TYPE CONFIGURATION_RESOURCE;

 //! This type is used to represent a system address.
 native SYSTEM_ADDRESS_TYPE;

 //! This type has a one nanosecond resolution.
 typedef SYSTEM_TIME_TYPE TIMEOUT_TYPE;

 //! This type is used to represent Globally Unique Identifiers.
 typedef long long GUID_TYPE;

};

#endif // __FACE_COMMON
```

C I/O Services Interface

C.1 Introduction

The IOS Interface includes nine supported I/O bus architectures. An I/O Service for a supported I/O bus architecture addresses both the common declarations and the declarations specific for that bus architecture. Each I/O Service provides a normalized interface for PSSS UoCs to communicate with I/O devices of the same bus architecture.

Declarations are provided using an IDL syntax that is mapped to a Programming Language, as described in Section 4.14.

Note: The code in this document is formatted to align with the formatting constraints of the printed document.

This appendix also describes the manner in which extended I/O bus architectures declarations can be provided with an IOS UoC. Extended declarations are supported to address undefined capabilities of supported I/O bus architectures. They are also used to implement an unsupported I/O bus architecture consistent with the I/O Services Interface.

C.2 Common Declarations

This section describes data types and functions that are common to the I/O Service for each supported I/O bus architecture. The declarations are in FACE/IOSS/IOS.idl.

```
//! Source file: FACE/IOSS/IOS.idl

#ifndef __FACE_IOSS_IOS
#define __FACE_IOSS_IOS

#include <FACE/Common.idl>

module FACE {
  module IOSS {

    // The status of the I/O device bus, separate from the status of a
    // single I/O connection or of a device attached to that bus.
    //
    // NOTE: Bus-specific status constants are defined in their respective
    //   namespaces.
    typedef unsigned short BUS_STATUS_TYPE;

    // I/O parameters are used to programmatically query or change
    // properties of an I/O connection or its underlying bus after
    // initialization. This may be a necessary part of PSSS UoC logic,
    // such as auto-negotiating serial baud rate, and is fundamentally
    // different from the FACE Configuration Interface that maps to an
    // underlying and existent configuration resource.
    //
    // Each I/O Service defines the specific I/O parameters it supports
    // based on the I/O connection analogy and the I/O bus architecture.
```

```
// The declarations here are the basis of those specific definitions.

// Used to declare constants to represent supported I/O parameters.
//
// NOTE: Bus-specific parameter IDs are defined in their namespaces.
typedef unsigned short IO_PARAMETER_ID_TYPE;

// Used by I/O functions to specify the I/O connection addressed
// by the handle.
typedef long long CONNECTION_HANDLE_TYPE;

// INVALID_CONNECTION_HANDLE and IGNORED_CONNECTION_HANDLE as sentinel
//   values.
const CONNECTION_HANDLE_TYPE INVALID_CONNECTION_HANDLE = -1;
const CONNECTION_HANDLE_TYPE IGNORED_CONNECTION_HANDLE = 0;

module IO_Service_Module<struct PAYLOAD_DATA_MSG_TYPE> {
  // All declarations are in this template module so that the
  // instantiated module for each I/O Service has a fully qualified
  // type that is distinct. This improves type safety in the resulting
  // language mappings.

  // The unique name of the I/O connection. It is passed to Open(I/O)
  // and typically is used to assign values specified in the
  // configuration resource given to Initialize(I/O).
  typedef STRING_TYPE CONNECTION_NAME_TYPE;

  // Enumeration of the status condition of the I/O connection.
  enum IO_CONNECTION_STATUS_TYPE {
    NOT_OPEN,      // initial state prior to opening the connection
    CONNECTING,// transient state where attempt is being made to open
               //   the connection
    READY,     // connection ready for service; it can now accept data
               //   r/w operations
    BUSY,      // connection has congestion and cannot accept more data
    DEGRADED   // connection not fully operational; there is some kind
               //   of failure
  };

  // This status represents the health of an I/O connection and
  // availability of messages. It does not represent the status of the
  // I/O device associated with the connection.
  struct CONNECTION_STATUS_TYPE {
    SYSTEM_TIME_TYPE          last_message_time;
    boolean                   message_available;
    IO_CONNECTION_STATUS_TYPE connection_status;
  };

  // List of possible types for an I/O parameter value.
  enum IO_PARAMETER_VALUE_TYPES_TYPE {
    FACE_SHORT,
    FACE_LONG,
    FACE_LONGLONG,
    FACE_USHORT,
    FACE_ULONG,
    FACE_ULONGLONG,
    FACE_FLOAT,
    FACE_DOUBLE,
    FACE_LONGDOUBLE,
    FACE_CHAR,
    FACE_BOOLEAN,
    FACE_OCTET
  };
```

```
// The value for an I/O parameter.
//
// NOTE: Bus-specific parameter values are defined in their
//   namespaces.
union IO_PARAMETER_VALUE_TYPE switch (IO_PARAMETER_VALUE_TYPES_TYPE) {
  case FACE_SHORT:      short              short_value;
  case FACE_LONG:       long               long_value;
  case FACE_LONGLONG:   long long          longlong_value;
  case FACE_USHORT:     unsigned short     ushort_value;
  case FACE_ULONG:      unsigned long      ulong_value;
  case FACE_ULONGLONG:  unsigned long long ulonglong_value;
  case FACE_FLOAT:      float              float_value;
  case FACE_DOUBLE:     double             double_value;
  case FACE_LONGDOUBLE: long double        longdouble_value;
  case FACE_CHAR:       char               char_value;
  case FACE_BOOLEAN:    boolean            boolean_value;
  case FACE_OCTET:      octet              octet_value;
};

// Represent a single (ID,value) pair and a list of pairs.
struct IO_PARAMETER {
  IO_PARAMETER_ID_TYPE    id;
  IO_PARAMETER_VALUE_TYPE value;
};
typedef sequence<IO_PARAMETER> IO_PARAMETER_LIST;

// Used to pass configuration properties across the I/O Service
// Interface.
struct IO_PARAMETER_TRANSACTION_TYPE {
  GUID_TYPE         guid; // used to differentiate transactions
  IO_PARAMETER_LIST items;
};

// The interface is designed to operate with a local or remote
// implementation. Thus the timeout parameter on the interfaces allow
// the user call to block while the request may be remotely processed.
// A value of NO_WAIT indicates the request should be made but there
// should be no waiting for a response before returning to the caller.
// In such a case the action may not occur immediately. A value of
// WAIT_FOREVER indicates waiting indefinitely.
const TIMEOUT_TYPE NO_WAIT = 0;
const TIMEOUT_TYPE WAIT_FOREVER = -1;

// Notification events are generated when the PSSS UoC has registered
// a handler callback (see Register_Notification_Event_Handler(I/O)).

// Enumeration of the type of the notification event.
enum NOTIFICATION_EVENT_TYPE {
  DATA_READ_EVENT,                // Data is available via Read()
  CONNECTION_CONFIG_CHANGE_EVENT, // Use Get_Connection_Configuration()
                                  //   for new parameters
  CONNECTION_STATUS_CHANGE_EVENT, // Use Get_Connection_Status()
  BUS_CONFIG_CHANGE_EVENT,        // Use Get_Bus_Configuration() for
                                  //   new parameters
  BUS_STATUS_CHANGE_EVENT         // Use Get_Bus_Status()
};

// Defines the function prototype for a notification callback.
//
// NOTE: 'handle' is IGNORED_CONNECTION_HANDLE when
//   'notification_event' is a bus event.
interface IO_Callback {
```

```
  void Process_Notification_Event (
    in CONNECTION_HANDLE_TYPE  handle,
    in NOTIFICATION_EVENT_TYPE notification_event);
}; // interface IO_Callback

interface IO_Service {
  // The Initialize(I/O) function allows the PSSS UoC to provide a
  // configuration resource to use when initializing an I/O Service.
  void Initialize (
    in  CONFIGURATION_RESOURCE config_resource,
    out RETURN_CODE_TYPE       return_code);

  // The Open_Connection(I/O) function is used by the PSSS UoC to
  // create a connection to an I/O device. A unique handle for the
  // connection is returned on success, or INVALID_CONNECTION_HANDLE.
  void Open_Connection (
    in  CONNECTION_NAME_TYPE   name,
    in  TIMEOUT_TYPE           timeout,
    out CONNECTION_HANDLE_TYPE handle,
    out RETURN_CODE_TYPE       return_code);

  // The Close_Connection(I/O) function is used by the PSSS UoC to
  // close a connection and release the addressed handle.
  void Close_Connection (
    in  CONNECTION_HANDLE_TYPE handle,
    out RETURN_CODE_TYPE       return_code);

  // This structure defines the data payload format for reads.
  // received_time is the time stamp most closely associated with the
  //   last byte going into the buffer.
  // If received_time is set to IGNORE_RECEIVED_TIME, it should be
  //   ignored.
  const SYSTEM_TIME_TYPE IGNORE_RECEIVED_TIME = 0;
  struct READ_PAYLOAD_TYPE {
    SYSTEM_TIME_TYPE  received_time;
    PAYLOAD_DATA_MSG_TYPE payload;
  };

  // The Read(I/O) function allows the PSSS UoC to synchronously receive
  // (poll for) payload data. It is also called by the PSSS UoC to
  // asynchronously receive payload data when a registered notification
  // callback receives a DATA_READ_EVENT.
  void Read (
    in    CONNECTION_HANDLE_TYPE handle,
    in    TIMEOUT_TYPE           timeout,
    inout READ_PAYLOAD_TYPE      payload,
    out   RETURN_CODE_TYPE       return_code);

  // This structure defines the data payload format for writes.
  struct WRITE_PAYLOAD_TYPE {
    PAYLOAD_DATA_MSG_TYPE payload;
  };

  // The Write(I/O) operation call allows the PSSS UoC to
  // synchronously send payload data.
  void Write (
    in  CONNECTION_HANDLE_TYPE handle,
    in  TIMEOUT_TYPE           timeout,
    in  WRITE_PAYLOAD_TYPE     payload,
    out RETURN_CODE_TYPE       return_code);

  // The Configure_Connection_Parameters(I/O) function allows the
  // PSSS UoC to assign I/O parameters for a connection.
```

```
    void Configure_Connection_Parameters (
      in  CONNECTION_HANDLE_TYPE        handle,
      in  TIMEOUT_TYPE                  timeout,
      in  IO_PARAMETER_TRANSACTION_TYPE parameters,
      out RETURN_CODE_TYPE              return_code);

    // The Get_Connection_Configuration(I/O) function allows the PSSS
    // UoC to query I/O parameters for a connection.
    void Get_Connection_Configuration (
      in    CONNECTION_HANDLE_TYPE        handle,
      in    TIMEOUT_TYPE                  timeout,
      inout IO_PARAMETER_TRANSACTION_TYPE parameters,
      out   RETURN_CODE_TYPE              return_code);

    // The Configure_Bus_Parameters(I/O) function allows the PSSS UoC
    // to assign I/O parameters for the I/O bus underlying an
    // I/O connection.
    void Configure_Bus_Parameters (
      in  CONNECTION_HANDLE_TYPE        handle,
      in  TIMEOUT_TYPE                  timeout,
      in  IO_PARAMETER_TRANSACTION_TYPE parameters,
      out RETURN_CODE_TYPE              return_code);

    // The Get_Bus_Configuration(I/O) function allows the PSSS UoC to
    // query I/O parameters for the I/O bus underlying an I/O connection.
    void Get_Bus_Configuration (
      in    CONNECTION_HANDLE_TYPE        handle,
      in    TIMEOUT_TYPE                  timeout,
      inout IO_PARAMETER_TRANSACTION_TYPE parameters,
      out   RETURN_CODE_TYPE              return_code);

    // The Get_Connection_Status(I/O) function allows the PSSS UoC to
    // query for connection status information.
    void Get_Connection_Status (
      in  CONNECTION_HANDLE_TYPE handle,
      out CONNECTION_STATUS_TYPE status,
      out RETURN_CODE_TYPE       return_code);

    // The Get_Bus_Status(I/O) function allows the PSSS UoC to query
    // for status information of the bus underlying an I/O connection.
    void Get_Bus_Status (
      in  CONNECTION_HANDLE_TYPE handle,
      out BUS_STATUS_TYPE        status,
      out RETURN_CODE_TYPE       return_code);

    // The Register_Notification_Event(I/O) function is used by
    // the PSSS UoC to register a callback function that is called by
    // the I/O Service when an event occurs on the given connection.
    //! Note: The io_callback parameter is semantically an in parameter
    //! but is inout to avoid an undesirable mapping in C++.
    void Register_Notification_Event (
      in     CONNECTION_HANDLE_TYPE handle,
      inout  IO_Callback            io_callback,
      out    RETURN_CODE_TYPE       return_code);

    // The Unregister_Notification_Event(I/O) function is used
    // by the PSSS UoC to unregister the callback previously associated
    // with the connection.
    void Unregister_Notification_Event (
      in  CONNECTION_HANDLE_TYPE handle,
      out RETURN_CODE_TYPE       return_code);

  }; // interface IO_Service
```

```
    }; // module IO_Service_Module<>
  }; // module IOSS
}; // module FACE

#endif // __FACE_IOS
```

C.2.1 Initialize(I/O) Function

The *Initialize*(I/O) function allows the PSSS UoC to provide a configuration resource to use when initializing an I/O Service. An I/O Service needs to be initialized before any PSSS UoC uses it.

```
module FACE {
  module IOSS {
    module IO_Service_Module<struct PAYLOAD_DATA_MSG_TYPE> {
      interface IO_Service {
        //! The Initialize(I/O) function allows the PSSS UoC to provide a
        //! configuration resource to use when initializing an I/O Service.
        void Initialize (
          in  CONFIGURATION_RESOURCE config_resource,
          out RETURN_CODE_TYPE       return_code);
      }; // interface IO_Service
    }; // module IO_Service_Module<>
  }; // module IOSS
}; // module FACE
```

The parameters to this function are as follows:

- *config_resource* – specifies the name of the configuration for the I/O Service
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the function executed successfully or failed for a specific reason.

The return code value returned from *Initialize*(I/O) is one of the following:

- NO_ERROR to indicate the I/O Service was successfully initialized according to the configuration data
- NO_ACTION to indicate the I/O Service has been initialized successfully previously
- NOT_AVAILABLE to indicate *config_resource* is not accessible
- INVALID_CONFIG to indicate *config_resource* has an error
- IN_PROGRESS to indicate the operation is still in progress and the I/O Service has not yet transitioned to a normal state

Note: To support minimal blocking at startup, the function may return before it transitions from an initial state to a normal state. Subsequent calls return IN_PROGRESS until it transitions out of its initial state to either its normal state or an error state. Once the transition occurs, the next call returns NO_ERROR for success or either NOT_AVAILABLE or INVALID_CONFIG dependent upon the error condition.

C.2.2 Open_Connection(I/O) Function

The *Open_Connection*(I/O) function is used by the PSSS UoC to create a connection to an I/O device. A handle unique to the specified name for the connection is returned on success, and INVALID_CONNECTION_HANDLE is returned on failure.

```
module FACE {
  module IOSS {
    module IO_Service_Module<struct PAYLOAD_DATA_MSG_TYPE> {
      interface IO_Service {
        //! The Open_Connection(I/O) function is used by the PSSS UoC to
        //! create a connection to an I/O device. A unique handle for the
        //! connection is returned on success, or
        //! INVALID_CONNECTION_HANDLE.
        void Open_Connection (
          in  CONNECTION_NAME_TYPE   name,
          in  TIMEOUT_TYPE           timeout,
          out CONNECTION_HANDLE_TYPE handle,
          out RETURN_CODE_TYPE       return_code);
      }; // interface IO_Service
    }; // module IO_Service_Module<>
  }; // module IOSS
}; // module FACE
```

The parameters to this function are as follows:

- *name* – specifies the name of the I/O connection to be opened
- *timeout* – specifies the upper limit of time the caller may be blocked when opening a connection, where NO_WAIT means return without blocking and WAIT_FOREVER means blocking until the operation completes
- *handle* – upon return, contains the value to be used on subsequent operations on this connection
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the function executed successfully or failed for a specific reason.

The return code value returned from *Open_Connection*(I/O) is one of the following:

- NO_ERROR to indicate successful completion of the operation
- INVALID_CONFIG to indicate that name is not valid, not configured, or an underlying operating system resource is not present
- TIMED_OUT to indicate that opening a connection took longer than the specified *timeout*
- INVALID_PARAM to indicate that the *timeout* parameter is out of range
- NO_ACTION to indicate that an underlying operating system API call failed

C.2.3 Close_Connection(I/O) Function

The *Close_Connection*(I/O) function releases the connection to the I/O device and erases any information being stored about the connection.

```
module FACE {
  module IOSS {
    module IO_Service_Module<struct PAYLOAD_DATA_MSG_TYPE> {
      interface IO_Service {
        //! The Close_Connection(I/O) function is used by the PSSS UoC to
        //! close a connection and release the addressed handle.
        void Close_Connection (
          in  CONNECTION_HANDLE_TYPE handle,
          out RETURN_CODE_TYPE       return_code);
      }; // interface IO_Service
    }; // module IO_Service_Module<>
  }; // module IOSS
}; // module FACE
```

The parameters to this function are as follows:

- *handle* – specifies the connection to close
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the function executed successfully or failed for a specific reason.

The return code value returned from *Close_Connection*(I/O) is one of the following:

- NO_ERROR to indicate successful completion of the operation
- INVALID_PARAM to indicate that *handle* does not refer to an existing connection
- CONNECTION_CLOSED to indicate that *handle* refers to an existing connection that is closed
- INVALID_MODE to indicate that the connection was configured in a manner that does not allow it to be closed
- NO_ACTION to indicate that an underlying operating system API call failed

C.2.4 Read(I/O) Function

The *Read*(I/O) function allows the PSSS UoC to synchronously receive (poll for) payload data. It is also called by the PSSS UoC to asynchronously receive payload data when a registered notification callback receives a DATA_READ_EVENT.

```
module FACE {
  module IOS {
    module IO_Service_Module<struct PAYLOAD_DATA_MSG_TYPE> {
      interface IO_Service {

        //! This structure defines the data payload format for reads.
        //! received_time is the time stamp most closely associated with
        //! the last byte going into the buffer. If received_time is set to
        //! IGNORE_RECEIVED_TIME, it should be ignored.
        const SYSTEM_TIME_TYPE IGNORE_RECEIVED_TIME = 0;
        struct READ_PAYLOAD_TYPE {
          SYSTEM_TIME_TYPE  received_time;
          PAYLOAD_DATA_MSG_TYPE payload;
        };

        //! The Read(I/O) function allows the PSSS UoC to synchronously
        //! receive (poll for) payload data. It is also called by the PSSS
```

```
        //! UoC to asynchronously receive payload data when a registered
        //! notification callback receives a DATA_READ_EVENT.
        void Read (
          in    CONNECTION_HANDLE_TYPE handle,
          in    TIMEOUT_TYPE           timeout,
          inout READ_PAYLOAD_TYPE      payload,
          out   RETURN_CODE_TYPE       return_code);
      }; // interface IO_Service
    }; // module IO_Service_Module<>
  }; // module IOS
}; // module FACE
```

The parameters to this function are as follows:

- *handle* – specifies the connection to read
- *timeout* – specifies the maximum length of time the caller may be blocked, where NO_WAIT means return without blocking and WAIT_FOREVER means blocking until the operation completes
- *payload* – specifies the payload data, as well as its received time
- *return_code* – upon return, contains a status code indicating success or failure

The fields of READ_PAYLOAD_TYPE are as follows:

- *received_time* – a timestamp that is assigned by the I/O Service correlated most closely to the system time the last field of *payload* was assigned before return
- *payload* – a structured representation of the payload data, defined by the parameterized type BUS_CONTEXT::PAYLOAD_TYPE. Each I/O Service declares its own PAYLOAD_TYPE that is bound to this declaration; through this mechanism the payload is not represented as a byte sequence

Upon return, the *return_code* output parameter contains a value indicating that the function executed successfully or failed for a specific reason.

The return code value returned from *Read*(I/O) is one of the following:

- NO_ERROR to indicate successful completion of the operation
- NOT_AVAILABLE to indicate there is no data and *timeout* is NO_WAIT
- TIMED_OUT to indicate the operation took longer than the specified *timeout*
- DATA_OVERFLOW to indicate the underlying read buffer has dropped received messages
- INVALID_MODE to indicate that the connection is configured in a manner that does not allow reading
- INVALID_PARAM to indicate that *handle* does not refer to an existing connection
- CONNECTION_CLOSED to indicate that *handle* refers to an existing connection that is closed
- INVALID_PARAM to indicate that the *timeout* parameter is out of range

- NO_ACTION to indicate that an underlying operating system API call failed

C.2.5 Write(I/O) Function

The *Write*(I/O) function allows the PSSS UoC to synchronously send payload data.

```
module FACE {
  module IOS {
    module IO_Service_Module<struct PAYLOAD_DATA_MSG_TYPE> {
      interface IO_Service {
        //! This structure defines the data payload format for writes.
        struct WRITE_PAYLOAD_TYPE {
          PAYLOAD_DATA_MSG_TYPE payload;
        };

        //! The Write(I/O) operation call allows the PSSS UoC to
        //! synchronously send payload data.
        void Write (
          in  CONNECTION_HANDLE_TYPE handle,
          in  TIMEOUT_TYPE           timeout,
          in  WRITE_PAYLOAD_TYPE     payload,
          out RETURN_CODE_TYPE       return_code);
      }; // interface IO_Service
    }; // module IO_Service_Module<>
  }; // module IOS
}; // module FACE
```

The parameters to this function are as follows:

- *handle* – specifies the connection to read
- *timeout* – specifies the maximum length of time the caller may be blocked, where NO_WAIT means return without blocking and WAIT_FOREVER means blocking until the operation completes
- *payload* – specifies the payload data, as well as its logical destination and received time
- *return_code* – upon return, contains a status code indicating success or failure

The fields of WRITE_PAYLOAD_TYPE are as follows:

- *payload* – a structured representation of the payload data, defined by the parameterized type BUS_CONTEXT::PAYLOAD_TYPE. Each I/O Service declares its own PAYLOAD_TYPE that is bound to this declaration; through this mechanism the payload is not represented as a byte sequence

The *payload.message_length* bytes of memory specified by *payload.data_buffer_address* consists of all data in the I/O Service Message Instance.

The *payload.timestamp* is recorded for use in the status call when required.

Upon return, the *return_code* output parameter contains a value indicating that the function executed successfully or failed for a specific reason.

The return code value returned from *Write*(I/O) is one of the following:

- NO_ERROR to indicate successful completion of the operation
- TIMED_OUT to indicate the operation took longer than the specified *timeout*

- DATA_OVERFLOW to indicate the underlying write buffer is full and *payload* is not sent
- INVALID_MODE to indicate that the connection is configured in a manner that does not allow writing
- INVALID_PARAM to indicate that *handle* does not refer to an existing connection
- CONNECTION_CLOSED to indicate that *handle* refers to an existing connection that is closed
- INVALID_PARAM to indicate that the *timeout* parameter is out of range
- NO_ACTION to indicate that an underlying operating system API call failed

C.2.6 Configure_Connection_Parameters(I/O) Function

The *Configure_Connection_Parameters*(I/O) function allows the PSSS UoC to assign configuration properties for a connection. The PSSS UoC can specify a subset of the defined I/O parameters. I/O parameters can be reconfigured by a subsequent call.

The I/O Service processes all I/O parameters as one transaction that succeeds or fails. If the transaction fails, the I/O parameter values are unchanged upon return.

If the operation results in a connection configuration change, then the I/O Service dispatches a CONNECTION_CONFIG_CHANGE_EVENT to each PSSS UoC with a registered notification callback on that connection.

If the operation results in a connection status change, then the I/O Service dispatches a CONNECTION_STATUS_CHANGE_EVENT to each PSSS UoC with a registered notification callback on that connection.

```
module FACE {
  module IOSS {
    module IO_Service_Module<struct PAYLOAD_DATA_MSG_TYPE> {
      interface IO_Service {
        //! The Configure_Connection_Parameters(I/O) function allows the
        //! PSSS UoC to assign I/O parameters for a connection.
        void Configure_Connection_Parameters (
          in  CONNECTION_HANDLE_TYPE        handle,
          in  TIMEOUT_TYPE                  timeout,
          in  IO_PARAMETER_TRANSACTION_TYPE parameters,
          out RETURN_CODE_TYPE              return_code);
      }; // interface IO_Service
    }; // module IO_Service_Module<>
  }; // module IOSS
}; // module FACE
```

The parameters to this function are as follows:

- *handle* – specifies the connection to be operated upon
- *timeout* – specifies the maximum length of time the caller may be blocked, where NO_WAIT means return without blocking and WAIT_FOREVER means blocking until the operation completes
- *parameters* – specifies one or more configuration properties to be assigned

- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the function executed successfully or failed for a specific reason.

The return code value returned from *Configure_Connection_Parameters*(I/O) is one of the following:

- NO_ERROR to indicate successful completion of the operation
- TIMED_OUT to indicate the operation took longer than the specified *timeout*
- INVALID_PARAM to indicate that *handle* does not refer to an existing connection
- CONNECTION_CLOSED to indicate that *handle* refers to an existing connection that is closed
- INVALID_PARAM to indicate that the *timeout* parameter is out of range
- INVALID_PARAM to indicate that an I/O parameter ID in *parameters* is unknown
- NO_ACTION to indicate that an underlying operating system API call failed
- INVALID_ CONFIG to indicate that an I/O parameter value in parameters is invalid

C.2.7 Get_Connection_Configuration(I/O) Function

The *Get_Connection_Configuration*(I/O) function allows the PSSS UoC to query a connection for configuration information. The PSSS UoC can specify a subset of the defined I/O parameters.

The I/O Service supports this operation after the connection is closed.

```
module FACE {
  module IOSS {
    module IO_Service_Module<struct PAYLOAD_DATA_MSG_TYPE> {
      interface IO_Service {
        //! The Get_Connection_Configuration(I/O) function allows the PSSS
        //! UoC to query I/O parameters for a connection.
        void Get_Connection_Configuration (
          in    CONNECTION_HANDLE_TYPE        handle,
          in    TIMEOUT_TYPE                  timeout,
          inout IO_PARAMETER_TRANSACTION_TYPE parameters,
          out   RETURN_CODE_TYPE              return_code);
      }; // interface IO_Service
    }; // module IO_Service_Module<>
  }; // module IOSS
}; // module FACE
```

The parameters to this function are as follows:

- *handle* – specifies the connection to be operated upon
- *timeout* – specifies the maximum length of time the caller may be blocked, where NO_WAIT means return without blocking and WAIT_FOREVER means blocking until the operation completes
- *parameters* – *u*pon return, contains the current value of specified I/O properties

- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the function executed successfully or failed for a specific reason.

The return code value returned from *Get_Connection_Configuration*(I/O) is one of the following:

- NO_ERROR to indicate successful completion of the operation
- TIMED_OUT to indicate the operation took longer than the specified *timeout*
- INVALID_PARAM to indicate that *handle* does not refer to an existing connection
- INVALID_PARAM to indicate that the *timeout* parameter is out of range
- INVALID_PARAM to indicate that an I/O parameter ID in *parameters* is unknown
- NO_ACTION to indicate that an underlying operating system API call failed

C.2.8 Configure_Bus_Parameters(I/O) Function

The *Configure_Bus_Parameters*(I/O) function allows the PSSS UoC to assign configuration properties for an I/O bus, given an I/O connection associated with that bus. This technique for selecting an I/O bus maintains consistency with other functions of the I/O Services Interface, supports multiple instances of the same I/O bus, and avoids using a sentinel connection ID value. Configuring the bus may impact open connections on that bus. The PSSS UoC can specify a subset of the defined I/O parameters. I/O parameters can be reconfigured by a subsequent call.

The I/O Service processes all I/O parameters as one transaction that succeeds or fails. If the transaction fails, the I/O parameter values are unchanged upon return.

The I/O Service supports this operation when called with a closed I/O connection.

If the operation results in a bus configuration change, then the I/O Service dispatches a BUS_CONFIG_CHANGE_EVENT to each PSSS UoC with a registered notification callback on an open connection of the configured bus. If the operation results in a bus status change of the configured bus, then the I/O Service dispatches a BUS_STATUS_CHANGE_EVENT to each PSSS UoC with a registered notification callback on an open connection of the configured bus.

If the operation results in a bus status change of the configured bus, then the I/O Service dispatches a BUS_STATUS_CHANGE_EVENT to each PSSS UoC with a registered notification callback on an open connection of the configured bus.

If the operation results in a connection parameter change on an open connection of the configured bus, then the I/O Service dispatches a CONNECTION_CONFIG_CHANGE_EVENT to each PSSS UoC with a registered notification callback on that connection.

If the operation results in a connection status change on an open connection of the configured bus, then the I/O Service dispatches a CONNECTION_STATUS_CHANGE_EVENT to each PSSS UoC with a registered notification callback on that connection.

```
module FACE {
  module IOSS {
    module IO_Service_Module<struct PAYLOAD_DATA_MSG_TYPE> {
```

```
      interface IO_Service {
        //! The Configure_Bus_Parameters(I/O) function allows the PSSS UoC
        //! to assign I/O parameters for the I/O bus underlying an
        //! I/O connection.
        void Configure_Bus_Parameters (
          in  CONNECTION_HANDLE_TYPE        handle,
          in  TIMEOUT_TYPE                  timeout,
          in  IO_PARAMETER_TRANSACTION_TYPE parameters,
          out RETURN_CODE_TYPE              return_code);
      }; // interface IO_Service
    }; // module IO_Service_Module<>
  }; // module IOSS
}; // module FACE
```

The parameters to this function are as follows:

- *handle* – specifies an I/O connection as the means to select its underlying bus
- *timeout* – specifies the maximum length of time the caller may be blocked, where NO_WAIT means return without blocking and WAIT_FOREVER means blocking until the operation completes
- *parameters* – specifies one or more configuration properties to be assigned
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the function executed successfully or failed for a specific reason.

The return code value returned from *Configure_Bus_Parameters*(I/O) is one of the following:

- NO_ERROR to indicate successful completion of the operation
- TIMED_OUT to indicate the operation took longer than the specified *timeout*
- INVALID_PARAM to indicate that *handle* does not refer to an existing connection
- INVALID_PARAM to indicate that the *timeout* parameter is out of range
- INVALID_PARAM to indicate that an I/O parameter ID in *parameters* is unknown
- NO_ACTION to indicate that an underlying operating system API call failed
- INVALID_ CONFIG to indicate that an I/O parameter value in parameters is invalid

C.2.9 Get_Bus_Configuration(I/O) Function

The *Get_Bus_Configuration*(I/O) function allows the PSSS UoC to query configuration properties for an I/O bus, given an I/O connection associated with that bus. This technique for selecting an I/O bus maintains consistency with other functions of the I/O Services Interface, supports multiple instances of the same I/O bus, and avoids using a sentinel connection ID value. The PSSS UoC can specify a subset of the defined I/O parameters.

The I/O Service supports this operation when called with a closed I/O connection.

```
module FACE {
  module IOSS {
    module IO_Service_Module<struct PAYLOAD_DATA_MSG_TYPE> {
      interface IO_Service {
```

```
        //! The Get_Bus_Configuration(I/O) function allows the PSSS UoC to
        //! query I/O parameters for the I/O bus underlying an I/O
        //! connection.
        void Get_Bus_Configuration (
          in    CONNECTION_HANDLE_TYPE        handle,
          in    TIMEOUT_TYPE                  timeout,
          inout IO_PARAMETER_TRANSACTION_TYPE parameters,
          out   RETURN_CODE_TYPE              return_code);
      }; // interface IO_Service
    }; // module IO_Service_Module<>
  }; // module IOSS
}; // module FACE
```

The parameters to this function are as follows:

- *handle* – specifies an I/O connection as the means to select its underlying bus
- *timeout* – specifies the maximum length of time the caller may be blocked, where NO_WAIT means return without blocking and WAIT_FOREVER means blocking until the operation completes
- *parameters* – *u*pon return, contains the current value of specified I/O properties
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the function executed successfully or failed for a specific reason.

The return code value returned from *Get_Bus_Configuration*(I/O) is one of the following:

- NO_ERROR to indicate successful completion of the operation
- TIMED_OUT to indicate the operation took longer than the specified *timeout*
- INVALID_PARAM to indicate that *handle* does not refer to an existing connection
- INVALID_PARAM to indicate that the *timeout* parameter is out of range
- INVALID_PARAM to indicate that an I/O parameter ID in *parameters* is unknown
- NO_ACTION to indicate that an underlying operating system API call failed

C.2.10 Get_Connection_Status(I/O) Function

The *Get_Connection_Status*(I/O) function allows the PSSS UoC to query for connection status information.

```
module FACE {
  module IOSS {
    module IO_Service_Module<struct PAYLOAD_DATA_MSG_TYPE> {
      interface IO_Service {
        //! The Get_Connection_Status(I/O) function allows the PSSS UoC to
        //! query for connection status information.
        void Get_Connection_Status (
          in  CONNECTION_HANDLE_TYPE handle,
          out CONNECTION_STATUS_TYPE status,
          out RETURN_CODE_TYPE       return_code);
      }; // interface IO_Service
    }; // module IO_Service_Module<>
  }; // module IOSS
```

```
}; // module FACE
```

The parameters to this function are as follows:

- *handle* – specifies the connection to be operated upon
- *status* – upon return, contains the status of the connection
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the function executed successfully or failed for a specific reason.

The return code value returned from *Get_Connection_Status*(I/O) is one of the following:

- NO_ERROR to indicate successful completion of the operation
- INVALID_PARAM to indicate that *handle* does not refer to an existing connection
- CONNECTION_CLOSED to indicate that *handle* refers to an existing connection that is closed
- NO_ACTION to indicate that an underlying operating system API call failed

C.2.11 Get_Bus_Status(I/O) Function

The *Get_Bus_Status*(I/O) function allows the PSSS UoC to query configuration properties for an I/O bus, given an I/O connection associated with that bus. This technique for selecting an I/O bus maintains consistency with other functions of the I/O Services Interface, supports multiple instances of the same I/O bus, and avoids using a sentinel connection ID value.

The I/O Service supports this operation when called with a closed I/O connection.

```
module FACE {
  module IOSS {
    module IO_Service_Module<struct PAYLOAD_DATA_MSG_TYPE> {
      interface IO_Service {
        //! The Get_Bus_Status(I/O) function allows the PSSS UoC to query
        //! for status information of the bus underlying an I/O connection.
        void Get_Bus_Status (
          in  CONNECTION_HANDLE_TYPE handle,
          out BUS_STATUS_TYPE        status,
          out RETURN_CODE_TYPE       return_code);
      }; // interface IO_Service
    }; // module IO_Service_Module<>
  }; // module IOSS
}; // module FACE
```

The parameters to this function are as follows:

- *handle* – specifies an I/O connection as the means to select its underlying bus
- *status* – upon return, contains the status of the bus; each I/O Service declares its own bus status constants
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the function executed successfully or failed for a specific reason.

The return code value returned from *Get_Bus_Status*(I/O) is one of the following:

- NO_ERROR to indicate successful completion of the operation
- INVALID_PARAM to indicate that *handle* does not refer to an existing connection
- NO_ACTION to indicate that an underlying operating system API call failed

C.2.12 Register_Notification_Event(I/O) Function

The *Register_Notification_Event*(I/O) function can be used by the PSSS UoC to register a callback function that is called by the I/O Service when an event occurs on the given connection. Notification events can be used to implement asynchronous read operations. They also can notify the PSSS UoC that the configuration or status has changed for the I/O connection or its underlying I/O bus.

A PSSS UoC is not required to process notification events of any type, and in such case would not call *Register_Notification_Event*(I/O).

A PSSS UoC is not required to process every notification event type, and in such case would implement callback behavior for the events of interest and allow the other events to fall through.

The I/O Service maintains only one notification callback per handle. A subsequent call to *Register_Notification_Event*(I/O) replaces the registered callback function. The I/O Service invokes the same registered callback function for every event on the connection as long as the connection is open or until *Unregister_Notification_Event*(I/O) is called.

```
module FACE {
  module IOSS {
    module IO_Service_Module<struct PAYLOAD_DATA_MSG_TYPE> {
      interface IO_Service {
        //! The Register_Notification_Event(I/O) function is used by
        //! the PSSS UoC to register a callback function that is called by
        //! the I/O Service when an event occurs on the given connection.
        //!
        //! Note: The io_callback parameter is semantically an in parameter
        //!       but is inout to avoid an undesirable mapping in C++.
        void Register_Notification_Event (
          in     CONNECTION_HANDLE_TYPE handle,
          inout  IO_Callback            io_callback,
          out    RETURN_CODE_TYPE       return_code);
      }; // interface IO_Service
    }; // module IO_Service_Module<>
  }; // module IOSS
}; // module FACE
```

The parameters to this function are as follows:

- *handle* – specifies the connection to be operated upon
- *io_callback* – specifies the interface to be registered
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the function executed successfully or failed for a specific reason.

The return code value returned from *Register_Notification_Event*(I/O) is one of the following:

- NO_ERROR to indicate successful completion of the operation
- NO_ACTION to indicate that the I/O Service does not support event notification for *handle*
- INVALID_PARAM to indicate that *handle* does not refer to an existing connection
- CONNECTION_CLOSED to indicate that *handle* refers to an existing connection that is closed

C.2.13 Unregister_Notification_Event(I/O) Function

The *Unregister_Notification_Event*(I/O) function is used by the PSSS UoC to unregister the callback previously associated with the connection.

The I/O Service supports this operation when called with a closed I/O connection.

```
module FACE {
  module IOSS {
    module IO_Service_Module<struct PAYLOAD_DATA_MSG_TYPE> {
      interface IO_Service {
        //! The Unregister_Notification_Event(I/O) function is used
        //! by the PSSS UoC to unregister the callback previously
        //! associated with the connection.
        void Unregister_Notification_Event (
          in  CONNECTION_HANDLE_TYPE handle,
          out RETURN_CODE_TYPE       return_code);
      }; // interface IO_Service
    }; // module IO_Service_Module<>
  }; // module IOSS
}; // module FACE
```

The parameters to this function are as follows:

- *handle* – specifies the connection to be operated upon
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the function executed successfully or failed for a specific reason.

The return code value returned from *Unregister_Notification_Event*(I/O) is one of the following:

- NO_ERROR to indicate successful completion of the operation
- INVALID_PARAM to indicate that *handle* does not refer to an existing connection
- NO_ACTION to indicate that there was not a handler registered to this connection

C.3 Supported I/O Bus Architecture Declarations

C.3.1 Generic I/O Service Declarations

FACE/IOSS/Generic.idl

```
//! Source file: FACE/IOSS/Generic.idl

#ifndef __FACE_IOSS_GENERIC
#define __FACE_IOSS_GENERIC

#include <FACE/IOSS/IOS.idl>

module FACE {
  module IOSS {

    //! Declarations for the Generic I/O Service.
    module Generic {
      //! Declarations for the buffer that becomes part of the 'payload'
      //! parameter for the IO_Service::Read and IO_Service::Write
      //! operations.
      const unsigned short MAX_BYTE_COUNT = 65535;
      typedef sequence<octet, MAX_BYTE_COUNT> BYTE_SEQUENCE;
      struct ReadWriteBuffer {
          BYTE_SEQUENCE data;
      };

      //! Declarations for the defined configuration parameters of the
      //! I/O Service. For each ID_PARAMETER_ID_TYPE, there is a comment
      //! for the expected corresponding ID_PARAMETER_VALUE_TYPE.
      const IO_PARAMETER_ID_TYPE ADDRESS = 0; // FACE_ULONGLONG
      //! Minimum value
      const IO_PARAMETER_ID_TYPE MIN_VALUE = 1; // FACE_ULONGLONG
      //! Maximum value
      const IO_PARAMETER_ID_TYPE MAX_VALUE = 2; // FACE_ULONGLONG
      //! Initial value
      const IO_PARAMETER_ID_TYPE INIT_VALUE = 3; // FACE_ULONGLONG
      //! Data buffer precision
      const IO_PARAMETER_ID_TYPE PRECISION = 4; // FACE_LONGDOUBLE

      //! Declarations for the defined bus status types for the I/O
      //! Service.
      //!
      //! Note there are no defined bus status types for the Generic
      //! I/O Service.
    };

    //! Instantiate the template module into the namespace for the I/O
    //! Service. This results in fully-qualified types in that namespace
    //! distinct to the I/O Service.
    module IO_Service_Module<Generic::ReadWriteBuffer> Generic;
  };
};

#endif //! __FACE_IOSS_GENERIC
```

C.3.2 Analog I/O Service Declarations

FACE/IOSS/Analog.idl

```
//! Source file: FACE/IOSS/Analog.idl

#ifndef __FACE_IOSS_ANALOG
#define __FACE_IOSS_ANALOG

#include <FACE/IOSS/IOS.idl>

module FACE {
  module IOSS {

    //! Declarations for the Analog I/O Service.
    module Analog {
      //! Declarations for the buffer that becomes part of the 'payload'
      //! parameter for the IO_Service::Read and IO_Service::Write
      //! operations.
      struct ReadWriteBuffer {
        long data;
      };

      //! Declarations for the defined configuration parameters of the
      //! I/O Service. For each ID_PARAMETER_ID_TYPE, there is a comment
      //! for the expected corresponding ID_PARAMETER_VALUE_TYPE.

      //! Minimum value
      const IO_PARAMETER_ID_TYPE MIN_VALUE = 0; // FACE_LONG
      //! Maximum value
      const IO_PARAMETER_ID_TYPE MAX_VALUE = 1; // FACE_LONG
      //! Initial value
      const IO_PARAMETER_ID_TYPE INIT_VALUE = 2; // FACE_LONG
      //! Data buffer precision - value type: long double
      const IO_PARAMETER_ID_TYPE PRECISION = 3; // FACE_DOUBLE
      //! Direction - FALSE = in; TRUE = out
      const IO_PARAMETER_ID_TYPE DIRECTION = 4; // FACE_BOOLEAN
      //! Voltage gain
      const IO_PARAMETER_ID_TYPE GAIN = 5; // FACE_LONGDOUBLE
      //! Voltage offset
      const IO_PARAMETER_ID_TYPE OFFSET = 6; // FACE_LONGDOUBLE

      //! Declarations for the defined bus status types for the I/O
      //! Service.
      //!
      //! Note there are no defined bus status types for the Analog
      //! I/O Service.
    };

    //! Instantiate the template module into the namespace for the I/O
    //! Service. This results in fully-qualified types in that namespace
    //! distinct to the I/O Service.
    module IO_Service_Module<Analog::ReadWriteBuffer> Analog;
  };
};

#endif //! __FACE_IOSS_ANALOG
```

C.3.3 ARINC 429 I/O Service Declarations

FACE/IOSS/ARINC429.idl

```
//! Source file: FACE/IOSS/ARINC429.idl

#ifndef __FACE_IOSS_ARINC429
#define __FACE_IOSS_ARINC429

#include <FACE/IOSS/IOS.idl>

module FACE {
  module IOSS {

    //! Declarations for the ARCINC-429 I/O Service.
    module ARINC429 {
      //! Declarations for the buffer that becomes part of the 'payload'
      //! parameter for the IO_Service::Read and IO_Service::Write
      //! operations.
      const unsigned short MAX_NUM_LABELS = 65535;
      typedef sequence<long, MAX_NUM_LABELS> LABEL_SEQUENCE;
      struct ReadWriteBuffer {
        LABEL_SEQUENCE label_payload;
      };

      //! Declarations for the defined configuration parameters of the
      //! I/O Service. For each ID_PARAMETER_ID_TYPE, there is a comment
      //! for the expected corresponding ID_PARAMETER_VALUE_TYPE.

      typedef unsigned short PARITY_TYPE;
      const PARITY_TYPE PARITY_NONE = 0;
      const PARITY_TYPE PARITY_ODD = 1;
      const PARITY_TYPE PARITY_EVEN = 2;
      const PARITY_TYPE PARITY_MARK = 3;
      const PARITY_TYPE PARITY_SPACE = 4;

      //! Channel direction - FALSE = TX; TRUE = RX
      const IO_PARAMETER_ID_TYPE DIRECTION = 0; // FACE_BOOLEAN
      //! Channel parity (PARITY_TYPE)
      const IO_PARAMETER_ID_TYPE PARITY = 1; // FACE_USHORT
      //! Channel speed - FALSE = High; TRUE = Low
      const IO_PARAMETER_ID_TYPE CHANNEL_SPEED = 2; // FACE_BOOLEAN

      //! Declarations for the defined bus status types for the I/O
      //! Service.
      const BUS_STATUS_TYPE HW_OPERATIONAL = 0;
      const BUS_STATUS_TYPE HW_FIFO_OVERFLOW = 1;
      const BUS_STATUS_TYPE SW_CIRCULAR_BUFF_OVERFLOW = 2;
      const BUS_STATUS_TYPE HW_ADDRESS_ERROR = 3;
      const BUS_STATUS_TYPE HW_SEQUENCE_ERROR = 4;
      const BUS_STATUS_TYPE HW_PARITY_ERROR = 5;
      const BUS_STATUS_TYPE CLOCK_LOSS = 6;
      const BUS_STATUS_TYPE UNKNOWN_ERROR = 7;
    };

    //! Instantiate the template module into the namespace for the I/O
    //! Service. This results in fully-qualified types in that namespace
    //! distinct to the I/O Service.
    module IO_Service_Module<ARINC429::ReadWriteBuffer> ARINC429;
  };
};
```

```
#endif //! __FACE_IOSS_ARINC429
```

C.3.4 Discrete I/O Service Declarations

FACE/IOSS/Discrete.idl

```
//! Source file: FACE/IOSS/Discrete.idl

#ifndef __FACE_IOSS_DISCRETE
#define __FACE_IOSS_DISCRETE

#include <FACE/IOSS/IOS.idl>

module FACE {
  module IOSS {

    //! Declarations for the Discrete I/O Service.
    module Discrete {
      //! Declarations for the buffer that becomes part of the 'payload'
      //! parameter for the IO_Service::Read and IO_Service::Write
      //! operations.
      typedef unsigned short DISCRETE_STATE_TYPE;
      const DISCRETE_STATE_TYPE LOW  = 0;
      const DISCRETE_STATE_TYPE HIGH = 1;
      const DISCRETE_STATE_TYPE OPEN = 2;
      const DISCRETE_STATE_TYPE UNDETERMINED = 3;
      struct ReadWriteBuffer {
        DISCRETE_STATE_TYPE state; // state of the discrete
      };

      //! Declarations for the defined configuration parameters of the
      //! I/O Service. For each ID_PARAMETER_ID_TYPE, there is a comment
      //! for the expected corresponding ID_PARAMETER_VALUE_TYPE.

      //! Maximum number of discrete inputs
      const IO_PARAMETER_ID_TYPE MAX_INPUTS = 0; // FACE_LONGLONG
      //! Maximum number of discrete outputs
      const IO_PARAMETER_ID_TYPE MAX_OUTPUTS = 1; // FACE_LONGLONG
      //! Channel direction (in or out) - FALSE = in; TRUE = out
      const IO_PARAMETER_ID_TYPE DIRECTION = 2; // FACE_BOOLEAN
      //! Initial value
      const IO_PARAMETER_ID_TYPE INITIAL_OUTPUT_VALUE = 3; // FACE_BOOLEAN

      //! Declarations for the defined bus status types for the I/O
      //! Service.
      //!
      //! Note there are no defined bus status types for the Discrete
      //! I/O Service.
    };

    //! Instantiate the template module into the namespace for the I/O
    //! Service. This results in fully-qualified types in that namespace
    //! distinct to the I/O Service.
    module IO_Service_Module<Discrete::ReadWriteBuffer> Discrete;
  };
};

#endif //! __FACE_IOSS_DISCRETE
```

C.3.5 High Precision Synchro I/O Service Declarations

FACE/IOSS/PrecisionSynchro.idl

```
//! Source file: FACE/IOSS/PrecisionSynchro.idl

#ifndef __FACE_IOSS_PRECISIONSYNCHRO
#define __FACE_IOSS_PRECISIONSYNCHRO

#include <FACE/IOSS/IOS.idl>

module FACE {
  module IOSS {

    //! Declarations for the Precision Synchro I/O Service.
    module PrecisionSynchro {
      //! Declarations for the buffer that becomes part of the 'payload'
      //! parameter for the IO_Service::Read and IO_Service::Write
      //! operations.
      enum ANGLE_INTENT_TYPE {
        USE_ANGLE,      // move to given angle at given velocity
        DISREGARD_ANGLE // disregard given angle and simply turn at given
                        //   velocity
      };
      struct ReadWriteBuffer {
        ANGLE_INTENT_TYPE angle_intent;
        long long         angle;
        long long         velocity;
      };

      //! Declarations for the defined configuration parameters of the
      //! I/O Service. For each ID_PARAMETER_ID_TYPE, there is a comment
      //! for the expected corresponding ID_PARAMETER_VALUE_TYPE.

      //! Minimum value
      const IO_PARAMETER_ID_TYPE MIN_VALUE = 0; // FACE_LONGLONG
      //! Maximum value
      const IO_PARAMETER_ID_TYPE MAX_VALUE = 1; // FACE_LONGLONG
      //! Initial value
      const IO_PARAMETER_ID_TYPE INIT_VALUE = 2; // FACE_LONGLONG
      //! Data buffer precision
      const IO_PARAMETER_ID_TYPE PRECISION = 3; // FACE_LONGDOUBLE

      //! Declarations for the defined bus status types for the I/O
      //! Service.
      //!
      //! Note there are no defined bus status types for the
      //! PrecisionSynchro I/O Service.
    };

    //! Instantiate the template module into the namespace for the I/O
    //! Service. This results in fully-qualified types in that namespace
    //! distinct to the I/O Service.
    module IO_Service_Module<PrecisionSynchro::ReadWriteBuffer>
      PrecisionSynchro;
  };
};

#endif //! __FACE_IOSS_PRECISIONSYNCHRO
```

C.3.6 I2C I/O Service Declarations

FACE/IOSS/I2C.idl

```
//! Source file: FACE/IOSS/I2C.idl

#ifndef __FACE_IOSS_I2C
#define __FACE_IOSS_I2C

#include <FACE/IOSS/IOS.idl>

module FACE {
  module IOSS {

    //! Declarations for the I2C I/O Service.
    module I2C {
      //! Declarations for the buffer that becomes part of the 'payload'
      //! parameter for the IO_Service::Read and IO_Service::Write
      //! operations.
      typedef unsigned short SLAVE_ADDRESS_TYPE;
      typedef unsigned short SLAVE_ADDRESS_SIZE_TYPE;

      typedef SYSTEM_ADDRESS_TYPE MESSAGE_ADDR_TYPE;

      const SLAVE_ADDRESS_SIZE_TYPE SLAVE_ADDRESS_SIZE_7  = 0;
      const SLAVE_ADDRESS_SIZE_TYPE SLAVE_ADDRESS_SIZE_10 = 1;
      const SLAVE_ADDRESS_SIZE_TYPE SLAVE_ADDRESS_SIZE_16 = 2;

      //! For slave read operation, if the slave buffer address is
      //! different than the slave RX Buffer address then the slave RX
      //! Buffer address is reset.
      //!
      //! For slave write operation, if the slave buffer address is
      //! different than the slave TX Buffer address then the slave TX
      //! Buffer address is reset.
      struct MASTER_COMMAND_TYPE {
        SLAVE_ADDRESS_SIZE_TYPE slave_address_size;
        SLAVE_ADDRESS_TYPE      slave_address;
        unsigned short          message_length;
        MESSAGE_ADDR_TYPE       data_buffer_address;
      };

      //! Declarations for the defined configuration parameters of the
      //! I/O Service. For each ID_PARAMETER_ID_TYPE, there is a comment
      //! for the expected corresponding ID_PARAMETER_VALUE_TYPE.

      //! Master or slave - FALSE = slave; TRUE = master
      const IO_PARAMETER_ID_TYPE IS_MASTER = 0; // FACE_BOOLEAN
      //! Baud
      const IO_PARAMETER_ID_TYPE BAUD = 1; // FACE_LONG
      //! Slave address - value type: SLAVE_ADDRESS_TYPE
      const IO_PARAMETER_ID_TYPE MY_ADDRESS = 2; // FACE_USHORT
      //! Slave RX buffer address - value type: MESSAGE_ADDR_TYPE
      const IO_PARAMETER_ID_TYPE RX_BUFFER_ADDRESS = 3;
      //! Slave RX buffer length
      const IO_PARAMETER_ID_TYPE RX_BUFFER_LENGTH = 4; // FACE_ULONG
      //! Slave TX buffer address - value type: MESSAGE_ADDR_TYPE
      const IO_PARAMETER_ID_TYPE TX_BUFFER_ADDRESS = 5;
      //! Slave TX buffer length
      const IO_PARAMETER_ID_TYPE TX_BUFFER_LENGTH = 6; // FACE_ULONG

      //! Declarations for the defined bus status types for the I/O
```

```
      //! Service.
      const BUS_STATUS_TYPE DEVICE_OPERATIONAL = 0;
      const BUS_STATUS_TYPE OVERRUN_ERROR      = 1;
      const BUS_STATUS_TYPE PARITY_ERROR       = 2;
      const BUS_STATUS_TYPE FRAMING_ERROR      = 3;
      const BUS_STATUS_TYPE ADDRESS_ERROR      = 4;
    };

    //! Instantiate the template module into the namespace for the I/O
    //! Service. This results in fully-qualified types in that namespace
    //! distinct to the I/O Service.
    module IO_Service_Module<I2C::MASTER_COMMAND_TYPE> I2C;

    //! Extend the I2C I/O Service interface for atomic combined
    //! read/write.
    module I2C {
      enum COMMAND_KIND_TYPE {
        READ,
        WRITE
      };
      struct ATOMIC_IO_DEF_ENTRY_TYPE {
        COMMAND_KIND_TYPE   cmd;
        MASTER_COMMAND_TYPE master_command;
      };
      typedef sequence <ATOMIC_IO_DEF_ENTRY_TYPE> MASTER_COMMANDS_TYPE;

      //! The Perform_Combined_Commands function allows the PSSS UoC to
      //! request a set of write and/or read commands be performed by a
      //! master.
      interface Combined_RW_IO_Service : I2C::IO_Service {
        void Perform_Combined_Commands (
          in    CONNECTION_HANDLE_TYPE handle,
          in    TIMEOUT_TYPE           timeout,
          inout MASTER_COMMANDS_TYPE   payload,
          out   RETURN_CODE_TYPE       return_code);
      };
    };
  };
};

#endif //! __FACE_IOSS_I2C
```

C.3.7 Perform_Combined_Commands(I2C) Function

The *Perform_Combined_Commands*(I2C) function is used by the PSSS UoC to request that the I2C master perform a set of write and/or read commands as one transaction.

```
module FACE {
  module IOSS {
    module I2C {
      //! The Perform_Combined_Commands function allows the PSSS UoC to
      //! request a set of write and/or read commands be performed by a
      //! master.
      interface Combined_RW_IO_Service : I2C::IO_Service {
        void Perform_Combined_Commands (
          in    CONNECTION_HANDLE_TYPE handle,
          in    TIMEOUT_TYPE           timeout,
          inout MASTER_COMMANDS_TYPE   payload,
          out   RETURN_CODE_TYPE       return_code);
      }; // interface Combined_IO_Service
    }; // module I2C
  }; // module IOSS
}; // module FACE
```

The parameters to this function are as follows:

- *handle* – specifies the connection to be operated upon
- *timeout* – specifies the maximum length of time the caller may be blocked, where NO_WAIT means return without blocking and WAIT_FOREVER means blocking until the operation completes
- *payload* – contains the set of write and/or read commands
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the function executed successfully or failed for a specific reason.

The return code value returned from *Perform_Combined_Commands*(I2C) is one of the following:

- NO_ERROR to indicate successful completion of the operation
- NOT_AVAILABLE when *payload* commands are not transactionally completed in accordance with the I2C protocol for combined read/write commands, or for hardware failures
- TIMED_OUT to indicate the operation took longer than the specified *timeout*
- INVALID_MODE to indicate that the connection is configured in a manner that does not allow commands described by *payload*
- INVALID_PARAM to indicate that *handle* does not refer to an existing connection, or that the *timeout* parameter is out of range
- CONNECTION_CLOSED to indicate that *handle* refers to an existing connection that is closed
- NO_ACTION to indicate that an underlying operating system API call failed

C.3.8 MIL-STD-1553 I/O Service Declarations

FACE/IOSS/M1553.idl

```
//! Source file: FACE/IOSS/M1553.idl

#ifndef __FACE_IOSS_M1553
#define __FACE_IOSS_M1553

#include <FACE/IOSS/IOS.idl>

module FACE {
  module IOSS {

    //! Declarations for the MIL-STD-1553 I/O Service.
    module M1553 {
      //! Declarations for the buffer that becomes part of the 'payload'
      //! parameter for the IO_Service::Read and IO_Service::Write
      //! operations.
      enum TRANSMISSION_TYPE {
        TRANSMIT,
```

```
  RECEIVE
};

typedef unsigned short WORD_TYPE;
const WORD_TYPE CMD = 0;
const WORD_TYPE STATUS = 1;
const WORD_TYPE DATA = 2;
const octet MAX_WORD_COUNT = 31;
typedef sequence<WORD_TYPE, MAX_WORD_COUNT> DATA_BUFFER_TYPE;

struct ReadWriteBuffer {
  octet bus_id;          // identify the MIL-STD-1553 bus
                         //   (valid values: 0 - 31)
  octet rt_number_1;     // remote Terminal Address
                         //   (valid values: 0 - 31)
  octet sa_number_1;     // sub Address Number
                         //   (valid values: 0 - 31)
  octet rt_number_2;     // remote Terminal Address used for RT-to-RT
                         //   (value values: 0 - 31)
  octet sa_number_2;     // sub Address Number used for RT-to-RT
                         //   (value values: 0 - 31)
  TRANSMISSION_TYPE t_r;
  DATA_BUFFER_TYPE  data;
};

//! Declarations for the defined configuration parameters of the
//! I/O Service. For each ID_PARAMETER_ID_TYPE, there is a comment
//! for the expected corresponding ID_PARAMETER_VALUE_TYPE.

typedef unsigned short CHANNEL_MODE_TYPE;
const CHANNEL_MODE_TYPE BC  = 0; // Bus Controller
const CHANNEL_MODE_TYPE BBC = 1; // Backup Bus Controller
const CHANNEL_MODE_TYPE RT  = 2; // Remote Terminal
const CHANNEL_MODE_TYPE BM  = 3; // Bus Monitor

//! Channel number; each channel may consist of one or more
//! redundant buses
const IO_PARAMETER_ID_TYPE CHANNEL_NUM = 0; // FACE_LONGLONG
//! Channel mode (CHANNEL_MODE_TYPE)
const IO_PARAMETER_ID_TYPE CHANNEL_MODE = 1; // FACE_USHORT
//! Release bus control state
const IO_PARAMETER_ID_TYPE
  RELEASE_BUS_CONTROL_STATE = 2; // FACE_BOOLEAN
//! Configured terminal address
const IO_PARAMETER_ID_TYPE
  CONFIGURED_TERMINAL_ADDRESS = 3; // FACE_ULONG

//! Declarations for the defined bus status types for the I/O
//! Service.
const BUS_STATUS_TYPE BC_IO_NO_RESPONSE = 0;
const BUS_STATUS_TYPE BC_IO_LOOP_TEST_FAIL = 1;
const BUS_STATUS_TYPE BC_IO_MSG_RETRIED = 2;
const BUS_STATUS_TYPE BC_IO_BAD_DATA_BLOCK = 3;
const BUS_STATUS_TYPE BC_IO_ADDRESS_ERROR = 4;
const BUS_STATUS_TYPE BC_IO_WORD_COUNT_ERROR = 5;
const BUS_STATUS_TYPE BC_IO_SYNC_ERROR = 6;
const BUS_STATUS_TYPE BC_IO_INVALID_WORD = 7;
const BUS_STATUS_TYPE RT_IO_TERMINAL_FLAG = 8;
const BUS_STATUS_TYPE RT_IO_SUBSYSTEM_FLAG = 9;
const BUS_STATUS_TYPE RT_IO_SERVICE_REQUEST = 10;
const BUS_STATUS_TYPE RT_IO_BUSY = 11;
const BUS_STATUS_TYPE RT_IO_DYNAMIC_BC = 12;
const BUS_STATUS_TYPE RT_IO_NO_RESPONSE = 13;
```

```
      const BUS_STATUS_TYPE RT_IO_LOOP_TEST_FAIL = 14;
      const BUS_STATUS_TYPE RT_IO_ILLEGAL_COMMAND_WORD = 15;
      const BUS_STATUS_TYPE RT_IO_WORD_COUNT_ERROR = 16;
      const BUS_STATUS_TYPE RT_IO_SYNC_ERROR = 17;
      const BUS_STATUS_TYPE RT_IO_INVALID_WORD = 18;
      const BUS_STATUS_TYPE RT_IO_RT_RT_GAP_SYNC_ADDR_ERROR = 19;
      const BUS_STATUS_TYPE RT_IO_RT_RT_2ND_CMD_ERROR = 20;
      const BUS_STATUS_TYPE RT_IO_COMMAND_WORD_ERROR = 21;
      const BUS_STATUS_TYPE BM_IO_NO_RESPONSE = 22;
      const BUS_STATUS_TYPE BM_IO_WORD_COUNT_ERROR = 23;
      const BUS_STATUS_TYPE BM_IO_SYNC_ERROR = 24;
      const BUS_STATUS_TYPE BM_IO_INVALID_WORD = 25;
      const BUS_STATUS_TYPE BM_IO_RT_RT_GAP_SYNC_ADDR_ERROR = 26;
      const BUS_STATUS_TYPE BM_IO_RT_RT_2ND_CMD_ERROR = 27;
      const BUS_STATUS_TYPE BM_IO_COMMAND_WORD_ERROR = 28;
      const BUS_STATUS_TYPE BM_IO_BAD_DATA_BLOCK = 29;
      const BUS_STATUS_TYPE BM_IO_MESSAGE_ERROR = 30;
      const BUS_STATUS_TYPE BM_IO_INSTRUMENTATION = 31;
      const BUS_STATUS_TYPE BM_IO_SERVICE_REQUEST = 32;
      const BUS_STATUS_TYPE BM_IO_RESERVED_BITS = 33;
      const BUS_STATUS_TYPE BM_IO_BROADCAST_RCVD = 34;
      const BUS_STATUS_TYPE BM_IO_BUSY = 35;
      const BUS_STATUS_TYPE BM_IO_SF = 36;
      const BUS_STATUS_TYPE BM_IO_DYNAMIC_BC = 37;
      const BUS_STATUS_TYPE BM_IO_TF = 38;
      const BUS_STATUS_TYPE DRIVER_READY = 39;
      const BUS_STATUS_TYPE DRIVER_ERROR = 40;
      const BUS_STATUS_TYPE UNKNOWN_ERROR = 41;
      const BUS_STATUS_TYPE RX_SUCCESS = 42;
      const BUS_STATUS_TYPE TX_SUCCESS = 43;
      const BUS_STATUS_TYPE RXMODE_SUCCESS = 44;
      const BUS_STATUS_TYPE TXMODE_SUCCESS = 45;
      const BUS_STATUS_TYPE RT_TO_RT_SUCCESS = 46;
      const BUS_STATUS_TYPE BC_IO_GO = 47;
      const BUS_STATUS_TYPE BC_IO_NOGO_A = 48;
      const BUS_STATUS_TYPE BC_IO_NOGO_B = 49;
      const BUS_STATUS_TYPE BC_IO_NOGO_T = 50;
    };

    //! Instantiate the template module into the namespace for the I/O
    //! Service. This results in fully-qualified types in that namespace
    //! distinct to the I/O Service.
    module IO_Service_Module<M1553::ReadWriteBuffer> M1553;
  };
};

#endif //! __FACE_IOSS_M1553
```

C.3.9 Serial I/O Service Declarations

FACE/IOSS/Serial.idl

```
//! Source file: FACE/IOSS/Serial.idl

#ifndef __FACE_IOSS_SERIAL
#define __FACE_IOSS_SERIAL

#include <FACE/IOSS/IOS.idl>

module FACE {
  module IOSS {
```

```
//! Declarations for the Serial I/O Service.
module Serial {
  //! Declarations for the buffer that becomes part of the 'payload'
  //! parameter for the IO_Service::Read and IO_Service::Write
  //! operations.
  const unsigned short MAX_BYTE_COUNT = 65535;
  typedef sequence<octet, MAX_BYTE_COUNT> DATA_BUFFER_TYPE;
  struct ReadWriteBuffer {
    octet            channel; // channel on which the message
                              // is transmitted or received
    DATA_BUFFER_TYPE data;    // serial data
  };

  //! Declarations for the defined configuration parameters of the
  //! I/O Service. For each ID_PARAMETER_ID_TYPE, there is a comment
  //! for the expected corresponding ID_PARAMETER_VALUE_TYPE.

  typedef short OPERATIONAL_MODE_TYPE;
  const OPERATIONAL_MODE_TYPE RS_232 = 0;
  const OPERATIONAL_MODE_TYPE RS_422 = 1;
  const OPERATIONAL_MODE_TYPE RS_485 = 2;

  typedef unsigned short FLOW_CONTROL_TYPE;
  const FLOW_CONTROL_TYPE NONE = 0;
  const FLOW_CONTROL_TYPE XON_XOFF = 1;
  const FLOW_CONTROL_TYPE RTS_CTS = 2;
  const FLOW_CONTROL_TYPE DSR_DTR = 3;

  typedef unsigned short PARITY_TYPE;
  const PARITY_TYPE PARITY_NONE = 0;
  const PARITY_TYPE PARITY_ODD = 1;
  const PARITY_TYPE PARITY_EVEN = 2;
  const PARITY_TYPE PARITY_MARK = 3;
  const PARITY_TYPE PARITY_SPACE = 4;

  //! Physical serial port number
  const IO_PARAMETER_ID_TYPE CHANNEL_NUM = 0; // FACE_OCTET
  //! Serial port operational mode (OPERATIONAL_MODE_TYPE)
  const IO_PARAMETER_ID_TYPE MODE = 1; // FACE_SHORT
  //! Baud rate for serial port
  const IO_PARAMETER_ID_TYPE BAUD_RATE = 2; // FACE_LONG
  //! Number of data bits to be configured
  const IO_PARAMETER_ID_TYPE DATA_BITS = 3; // FACE_SHORT
  //! Number of stop bits for serial port
  const IO_PARAMETER_ID_TYPE STOP_BITS = 4; // FACE_SHORT
  //! Parity for serial port (PARITY_TYPE)
  const IO_PARAMETER_ID_TYPE PARITY = 5; // FACE_USHORT
  //! Flow control (FLOW_CONTROL_TYPE)
  const IO_PARAMETER_ID_TYPE FLOW_CONTROL = 6; // FACE_USHORT

  //! Declarations for the defined bus status types for the I/O
  //! Service.
  const BUS_STATUS_TYPE DEVICE_OPERATIONAL = 0;
  const BUS_STATUS_TYPE OVERRUN_ERROR      = 1;
  const BUS_STATUS_TYPE PARITY_ERROR       = 2;
  const BUS_STATUS_TYPE FRAMING_ERROR      = 3;
  const BUS_STATUS_TYPE BREAK_ERROR        = 4;
};

//! Instantiate the template module into the namespace for the I/O
//! Service. This results in fully-qualified types in that namespace
//! distinct to the I/O Service.
module IO_Service_Module<Serial::ReadWriteBuffer> Serial;
```

```
  };
};

#endif //! __FACE_IOSS_SERIAL
```

C.3.10 Synchro I/O Service Declarations

FACE/IOSS/Synchro.idl

```
//! Source file: FACE/IOSS/Synchro.idl

#ifndef __FACE_IOSS_SYNCHRO
#define __FACE_IOSS_SYNCHRO

#include <FACE/IOSS/IOS.idl>

module FACE {
  module IOSS {

    //! Declarations for the Synchro I/O Service.
    module Synchro {
      //! Declarations for the buffer that becomes part of the 'payload'
      //! parameter for the IO_Service::Read and IO_Service::Write
      //! operations.
      enum ANGLE_INTENT_TYPE {
        USE_ANGLE,      // move to given angle at given velocity
        DISREGARD_ANGLE // disregard given angle and simply turn at given
                        //   velocity
      };
      struct ReadWriteBuffer {
        ANGLE_INTENT_TYPE angle_intent;
        long              angle;
        long              velocity;
      };

      //! Declarations for the defined configuration parameters of the
      //! I/O Service. For each ID_PARAMETER_ID_TYPE, there is a comment
      //! for the expected corresponding ID_PARAMETER_VALUE_TYPE.

      //! Minimum value
      const IO_PARAMETER_ID_TYPE MIN_VALUE = 0; // FACE_LONG
      //! Maximum value
      const IO_PARAMETER_ID_TYPE MAX_VALUE = 1; // FACE_LONG
      //! Initial value
      const IO_PARAMETER_ID_TYPE INIT_VALUE = 2; // FACE_LONG
      //! Data buffer precision
      const IO_PARAMETER_ID_TYPE PRECISION = 3; // FACE_LONGDOUBLE

      //! Declarations for the defined bus status types for the I/O
      //! Service.
      //!
      //! Note there are no defined bus status types for the Synchro
      //! I/O Service.
    };

    //! Instantiate the template module into the namespace for the I/O
    //! Service. This results in fully-qualified types in that namespace
    //! distinct to the I/O Service.
    module IO_Service_Module<Synchro::ReadWriteBuffer> Synchro;
  };
};
```

```
#endif //! __FACE_IOSS_SYNCHRO
```

C.3.11 ARINC 825 I/O Service Declarations

FACE/IOSS/ARINC825.idl

```
//! Source file: FACE/IOSS/ARINC825.idl

#ifndef __FACE_IOSS_ARINC825
#define __FACE_IOSS_ARINC825

#include <FACE/IOSS/IOS.idl>

module FACE {
  module IOSS {

    //! Declarations for the ARINC 825 I/O Service. It provides a
    //! portable abstraction of a CAN controller interface to a PSSS UoC.
    //! Basic support for Classical CAN and CAN FD is included. All section
    //! references are based on ARINC 825-4.

    module ARINC825 {

      //! A PSSS UoC may need to participate in intra-system, inter-system,
      //! and inter-network communications (Section 2.7). Therefore, an
      //! appropriate analogy for an ARINC 825 I/O connection is a named
      //! association to an CAN node. A PSSS UoC instance can communicate
      //! across multiple CAN buses via distinct CAN nodes by opening
      //! multiple I/O connections via Open(I/O). This analogy provides the
      //! PSSS UoC significant flexibility to compose Data Frames in order
      //! to achieve required behavior and performance.

      //! Read(I/O) and Write(I/O) are viewed to be frame-based, meaning
      //! the payload is intended to represent a Data Frame
      //! (Section 4.2.1). The following declarations lead to an
      //! abstraction of a Data Frame appropriate for a PSSS UoC.

      //! CAN Identifier (Section 5.2) is a 29-bit structure, supported
      //! here as an unsigned 32-bit integer. The PSSS UoC is responsible
      //! for assigning bit values in accordance with ARINC 825 in order
      //! to achieve its desired communication behavior. Exposing the
      //! CAN Identifier increases flexibility for the PSSS UoC but may
      //! consequentially decrease portability to other system deployments.
      typedef unsigned long CAN_IDENTIFIER_TYPE;

      //! Buffer for Data Field. The buffer length corresponds to DLC in
      //! Control Field. The Classical CAN Extended Data Frame
      //! (Section 4.2.1.1) can carry up to 8 bytes, while the CAN FD
      //! Extended Data Frame (Section 4.2.1.2) can carry up to 64 bytes.
      //! The buffer is therefore bound to the larger of the two.
      //! ARINC 825 Write(I/O) returns INVALID_PARAM when framing a
      //! Classical CAN Extended Data Frame if the buffer is longer than
      //! 8 bytes.
      const octet CAN_MAX_DLC = 8;
      const octet CAN_FD_MAX_DLC = 64;
      typedef sequence<octet,CAN_FD_MAX_DLC> CAN_BUFFER_TYPE;

      struct DataFrameAbstraction {
        //! Maps to Base Identifier and Identifier Extension of
        //! Arbitration Field.
        //! ARINC 825 Write(I/O) encodes CAN Identifier to Arbitration
        //! Field. ARINC 825 Read(I/O) decodes CAN Identifier from
```

```
  //! Arbitration Field.
  CAN_IDENTIFIER_TYPE id;

  //! SRR and IDE bits of Arbitration Field are assigned by
  //! ARINC 825, and hence are not part of the Data Frame
  //! abstraction. ARINC 825 Write(I/O) encodes SRR and IDE bits.

  //! Control Field is not part of the Data Frame abstraction.
  //! RTR bit is assigned by ARINC 825. FDF bit is determined by
  //! the CAN node characteristics, and thus is associated with the
  //! I/O connection. R0 bit is assigned by ARINC 825. DLC is related
  //! to FDF and to Data Field, and thus is derived.
  //! ARINC 825 Write(I/O) encodes Control Field.

  //! Maps to Data Field and to DLC of Control Field.
  CAN_BUFFER_TYPE buffer;

  //! CRC Field is not part of the Data Frame abstraction.
  //! ARINC 825 Write(I/O) encodes CRC Field based on FDF.
  //! ARINC 825 Read(I/O) decodes CRC Field based on FDF and triggers
  //! a DEGRADED CONNECTION_STATUS_CHANGE_EVENT on CRC error.

  //! ACK Field is not part of the Data Frame abstraction. ACK
  //! errors are handled as a fault type.
};

//! Declarations for the defined configuration parameters of the
//! I/O Service. For each IO_PARAMETER_ID_TYPE, there is a comment
//! for the expected corresponding IO_PARAMETER_VALUE_TYPE.

//! Corresponds to FDF in Control Field. This value is assigned by
//! ARINC 825 Open(I/O) based on the named connection. The PSSS UoC
//! may assume its named connection is Classical CAN or CAN FD, so
//! it can query the open connection to confirm.
//! FDF cannot be changed on an open connection, so ARINC 825
//! Configure_Connection_Parameters(I/O) returns INVALID_PARAM if
//! QUERY_FDF is part of the parameter set.
const IO_PARAMETER_ID_TYPE QUERY_FDF = 0; // FACE_BOOLEAN

//! Bit rate is a CAN bus characteristic determined at the physical
//! layer based on the CAN bus architecture. It is not configurable
//! but may be queried by the PSSS UoC. These declarations support
//! ARINC 825 Get_Bus_Configuration(I/O). Bit rate cannot be changed
//! dynamically, so ARINC 825 Configure_Bus_Parameters(I/O) returns
//! INVALID_PARAM if QUERY_BIT_RATE is part of the parameter set.
typedef octet CAN_BIT_RATE;
const CAN_BIT_RATE MAX_CAN_BIT_RATE_83_KBPS  = 0;
const CAN_BIT_RATE MAX_CAN_BIT_RATE_125_KBPS = 1;
const CAN_BIT_RATE MAX_CAN_BIT_RATE_250_KBPS = 2;
const CAN_BIT_RATE MAX_CAN_BIT_RATE_500_KBPS = 3;
const CAN_BIT_RATE MAX_CAN_BIT_RATE_1_MBPS   = 4;
const CAN_BIT_RATE MAX_CAN_BIT_RATE_2_MBPS   = 5;
const CAN_BIT_RATE MAX_CAN_BIT_RATE_4_MBPS   = 6;
const IO_PARAMETER_ID_TYPE QUERY_BIT_RATE = 1; // FACE_OCTET

//! Fault types (Section 4.5.1). These faults are detected by a CAN
//! controller, so different CAN nodes can report different fault
//! conditions. Since these are not CAN bus fault conditions, they
//! must be reportable to the PSSS UoC per I/O connection. And, since
//! Get_Connection_Status(I/O) cannot return a fault code these
//! declarations support ARINC 825 Get_Connection_Configuration(I/O).
//! ARINC 825 Read(I/O) triggers DEGRADED
//! CONNECTION_STATUS_CHANGE_EVENT on these fault conditions, and the
```

```
        //! PSSS UoC can query for fault conditions in the event handler.
        //! Fault condition cannot be assigned, so ARINC 825
        //! Configure_Connection_Parameters(I/O) returns INVALID_PARAM if
        //! QUERY_CAN_FAULT is part of the parameter set.
        typedef octet CAN_FAULT_TYPE;
        const CAN_FAULT_TYPE CAN_FAULT_NONE = 0;
        const CAN_FAULT_TYPE CAN_FAULT_BIT_ERROR = 1;
        const CAN_FAULT_TYPE CAN_FAULT_BIT_STUFFING = 2;
        const CAN_FAULT_TYPE CAN_FAULT_CRC_ERROR = 3;
        const CAN_FAULT_TYPE CAN_FAULT_FORM_ERROR = 4;
        const CAN_FAULT_TYPE CAN_FAULT_ACK_ERROR = 5;
        const IO_PARAMETER_ID_TYPE QUERY_CAN_FAULT = 2; // FACE_OCTET

        //! CAN node state machine (Section 4.6). Different CAN nodes can
        //! report different states. Since these are not CAN bus states, they
        //! must be reportable to the PSSS UoC per I/O connection. And, since
        //! Get_Connection_Status(I/O) cannot return a state value these
        //! declarations support ARINC 825 Get_Connection_Configuration(I/O).
        //! The ARINC 825 I/O Service triggers CONNECTION_STATUS_CHANGE_EVENT
        //! when the CAN node of an open I/O connection transitions state,
        //! and the PSSS UoC can query state in the event handler.
        //! The ARINC 825 I/O Service triggers NOT_OPEN when the CAN node of
        //! an open I/O connection transitions from CAN_NODE_STATE_BUS_OFF to
        //! CAN_NODE_STATE_INIT.
        //! The ARINC 825 I/O Service triggers READY when the CAN node of an
        //! open I/O connection transitions from CAN_NODE_STATE_INIT to
        //! CAN_NODE_STATE_NORMAL.
        //! The ARINC 825 I/O Service triggers DEGRADED when the CAN node of
        //! an open I/O connection transitions from CAN_NODE_STATE_NORMAL to
        //! CAN_NODE_STATE_BUS_OFF.
        typedef octet CAN_NODE_STATE_TYPE;
        const CAN_NODE_STATE_TYPE CAN_NODE_STATE_INIT    = 0;
        const CAN_NODE_STATE_TYPE CAN_NODE_STATE_NORMAL  = 1;
        const CAN_NODE_STATE_TYPE CAN_NODE_STATE_BUS_OFF = 2;
        const IO_PARAMETER_ID_TYPE QUERY_CAN_NODE_STATE = 3; // FACE_OCTET

        //! The PSSS UoC uses ARINC 825 Configure_Connection_Parameters(I/O)
        //! with CAN_NODE_RESET as the sole element of the parameter set to
        //! command the CAN node to transition from CAN_NODE_STATE_BUS_OFF to
        //! CAN_NODE_STATE_INIT. The CAN node reset cannot be queried, so
        //! ARINC 825 Get_Connection_Parameters(I/O) returns INVALID_PARAM if
        //! CAN_NODE_RESET is part of the parameter set.
        const IO_PARAMETER_ID_TYPE CAN_NODE_RESET = 4; // FACE_OCTET

    }; // module ARINC825

    //! Instantiate the template module into the namespace for the I/O
    //! Service. This results in fully-qualified types in that namespace
    //! distinct to the I/O Service.
    module IO_Service_Module<ARINC825::DataFrameAbstraction> ARINC825;

 }; // module IOSS
}; // module FACE

#endif // __FACE_IOSS_ARINC825
```

C.4 Extending I/O Bus Architecture Declarations

There are I/O Services for each of nine supported I/O bus architectures. An I/O Service includes the common declarations from Section C.2 and the corresponding specific declarations from

Section C.3. The declarations for these I/O Services are used by a PSSS UoC to utilize supported capabilities with supported I/O bus architectures.

There are two scenarios for extending these declarations to provide an I/O Service with additional capabilities. The intent for these scenarios is to allow the Software Supplier of an IOS UoC containing such an I/O Service to achieve FACE Conformance to a published standard while using extensions to that standard.

In the first scenario, additional configurable parameters and/or status values are defined for a supported I/O Service because its declarations are technically insufficient. In the second scenario, a new I/O Service is declared for an unsupported I/O bus architecture. In both scenarios, the extended declarations need to be unique from supported I/O Service declarations:

- No symbol name conflicts
- No new symbol names for the same constant integral values
- No new functions to provide the same capability of an existing function

In both scenarios, the FACE PR/CR process is used to submit a change request with the extended declarations for consideration in a future version of the FACE Technical Standard. The change request is analyzed to confirm the extended declarations do not conflict with the published standard.

D Life Cycle Management Services Interface

D.1 Introduction

This appendix specifies the Interface for the Life Cycle Management (LCM) Services. Each LCM Capability has a corresponding IDL module containing an IDL interface. As the LCM Services Capabilities are independent and optional, only the interface declarations pertaining to supported Capabilities are relevant for the providing UoC.

Declarations are provided using an IDL syntax that is mapped to a Programming Language as described in Section 4.14.

Note: The code in this document is formatted to align with the formatting constraints of the printed document.

D.2 Initializable Capability Interface

D.2.1 Initialize(LCM:: Initializable)

The *Initialize*(*LCM::Initializable*) function supports the distinct execution point of initialization. This function is an entry point into a Managed UoC instance, providing the instance with a thread of control to perform any appropriate behaviors at this execution point. It is intended to be called once transactionally, meaning called until a return code other than IN_PROGRESS is returned.

It is common in embedded systems and safety-critical systems to have a phase of execution for resource acquisition behavior, such as memory allocation, that is prohibited in later phases. LCM Services supports a two-stage resource acquisition process. The first stage, associated with the initialization execution point and hence the *Initialize*(*LCM::Initializable*) function, supports resource acquisition that does not require the Configuration Interface. The second stage, associated with the configuration execution point and hence the *Configure*(*LCM::Configurable*) function, supports leveraging the Configuration Interface when acquiring resources. Between the two execution points is when dependencies are resolved between interface users and interface providers via the Injectable Interface.

A Managed UoC with no designed behaviors at this execution point may still have this function called. As the FACE Technical Standard prescribes no particular behavior, it would be appropriate to return immediately.

```
module FACE {
  module LCM {

  //! The Initializable module corresponds to the Initializable Capability.
    module Initializable {
      interface InitializableInstance {

        //! The Initialize(LCM::Initializable) operation provides the
        //! instance an opportunity to perform appropriate behaviors at the
```

```
        //! corresponding execution point in its life-cycle.
        void Initialize(
          out RETURN_CODE_TYPE          return_code);

      }; // interface InitializableInstance
    }; // module Initializable
  }; // module LCM
}; // module FACE
```

The parameters to this function are as follows:

- *configuration* – configured parameters available at this execution point
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the function executed successfully or failed for a specific reason. The return code value is one of the following:

- NO_ERROR to indicate successful completion of the operation
- NO_ACTION to indicate that this behavior is not supported
- NOT_AVAILABLE to indicate that one or more resources could not be acquired
- IN_PROGRESS to indicate that a previous operation is still in progress
- INVALID_CONFIG to indicate an error or inconsistency in the configuration data, or that the configuration data itself is not accessible

D.2.2 Finalize(LCM:: Initializable)

The *Finalize*(*LCM:: Initializable*) function supports the distinct execution point of finalization. This function is an entry point into a Managed UoC instance, providing the instance with a thread of control to perform any appropriate behaviors at this execution point. It is intended to be called once transactionally, meaning called until a return code other than IN_PROGRESS is returned.

After this execution point the Managed UoC instance exists but may have released its resources. A Managed UoC with relevant safety-critical requirements may be designed to never release its resources, and it would be appropriate to return immediately.

A Managed UoC designed to release its resources at the destruction execution point, with no designed behaviors at the finalization execution point, may still have this function called. As the FACE Technical Standard prescribes no particular behavior, it would be appropriate to return immediately.

```
module FACE {
  module LCM {

    // The Initializable module corresponds to the Initializable Capability.
    module Initializable {
      interface InitializableInstance {

        // The Finalize(LCM::Initializable) operation provides the instance
        // an opportunity to perform appropriate behaviors at the
        // corresponding execution point in its life-cycle.
        void Finalize(
```

```
          out RETURN_CODE_TYPE          return_code);

      }; // interface InitializableInstance
    }; // module Initializable
  };// module LCM
}; // module FACE
```

The parameters to this function are as follows:

- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the function executed successfully or failed for a specific reason. The return code value is one of the following:

- NO_ERROR to indicate successful completion of the operation
- NO_ACTION to indicate that this behavior is not supported
- TIMED_OUT to indicate an error releasing one or more resources but the operation has completed
- IN_PROGRESS to indicate that a previous operation is still in progress

D.3 Configurable Capability Interface

D.3.1 Configure(LCM::Configurable)

The *Configure*(*LCM::Configurable*) function supports assignment and reassignment of the configuration parameters supported by the Managed UoC. It is intended to be called at the distinction execution point of configuration to assign the initial value for all configuration parameters. It may also be called at a later execution point in order to change one or more configuration parameters.

```
module FACE {
  module LCM {

    // The Configurable module corresponds to the Configurable Capability.
    module Configurable {
      interface ConfigurableInstance {

        // The Configure(LCM::Configurable) operation is called to provide
        // the instance with configuration parameters at the corresponding
        // execution point in its life-cycle.
        void Configure(
          in CONFIGURATION_RESOURCE     configuration,
          out RETURN_CODE_TYPE          return_code);

      }; // interface ConfigurableInstance
    }; // module Configurable
  };// module LCM
}; // module FACE
```

It is common in embedded systems and safety-critical systems to have a phase of execution for resource acquisition behavior, such as memory allocation, that is prohibited in later phases. LCM Services supports a two-stage resource acquisition process. The first stage, associated with the initialization execution point and hence the *Initialize*(*LCM::Initializable*) function, supports

resource acquisition that does not require the Configuration Interface. The second stage, associated with the configuration execution point and hence the *Configure*(*LCM::Configurable*) function, supports leveraging the Configuration Interface when acquiring resources. Between the two execution points is when dependencies are resolved between interface users and interface providers via the Injectable Interface.

The parameters to this function are as follows:

- *configuration* – configured parameters available at this execution point; this may be a partial set of the supported configuration parameters, and in such case the Managed UoC can preserve the current value of unspecified parameters
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the function executed successfully or failed for a specific reason. The return code value is one of the following:

- NO_ERROR to indicate successful completion of the operation
- NO_ACTION to indicate that this behavior is not supported
- INVALID_CONFIG to indicate an error or inconsistency in the configuration data, or that the configuration data itself is not accessible

D.4 Connectable Capability Interface

D.4.1 Framework_Connect(LCM::Connectable)

The *Framework_Connect*(*LCM::Connectable*) callback function supports the distinct execution point of Component Framework startup. Component Frameworks commonly have a startup phase where the various component instances each in turn become connected to the framework. Prior to this execution point, the component instance exists but cannot use framework services. This function is an entry point into a Managed UoC instance, providing the instance with a thread of control to perform any appropriate behaviors at this execution point, and also serving as notification that framework services are henceforth available to the Managed UoC instance. It is intended to be called once transactionally, meaning called until a return code other than IN_PROGRESS is returned, by the Component Framework.

A Managed UoC with no designed behaviors at this execution point may still have this function called by the Component Framework. As the FACE Technical Standard prescribes no particular behavior, it would be appropriate to return immediately.

```
module FACE {
  module LCM {

    // The Connectable module corresponds to the Connectable Capability.
    module Connectable {
      interface ConnectableInstance {

        // The Framework_Connect(LCM::Connectable) operation is called by a
        // Component Framework at the point during framework startup when
        // the instance is being connected. It provides the instance an
        // opportunity to perform appropriate behaviors at that point in its
        // life-cycle. The caller determines whether the operation is invoked
```

```
        // before or after the connection is completed.
        void Framework_Connect(
          in CONFIGURATION_RESOURCE     configuration,
          out RETURN_CODE_TYPE          return_code);

      }; // interface ConnectableInstance
    }; // module Connectable
  };// module LCM
}; // module FACE
```

The parameters to this function are as follows:

- *configuration* – configured parameters available at this execution point
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the function executed successfully or failed for a specific reason. The return code value is one of the following:

- NO_ERROR to indicate successful completion of the operation, or that the instance is already connected
- NO_ACTION to indicate that this behavior is not supported
- IN_PROGRESS to indicate that a previous operation is still in progress
- INVALID_CONFIG to indicate an error or inconsistency in the configuration data

D.4.2 Framework_Disconnect(LCM::Connectable)

The *Framework_Disconnect*(*LCM::Connectable*) function supports the distinct execution point of Component Framework teardown. Component Frameworks commonly have a teardown phase where the various component instances each in turn become disconnected from the framework. Following this execution point, the component instance exists but cannot use framework services. This function is an entry point into a Managed UoC instance, providing the instance with a thread of control to perform any appropriate behaviors at this execution point, and also serving as notification that framework services are henceforth unavailable to the Managed UoC instance. It is intended to be called once transactionally, meaning called until a return code other than IN_PROGRESS is returned, by the Component Framework.

A Managed UoC with no designed behaviors at this execution point may still have this function called by the Component Framework. As the FACE Technical Standard prescribes no particular behavior, it would be appropriate to return immediately.

```
module FACE {
  module LCM {

    // The Connectable module corresponds to the Connectable Capability.
    module Connectable {
      interface ConnectableInstance {

        // The Framework_Disconnect(LCM::Connectable) operation is called by
        // a Component Framework at the point during framework teardown when
        // the instance is being disconnected. It provides the instance an
        // opportunity to perform appropriate behaviors at that point in its
        // life-cycle. The caller determines whether the operation is invoked
        // before or after the disconnection is completed.
```

```
        void Framework_Disconnect(
          out RETURN_CODE_TYPE          return_code);

      }; // interface ConnectableInstance
    }; module Connectable
  };// module LCM
}; // module FACE
```

The parameters to this function are as follows:

- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the function executed successfully or failed for a specific reason. The return code value is one of the following:

- NO_ERROR to indicate successful completion of the operation, or that the instance is already disconnected
- NO_ACTION to indicate that this behavior is not supported
- IN_PROGRESS to indicate that a previous operation is still in progress

D.5 Stateful Capability Interface

D.5.1 Query_State(LCM::Stateful)

The *Query_State*(*LCM::Stateful*) function returns the current state of an instance of a Managed UoC.

```
module FACE {
  module LCM {
    // The Stateful module corresponds to the Stateful Capability.
    module Stateful<typename REQUESTED_STATE_VALUE_TYPE,
                    typename REPORTED_STATE_VALUE_TYPE> {
      interface StatefulInstance {

        // The Query_State(LCM::Stateful) operation is called to retrieve
        // the instance's current state.
        void Query_State(
          out REPORTED_STATE_VALUE_TYPE current_state,
          out RETURN_CODE_TYPE          return_code);

      }; // interface StatefulInstance
    }; // module Stateful
  };// module LCM
}; // module FACE
```

The parameters to this function are as follows:

- *current_state* – upon return, the state of the instance
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the function executed successfully or failed for a specific reason. The return code value is one of the following:

- NO_ERROR to indicate successful completion of the operation
- NO_ACTION to indicate that this behavior is not supported
- NOT_AVAILABLE to indicate that the state could not be returned because it was not in a steady state

D.5.2 Request_State_Transition(LCM::Stateful)

The *Request_State_Transition*(*LCM::Stateful*) function is a request to change the state of an instance of a Managed UoC.

```
module FACE {
  module LCM {
    // The Stateful module corresponds to the Stateful Capability.
    module Stateful<typename REQUESTED_STATE_VALUE_TYPE,
                    typename REPORTED_STATE_VALUE_TYPE> {
      interface StatefulInstance {

        // The Request_State_Transition(LCM::Stateful) operation is called
        // to request that the instance transition to 'new_state'.
        void Request_State_Transition(
          in REQUESTED_STATE_VALUE_TYPE new_state,
          out RETURN_CODE_TYPE          return_code);

      }; // interface StatefulInstance
    }; // module Stateful
  };// module LCM
}; // module FACE
```

The parameters to this function are as follows:

- *new_state* – the state to which the instance is requested to transition
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the function executed successfully or failed for a specific reason. The return code value is one of the following:

- NO_ERROR to indicate successful completion of the operation, or that the component is already in the requested state
- NO_ACTION to indicate that this behavior is not supported
- IN_PROGRESS to indicate that a previous operation is still in progress
- NOT_AVAILABLE to indicate that the requested state transition could not be executed because it was not in an appropriate state

D.6 Complete Declarations

This section provides the complete IDL declarations for each of the four LCM Services.

D.6.1 Initializable IDL Declarations

```
//! Source file: FACE/LCM/Initializable.idl
```

```
#ifndef __FACE_LCM_INITIALIZABLE
#define __FACE_LCM_INITIALIZABLE

#include <FACE/Common.idl>

module FACE {
  module LCM {

  //! FACE does not define an interface to create or destroy instances. The
  //! IDL language bindings address the syntax for those operations. The
  //! LCM Interface assumes an existent software object that implements
  //! one or more of the interfaces defined.

  //! The Initializable module corresponds to the Initializable Capability.
    module Initializable {
      interface InitializableInstance {

        //! The Initialize(LCM::Initializable) operation provides the
        //! instance an opportunity to perform appropriate behaviors at the
        //! corresponding execution point in its life-cycle.
        void Initialize(
          out RETURN_CODE_TYPE          return_code);

        //! The Finalize(LCM::Initializable) operation provides the
        //! instance an opportunity to perform appropriate behaviors at the
        //! corresponding execution point in its life-cycle.
        void Finalize(
          out RETURN_CODE_TYPE          return_code);

      }; // interface InitializableInstance
    }; // module Initializable
  }; // module LCM
}; // module FACE

#endif //! __FACE_LCM_INITIALIZABLE
```

D.6.2 Configurable IDL Declarations

```
//! Source file: FACE/LCM/Configurable.idl

#ifndef __FACE_LCM_CONFIGURABLE
#define __FACE_LCM_CONFIGURABLE

#include <FACE/Common.idl>

module FACE {
  module LCM {

  //! FACE does not define an interface to create or destroy instances. The
  //! IDL language bindings address the syntax for those operations. The
  //! LCM Interface assumes an existent software object that implements
  //! one or more of the interfaces defined.

  //! The Configurable module corresponds to the Configurable Capability.
    module Configurable {
      interface ConfigurableInstance {

        //! The Configure(LCM::Configurable) operation is called to provide
        //! the instance with configuration parameters at the corresponding
        //! execution point in its life-cycle.
        void Configure(
          in CONFIGURATION_RESOURCE     configuration,
          out RETURN_CODE_TYPE          return_code);
```

```
      }; // interface ConfigurableInstance
    }; // module Configurable
  }; // module LCM
}; // module FACE

#endif //! __FACE_LCM_CONFIGURABLE
```

D.6.3 Connectable IDL Declarations

```
//! Source file: FACE/LCM/Connectable.idl

#ifndef __FACE_LCM_CONNECTABLE
#define __FACE_LCM_CONNECTABLE

#include <FACE/Common.idl>

module FACE {
  module LCM {

  //! FACE does not define an interface to create or destroy instances. The
  //! IDL language bindings address the syntax for those operations. The
  //! LCM Interface assumes an existent software object that implements
  //! one or more of the interfaces defined.

  //! The Connectable module corresponds to the Connectable Capability.
    module Connectable {
      interface ConnectableInstance {

        //! The Framework_Connect(LCM::Connectable) operation is called by
        //! a Component Framework at the point during framework startup
        //! when the instance is being connected. It provides the instance
        //! an opportunity to perform appropriate behaviors at that point
        //! in its life-cycle. The caller determines whether the operation
        //! is invoked before or after the connection is completed.
        void Framework_Connect(
          in CONFIGURATION_RESOURCE     configuration,
          out RETURN_CODE_TYPE          return_code);

        //! The Framework_Disconnect(LCM::Connectable) operation is called
        //! by a Component Framework at the point during framework teardown
        //! when the instance is being disconnected. It provides the
        //! instance an opportunity to perform appropriate behaviors at
        //! that point in its life-cycle. The caller determines whether the
        //! operation is invoked before or after the disconnection is
        //! completed.
        void Framework_Disconnect(
          out RETURN_CODE_TYPE          return_code);

      }; // interface ConnectableInstance
    }; // module Connectable
  }; // module LCM
}; // module FACE

#endif //! __FACE_LCM_CONNECTABLE
```

D.6.4 Stateful IDL Declarations

```
//! Source file: FACE/LCM/Stateful.idl

#ifndef __FACE_LCM_STATEFUL
#define __FACE_LCM_STATEFUL
```

```
#include <FACE/Common.idl>

module FACE {
  module LCM {

  //! FACE does not define an interface to create or destroy instances. The
  //! IDL language bindings address the syntax for those operations. The
  //! LCM Interface assumes an existent software object that implements
  //! one or more of the interfaces defined.

  //! The Stateful module corresponds to the Stateful Capability.
    module Stateful<typename REQUESTED_STATE_VALUE_TYPE,
                    typename REPORTED_STATE_VALUE_TYPE> {
      interface StatefulInstance {

        //! The Query_State(LCM::Stateful) operation is called to retrieve
        //! the instance's current state.
        void Query_State(
          out REPORTED_STATE_VALUE_TYPE current_state,
          out RETURN_CODE_TYPE          return_code);

        //! The Request_State_Transition(LCM::Stateful) operation is called
        //! to request that the instance transition to 'new_state'.
        void Request_State_Transition(
          in REQUESTED_STATE_VALUE_TYPE new_state,
          out RETURN_CODE_TYPE          return_code);

      }; // interface StatefulInstance
    }; // module Stateful
  }; // module LCM
}; // module FACE

#endif //! __FACE_LCM_STATEFUL
```

E Transport Services Interfaces

E.1 Introduction

The TS Interface is defined by an abstraction interface allowing portable software components to access transport mechanisms used by the TSS library. These mechanisms include queues, sockets, sampling ports, etc. The goal of the TS Interface is to enhance portability by abstracting multiple transport mechanism interfaces from the portable software component. The TS Interface and TSS are described in Section 4.7 and Section 4.8, respectively.

Declarations are provided using an IDL syntax that is mapped to a Programming Language, as described in Section 4.14.

Note: The IDL in this document is formatted to align with the formatting constraints of the printed document.

E.2 Data Types

E.2.1 TSS Common Data Types

```
//! Source file: FACE/TSS/Common.idl

#ifndef __FACE_TSS_COMMON
#define __FACE_TSS_COMMON

#include <FACE/Common.idl>

module FACE {
  module TSS {
    //! String containing the connection name used in the TSS create_connection
    //! function.
    typedef STRING_TYPE CONNECTION_NAME_TYPE;

    //! Length of the TS Message.
    typedef long MESSAGE_SIZE_TYPE;

    //! Link to the Data Model Type Information.
    typedef GUID_TYPE MESSAGE_GUID_TYPE;

    //!  UID Type is scoped to be unique within a system rather than global
    typedef long long UID_TYPE;

    //! Unique identifier for a TSS connection obtained in create_connection
    //! but used in other TSS functions.
    typedef UID_TYPE CONNECTION_ID_TYPE;

    //! Used to tie together request/reply messages.
    typedef UID_TYPE TRANSACTION_ID_TYPE;

    //QoS Key, Value struct
    struct QoS_Element{
      STRING_TYPE keyname;
```

```
      STRING_TYPE value;
    };

    typedef sequence<QoS_Element> QoS_EVENT_TYPE;

    // "contains instance UID, source UID, and timestamp"
    struct HEADER_TYPE {
      UID_TYPE        instance_uid;
      UID_TYPE        source_uid;
      SYSTEM_TIME_TYPE timestamp;
    };

    //! This type is used to represent a size in bytes.
    typedef long BYTE_SIZE_TYPE;

    //! This type is used to represent a raw data buffer.
    struct DATA_BUFFER_TYPE {
      SYSTEM_ADDRESS_TYPE buffer_address;
      BYTE_SIZE_TYPE buffer_capacity;
    };
    struct MESSAGE_TYPE {
      MESSAGE_GUID_TYPE message_guid;
      DATA_BUFFER_TYPE  buffer;
    };

  };
};

#endif // __FACE_TSS_COMMON
```

E.3 TSS Inter-Segment Interfaces

E.3.1 Type-Specific Base Interface Specification

```
//! Source file: FACE/TSS/Base.idl

#ifndef __FACE_TSS_BASE
#define __FACE_TSS_BASE

#include <FACE/TSS/Common.idl>

module FACE {
  module TSS {
    //! Base interface provides the common TSS functions
    interface Base {
      //! The Initialize(TS) function call allows for the PCS and PSSS
      //! UoC to trigger the initialization of the TS Interface.
      void Initialize (
        in  CONFIGURATION_RESOURCE configuration,
        out RETURN_CODE_TYPE       return_code);

      //! The TSS provides an interface to create a connection. This interface
      //! allows the use of DDS, CORBA, ARINC 653, and POSIX connections.
      void Create_Connection (
        in  CONNECTION_NAME_TYPE connection_name,
        in  TIMEOUT_TYPE         timeout,
        out CONNECTION_ID_TYPE connection_id,
        out MESSAGE_SIZE_TYPE  max_message_size,
        out RETURN_CODE_TYPE   return_code);

      void Destroy_Connection (
```

```
        in  CONNECTION_ID_TYPE connection_id,
        out RETURN_CODE_TYPE   return_code);

      //! The purpose of Unregister_Callback(TS) is to unregister a callback.
      void Unregister_Callback (
        in  CONNECTION_ID_TYPE connection_id,
        out RETURN_CODE_TYPE   return_code);
    };
  };
};

#endif // __FACE_TSS_BASE
```

E.3.1.1 Initialize(TS) Function

The *Initialize*(TS) function call allows for the PCS and PSSS UoC to trigger the initialization of the TSS UoC.

```
/* IDL declaration */
module FACE {
  module TSS {
      //! The Initialize(TS) function call allows for the PCS and PSSS
      //! UoC to trigger the initialization of the TS Interface.
      void Initialize (
        in  CONFIGURATION_RESOURCE configuration,
        out RETURN_CODE_TYPE       return_code);

    };
  };
};
```

The parameters to this method are as follows:

- *configuration* – specifies the name of the configuration for the TS Interface
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Initialize*(TS) is one of the following:

- NO_ERROR to indicate TS was successfully initialized according to the configuration data
- NO_ACTION to indicate TS has been initialized successfully previously
- NOT_AVAILABLE to indicate the *configuration* data is not accessible
- INVALID_CONFIG to indicate the *configuration* data has an error
- IN_PROGRESS to indicate the *initialize* is still in progress and the TSS has not yet transitioned to a normal state

Note: To support minimal blocking at startup, the initialize may return before it transitions from an initialize state to a normal state. Subsequent calls to initialize return IN_PROGRESS until it transitions out of its initial state. Once the transition occurs, the next call to initialize returns NO_ACTION.

E.3.1.2 Create_Connection(TS) Function

The create connection call allows for a PCS and/or PSSS UoC to establish a TSS connection (TS-UoP Connection). The TSS may use underlying transports such as DDS, CORBA, ARINC 653, and/or POSIX function calls. The parameters for the transport's connections are determined through the TSS Configuration Capability.

```
/* IDL declaration */
module FACE {
  module TSS {
      //! Base interface provides the common TSS functions
      interface Base {

      //! The TSS provides an interface to create a connection. This interface
      //! allows the use of DDS, CORBA, ARINC 653, and POSIX connections.
      void Create_Connection (
        in  CONNECTION_NAME_TYPE connection_name,
        in  TIMEOUT_TYPE         timeout,
        out CONNECTION_ID_TYPE connection_id,
        out MESSAGE_SIZE_TYPE  max_message_size,
        out RETURN_CODE_TYPE   return_code);      };
  };
};
```

Note: Care should be taken when implementing the *FACE::TSS::Create_Connection*(TS) method to minimize blocking time.

The parameters to this method are as follows:

- *connection_name* – reference to a connection name in the *configuration*
- *timeout* – an upper limit on the blocking time a UoP is willing to wait for the *Create_Connection* method to return control to the UoP; *Create_Connection* can return earlier, but no later than the timeout provided
- *connection_id* – identifier for this connection which is returned by the TS Interface; the identifier is used in subsequent interactions with the TS pertaining to this connection
- *max_message_size* – returned by the TS Interface indicating the selected value of the maximum message size for this connection; for example, a PCS or PSSS UoC may need to perform special handling, such as validation of UoC buffer sizes, UoC fragmentation, and re-assembly of messages, etc.
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *create_connection* is one of the following:

- NO_ERROR to indicate the TS-UoP connection was successfully created
- NO_ACTION to indicate a failure due to unknown reasons
- NOT_AVAILABLE to indicate the TS is not yet initialized or the underlying technology is unavailable
- INVALID_PARAM to indicate one or more parameters supplied is null or not in range

- INVALID_CONFIG to indicate the configuration data does not match one or more supplied parameter
- TIMED_OUT to indicate a timeout was specified and exceeded

E.3.1.3 Destroy_Connection(TS) Function

The *Destroy_Connection*(TS) function frees up any resources allocated to the connection. This can be an empty function if no cleanup is required.

```
/* IDL declaration */
module FACE {
  module TSS {
      //! Base interface provides the common TSS functions
      interface Base {

        void Destroy_Connection (
        in  CONNECTION_ID_TYPE connection_id,
        out RETURN_CODE_TYPE   return_code);
    };
  };
};
```

The parameters to this method are as follows:

- *connection_id* – identifier for the connection to destroy
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *destroy_connection* is one of the following:

- NO_ERROR to indicate the TS-UoP connection was successfully destroyed
- NO_ACTION to indicate a failure due to unknown reasons
- NOT_AVAILABLE to indicate the TS is not yet initialized or the underlying technology is unavailable
- INVALID_PARAM to indicate *connection_id* supplied is null or not in range

E.3.1.4 Unregister_Callback(TS) Function

The purpose of *Unregister_Callback*(TS) is to provide a mechanism to unregister the callback associated with a *connection_id*.

```
/* IDL declaration */
module FACE {
  module TS {
      //! Base interface provides the common TSS functions
      interface Base {

      //! The purpose of Unregister_Callback(TS) is to unregister a callback.
      void Unregister_Callback (
        in  CONNECTION_ID_TYPE connection_id,
        out RETURN_CODE_TYPE   return_code);
    };
  };
```

```
  };
```

The parameters to this method are as follows:

- *connection_id* – identifier for the connection to unregister a callback
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Unregister_Callback* is one of the following:

- NO_ERROR to indicate the *Unregister_Callback* method was successful
- NOT_AVAILABLE to indicate the TS is not yet initialized
- INVALID_PARAM to indicate the *connection_id* supplied is null or not in range

E.3.2 Type-SpecificTyped Interface Specification

```
//! Source file: FACE/TSS/Typed.idl

#ifndef __FACE_TSS_TYPED
#define __FACE_TSS_TYPED

#include <FACE/TSS/Common.idl>

module FACE {
  module TSS {
    //! Template provides the operations for a given data type
    //! Unique modules are instantiated to accommodate UoCs in the same memory
    //! space
    module Typed<typename DATATYPE_TYPE> {

      //! The Read_Callback interface provides a callback prototype for
PCS/PSSS UoCs
      //! and is used to support receiving data without the PCS/PSSS UoC having
to poll
      //! for the data.  If a callback is used it is registered using the
      //! Register_Callback
      interface Read_Callback {
        void Callback_Handler (
           in   CONNECTION_ID_TYPE  connection_id,
           in   TRANSACTION_ID_TYPE transaction_id,
           in   DATATYPE_TYPE       message,
           in   HEADER_TYPE         header,
           in   QoS_EVENT_TYPE      qos_parameters,
           out  RETURN_CODE_TYPE    return_code);
      };

      interface TypedTS {

        //! The purpose of Receive_Message (TS) is to provide a
        //! mechanism to receive data from another source
        void Receive_Message (
           in    CONNECTION_ID_TYPE  connection_id,
           in    TIMEOUT_TYPE        timeout,
           inout TRANSACTION_ID_TYPE transaction_id,
           inout DATATYPE_TYPE       message,
           out   HEADER_TYPE         header,
```

```
            out   QoS_EVENT_TYPE      qos_parameters,
            out   RETURN_CODE_TYPE    return_code);

          //! The purpose of Send_Message (TS) is to provide a mechanism
          //! to send data to a destination
          void Send_Message (
            in    CONNECTION_ID_TYPE  connection_id,
            in    TIMEOUT_TYPE        timeout,
            inout TRANSACTION_ID_TYPE transaction_id,
            in    DATATYPE_TYPE       message,
            out   RETURN_CODE_TYPE    return_code);

          //! The purpose of Register_Callback(TS) is to provide a mechanism
          //! to read data without polling.
          void Register_Callback (
            in     CONNECTION_ID_TYPE connection_id,
            inout  Read_Callback      callback,
            out    RETURN_CODE_TYPE   return_code);
        };
      };
    };
  };

#endif // __FACE_TSS_TYPED
```

E.3.2.1 Callback_Handler(RC) Function

When registered by a PCS/PSSS UoC, the *Callback_Handler* function provided by the PCS/PSSS UoC is called by a TSS UoC upon receipt of data. Data is then provided by the TSS invoking the *Callback_Handler* function as a callback to the PCS/PSSS UoC.

```
module FACE {
  module TSS {
    module Typed<typename DATATYPE_TYPE> {
      //! The Read_Callback interface provides a callback prototype for
      //! PCS/PSSS UoCs and is used to support receiving data without the
      //! PCS/PSSS UoC having to poll for the data. If a callback is used
      //! it is registered using the Register_Callback.
      interface Read_Callback {
        void Callback_Handler (
           in   CONNECTION_ID_TYPE  connection_id,
           in   TRANSACTION_ID_TYPE transaction_id,
           in   DATATYPE_TYPE       message,
           in   HEADER_TYPE         header,
           in   QoS_EVENT_TYPE      qos_parameters,
           out  RETURN_CODE_TYPE    return_code);
      };
    };
  };
};
```

The parameters to this method are as follows:

- *connection_id* – identifier for the connection on which data was received
- *transaction_id* – identifier used to associate messages in the request/response message pattern

  Clients are provided with the *transaction_id* along with its response message when a response message from a server is received. Servers are provided with the *transaction_id*

along with a request message when a request message is received. The TSS UoC ensures the server's reply message is correctly associated with the client's request message.

- *message* – a reference to the data of interest to the PCS/PSSS UoC
- *header* – a reference to the header instance for this Message Instance
- *qos_parameters* – a reference to the QoS event values for this Message Instance
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Callback_Handler* is one of the following:

- NO_ERROR to indicate the *Callback_Handler* method was successful
- DATA_OVERFLOW to indicate the rate of received messages sent via callback exceeds the ability of the UoP to process those messages

E.3.2.2 Receive_Message(TS) Function

The *Receive_Message*(TS) function is used to receive data from another source.

```
/* IDL declaration */
module FACE {
  module TSS {
    module Typed<typename DATATYPE_TYPE> {
interface TypedTS {

        //! The purpose of Receive_Message (TS) is to provide a
        //! mechanism to receive data from another source.

        void Receive_Message (
           in    CONNECTION_ID_TYPE  connection_id,
           in    TIMEOUT_TYPE        timeout,
           inout TRANSACTION_ID_TYPE transaction_id,
           inout DATATYPE_TYPE       message,
           out   HEADER_TYPE         header,
           out   QoS_EVENT_TYPE      qos_parameters,
           out   RETURN_CODE_TYPE    return_code);
      };
    };
  };
};
```

The parameters to this method are as follows:

- *connection_id* – identifier for the connection on which to receive data
- *timeout* – an upper limit on the blocking time a UoP is willing to wait for the *Receive_Message* method to return control to the UoP; *Receive_Message* can return earlier, but no later than the timeout provided
- *transaction_id* – identifier used to associate messages in the request/response message pattern

  Clients provide the *transaction_id* to *Receive_Message* to get its response from a server. Servers are provided with the *transaction_id* when a request message is received. The

TSS UoC ensures the server's reply message is correctly associated with the client's request message.

- *message* – a reference to the data of interest to the PCS/PSSS UoC
- *header* – a reference to the header instance for this Message Instance
- *qos_parameters* – a reference to the QoS event values for this Message Instance
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Receive_Message* is one of the following:

- NO_ERROR to indicate the *Receive_Message* method was successful
- NO_ACTION to indicate a failure due to unknown reasons
- NOT_AVAILABLE to indicate the TS is not yet initialized or the connection for the underlying technology is unavailable
- INVALID_PARAM to indicate one or more parameters supplied is null or not in range or the *connection_id* does not exist
- INVALID_MODE to indicate a callback function has been registered for this connection
- TIMED_OUT to indicate a timeout was specified and exceeded
- MESSAGE_STALE to indicate the message lifespan has been exceeded
- CONNECTION_CLOSED to indicate the TS-UoP connection is not open/available
- DATA_BUFFER_TOO_SMALL to indicate the message received exceeds the message size given on a single transaction
- DATA_OVERFLOW to indicate the rate of incoming messages exceeds the rate of messages being read

E.3.2.3 Send_Message(TS) Function

The *Send_Message*(TS) function is used to send data to a destination.

```
/* IDL declaration */
module FACE {
  module TSS {
module Typed<typename DATATYPE_TYPE> {
      interface TypedTS {

        //! The purpose of Send_Message (TS) is to provide a mechanism
        //! to send data to a destination.
        void Send_Message (
          in    CONNECTION_ID_TYPE  connection_id,
          in    TIMEOUT_TYPE        timeout,
          inout TRANSACTION_ID_TYPE transaction_id,
          in    DATATYPE_TYPE       message,
          out   RETURN_CODE_TYPE    return_code);
```

```
      };
    };
  };
};
```

The parameters to this method are as follows:

- *connection_id* – identifier for the connection on which to send data
- *timeout* – an upper limit on the blocking time a UoP is willing to wait for the *Send_Message* method to return control to the UoP; *Send_Message* can return earlier, but no later than the timeout provided
- *transaction_id* – identifier used to associate messages in the request/response message pattern

  Clients are returned the *transaction_id* from *Send_Message* when sending a request message. Servers provide the *transaction_id* to *Send_Message* when sending a response message. The TSS UoC ensures the server's reply message is correctly associated with the client's request message.
- *message* – a reference to the PCS/PSSS data to send
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Send_Message* is one of the following:

- NO_ERROR to indicate the *Send_Message* method was successful
- NO_ACTION to indicate a failure due to unknown reasons
- NOT_AVAILABLE to indicate the TS is not yet initialized or the connection for the underlying technology is unavailable
- INVALID_PARAM to indicate one or more parameters supplied is null or not in range or the *connection_id* does not exist
- TIMED_OUT to indicate a timeout was specified and exceeded
- CONNECTION_CLOSED to indicate the TS-UoP connection is not open/available
- DATA_OVERFLOW to indicate the rate of messages to send exceeds the ability of the transport to send the message

E.3.2.4 Register_Callback(TS) Function

The purpose of *Register_Callback*(TS) is to provide a mechanism to read data without polling. Once the PCS/PSSS UoP registers its *Read_Callback* interface with the TS Interface, the TSS UoC invokes the *Read_Callback* method associated with the message received.

```
/* IDL declaration */
module FACE {
  module TSS {
    module Typed<typename DATATYPE_TYPE> {
      interface TypedTS {
```

```
        //! The purpose of Register_Callback(TS) is to provide a mechanism
        //! to read data without polling.
        void Register_Callback (
          in     CONNECTION_ID_TYPE connection_id,
          inout  Read_Callback      callback,
          out    RETURN_CODE_TYPE   return_code);
      };
    };
  };
};
```

The parameters to this method are as follows:

- *connection_id* – connection identifier to associate the read callback
- *callback* – a reference to a *Callback_Handler* method to handle received type-specific messages
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Register_Callback* is one of the following:

- NO_ERROR to indicate the *Register_Callback* method was successful
- NO_ACTION to indicate a callback has already been registered for the *connection_id* given
- NOT_AVAILABLE to indicate the TS is not yet initialized
- INVALID_PARAM to indicate one or more parameters supplied is null or not in range or the *connection_id* does not exist
- CONNECTION_CLOSED to indicate the TS-UoP connection is not open/available

E.3.3 Serialization Interface Specification

```
//! Source file: FACE/TSS/Serialization.idl

#ifndef __FACE_TSS_SERIALIZATION
#define __FACE_TSS_SERIALIZATION

#include <FACE/TSS/Common.idl>
#include <FACE/TSS/TPM.idl>

module FACE {
  module TSS {
    interface Message_Serialization {
      void Serialize (
        in    MESSAGE_TYPE               message,
        in    DATA_BUFFER_TYPE           buffer,
        inout TPM::Primitive_Marshalling marshalling_interface,
        out   BYTE_SIZE_TYPE             bytes_consumed,
        out   RETURN_CODE_TYPE           return_code);

      void DeSerialize (
        in    DATA_BUFFER_TYPE           buffer,
        out   MESSAGE_TYPE               message,
```

```
        inout TPM::Primitive_Marshalling marshalling_interface,
        out   BYTE_SIZE_TYPE             bytes_consumed,
        out   RETURN_CODE_TYPE           return_code);
    };

    interface Serialization {
      //! The TPM gets the serialization interface for the message type
      //! from the TS
      void Get_Serialization (
        in  GUID_TYPE             message_type_id,
        out Message_Serialization serialization,
        out RETURN_CODE_TYPE      return_code);
    };
  };
};

#endif //! __FACE_TSS_SERIALIZATION
```

E.3.3.1 Serialize(TS) Function

Prototype of the serialize function for messages of a given data type. Used by the TPM to help serialize complex message structures and implemented by a TS Interface helper function.

```
/* IDL declaration */
module FACE {
  module TSS {
    interface Message_Serialization {
      void Serialize (
        in    MESSAGE_TYPE               message,
        in    DATA_BUFFER_TYPE           buffer,
        inout TPM::Primitive_Marshalling marshalling_interface,
        out   BYTE_SIZE_TYPE             bytes_consumed,
        out   RETURN_CODE_TYPE           return_code);

    };
  };
};
```

The parameters to this method are as follows:

- *message* – a reference to the instance of PCS/PSSS data passed between the TS and TPM; message is not encoded
- *buffer* – a reference to the buffer location to store the serialized message once serialization completes
- *marshalling_interface* – a reference to the serialize functions of the base types (e.g., float, integer) that the TPM is providing
- *bytes consumed* – the number of bytes within the data buffer consumed by the serialized message
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Serialize*(TPM) is one of the following:

- NO_ERROR to indicate serialization was successful

- NO_ACTION to indicate a failure due to unknown reasons
- INVALID_PARAM to indicate one or more parameters supplied is null or not in range

E.3.3.2 DeSerialize(TS) Function

Prototype of the deserialize function for messages of a given data type. Used by the TPM to help deserialize complex message structures. Implemented by a TS Interface helper function.

```
/* IDL declaration */
module FACE {
  module TSS {
    interface Message_Serialization {
      void DeSerialize (
        in    DATA_BUFFER_TYPE           buffer,
        out   MESSAGE_TYPE               message,
        inout TPM::Primitive_Marshalling marshalling_interface,
        out   BYTE_SIZE_TYPE             bytes_consumed,
        out   RETURN_CODE_TYPE           return_code);
    };
  };
};
```

The parameters to this method are as follows:

- *buffer* – a reference to the buffer location currently holding the encoded datagram element on which the deserialization is performed
- *message* – a reference to the instance of PCS/PSSS data passed between the TS and TPM; message is not encoded
- *marshalling_interface* – a reference to the deserialize functions of the base types (e.g., float, integer) that the TPM is providing
- *bytes_consumed* – the number of bytes within the data buffer used to create the deserialized message
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *DeSerialize*(TPM) is one of the following:

- NO_ERROR to indicate serialization was successful
- NO_ACTION to indicate a failure due to unknown reasons
- INVALID_PARAM to indicate one or more parameters supplied is null or not in range

E.3.3.3 Get_Serialization(TSS) Function

Provided by a TSS UoC, *Get_Serialization*(TS) allows the TPM to get the location of the message serialization helper function from the TSS. A TPM instance uses one instance of *Serialization*(TS) for all of its queries.

```
module FACE {
  module TSS {
    interface Serialization {
```

```
      //! The TPM gets the serialization interface for the message type from
the TS
      void Get_Serialization (
        in  GUID_TYPE             message_type_id,
        out Message_Serialization serialization,
        out RETURN_CODE_TYPE      return_code);
    };
  };  }; //end TSS
}; //end FACE
```

The parameters to this method are as follows:

- *message_type_id* – a reference to the UUID instance for this message instance
- *serialization* – a reference to the *Message_Serialization* interface which provides the serialize and deserialize functions for the message instance
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Get_Serialization*(TS) is one of the following:

- NO_ERROR to indicate *Get_Serialization*(TS) was successful
- NOT_AVAILABLE to indicate a UoC implementing *Serialization*(TS) is not yet initialized
- INVALID_PARAM to indicate the *message_type_id* supplied is null or not in range

E.3.4 Type-Specific Extended Typed Interface Specification

```
//! Source file: FACE/TSS/Extended.idl

#ifndef __FACE_TSS_EXTENDED
#define __FACE_TSS_EXTENDED

#include <FACE/TSS/Common.idl>

module FACE {
  module TSS {
    module Typed<typename DATATYPE_TYPE, typename RESPONSE_DATATYPE> {

      interface Read_Callback {
        void Callback_Handler (
           in    CONNECTION_ID_TYPE  connection_id,
           in    TRANSACTION_ID_TYPE transaction_id,
           in    RESPONSE_DATATYPE   message,
           in    HEADER_TYPE         header,
           in    QoS_EVENT_TYPE      qos_parameters,
           out   RETURN_CODE_TYPE    return_code);
      };

      interface TypedTS {
        void Send_Message_Blocking (
          in    CONNECTION_ID_TYPE connection_id,
          in    TIMEOUT_TYPE       timeout,
          in    DATATYPE_TYPE      message,
          out   HEADER_TYPE        header,
          out   QoS_EVENT_TYPE     qos_parameters,
```

```
          out   RESPONSE_DATATYPE  return_data,
          out   RETURN_CODE_TYPE   return_code);

        //! Note: The callback parameter is semantically an in parameter
        //!      but is inout to avoid an undesirable mapping in C++.
        void Send_Message_Async (
          in    CONNECTION_ID_TYPE  connection_id,
          in    TIMEOUT_TYPE        timeout,
          in    TRANSACTION_ID_TYPE transaction_id,
          in    DATATYPE_TYPE       message,
          inout Read_Callback       callback,
          out   RETURN_CODE_TYPE    return_code);
      };
    };
  };
};

#endif //! __FACE_TSS_EXTENDED
```

E.3.4.1 Callback_Handler(TS) Function

When provided in the *Send_Message_Async*, the *Callback_Handler* function, provided by the PCS/PSSS UoC, is called asynchronously by a TSS UoC upon receipt of data. Data is then provided by the TSS invoking the *Callback_Handler* function as a callback to the PCS/PSSS UoC. The callback is discarded by the TSS once data is received.

```
/* IDL declaration */
module FACE {
  module TSS {
    module Typed<typename DATATYPE_TYPE, typename RESPONSE_DATATYPE> {

      interface Read_Callback {
        void Callback_Handler (
           in    CONNECTION_ID_TYPE  connection_id,
           in    TRANSACTION_ID_TYPE transaction_id,
           in    RESPONSE_DATATYPE   message,
           in    HEADER_TYPE         header,
           in    QoS_EVENT_TYPE      qos_parameters,
           out   RETURN_CODE_TYPE    return_code);
      };
    };
  };
};
```

The parameters to this method are as follows:

- *connection_id* – identifier for the connection on which data was received
- *transaction_id* – identifier used to associate messages in the request/response message pattern

  Clients are provided with the *transaction_id* along with its response message when a response message from a server is received. Servers are provided with the *transaction_id* along with a request message when a request message is received. The TSS UoC ensures the server's reply message is correctly associated with the client's request message.
- *message* – a reference to the data of returned in response to the sent message
- *header* – a reference to the header instance for this Message Instance
- *qos_parameters* – a reference to the QoS event values for this Message Instance

- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Callback_Handler* is one of the following:

- NO_ERROR to indicate the *Callback_Handler* method was successful
- DATA_OVERFLOW to indicate the rate of received messages sent via callback exceeds the ability of the UoP to process those messages

E.3.4.2 Send_Message_Blocking(TS) Function

The *Send_Message_Blocking*(TS) function is used to send data to a destination when the sender requires a response. The response to the sent data is returned in the same method.

```
/* IDL declaration */
module FACE {
  module TSS {
    module Typed<typename DATATYPE_TYPE, typename RESPONSE_DATATYPE> {
      interface TypedTS {
        void Send_Message_Blocking (
          in    CONNECTION_ID_TYPE connection_id,
          in    TIMEOUT_TYPE       timeout,
          in    DATATYPE_TYPE      message,
          out   HEADER_TYPE        header,
          out   QoS_EVENT_TYPE     qos_parameters,
          out   RESPONSE_DATATYPE  return_data,
          out   RETURN_CODE_TYPE   return_code);
      };
    };
  };
};
```

The parameters to this method are as follows:

- *connection_id* – identifier for the connection on which to send data
- *timeout* – an upper limit on the blocking time a UoP is willing to wait for the *Send_Message_Blocking* method to return control to the UoP; *Send_Message_Blocking* can return earlier, but no later than the timeout provided
- *message* – a reference to the PCS/PSSS data to send
- *return_data* – a reference to the PCS/PSSS data returned in response to the sent data
- *header* – a reference to the header instance for this returned Message Instance
- *qos_parameters* – *a* reference to the QoS event values for this returned Message Instance
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Send_Message_Blocking* is one of the following:

- NO_ERROR to indicate the *Send_Message_Blockin*g method was successful

- NO_ACTION to indicate a failure due to unknown reasons
- NOT_AVAILABLE to indicate the TS is not yet initialized or the connection for the underlying technology is unavailable
- INVALID_PARAM to indicate one or more parameters supplied is null or not in range or the *connection_id* does not exist
- TIMED_OUT to indicate a timeout was specified and exceeded
- CONNECTION_CLOSED to indicate the TS-UoP connection is not open/available
- DATA_OVERFLOW to indicate the rate of messages to send exceeds the ability of the transport to send the message

E.3.4.3 Send_Message_Async(TS) Function

The *Send_Message_Async*(TS) function is used to send data to a destination that requires a response. The data returned is provided using the *Callback_Handler* function. The callback is provided during the call rather than being registered at startup and the callback is discarded by the TSS once data for the transaction is received.

```
/* IDL declaration */
module FACE {
  module TSS {
    module Typed<typename DATATYPE_TYPE, typename RESPONSE_DATATYPE> {
      interface TypedTS {
        void Send_Message_Async (
          in    CONNECTION_ID_TYPE  connection_id,
          in    TIMEOUT_TYPE        timeout,
          in    TRANSACTION_ID_TYPE transaction_id,
          in    DATATYPE_TYPE       message,
          inout Read_Callback       callback,
          out   RETURN_CODE_TYPE    return_code);
      };
    };
  };
};
```

The parameters to this method are as follows:

- *connection_id* – identifier for the connection on which to send data
- *timeout* – an upper limit on the blocking time a UoP is willing to wait for the *Send_Message_Async* method to return control to the UoP; *Send_Message_Async* can return earlier, but no later than the timeout provided
- *transaction_id* – identifier used to associate messages in the request/response message pattern

  Clients are returned the *transaction_id* from *Send_Message_Async* when sending a request message. Servers provide the *transaction_id* to *Send_Message_Async* when sending a response message. The TSS UoC ensures the server's reply message is correctly associated with the client's request message.
- *message* – a reference to the PCS/PSSS data to send

- *callback* – a reference to a *Callback_Handler* method to handle the received type-specific message as an asynchronous response to the sent data
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Send_Message_Async* is one of the following:

- NO_ERROR to indicate the *Send_Message_Async* method was successful
- NO_ACTION to indicate a failure due to unknown reasons
- NOT_AVAILABLE to indicate the TS is not yet initialized or the connection for the underlying technology is unavailable
- INVALID_PARAM to indicate one or more parameters supplied is null or not in range or the *connection_id* does not exist
- TIMED_OUT to indicate a timeout was specified and exceeded
- CONNECTION_CLOSED to indicate the TS-UoP connection is not open/available
- DATA_OVERFLOW to indicate the rate of messages to send exceeds the ability of the transport to send the message

E.3.5 Component State Persistence Interface Specification

The CSP Interface is optional. The CSP Interface is defined by an abstraction interface allowing UoCs to store and retrieve internal state information or private data without defining the data in the FACE Data Architecture. A PCS, PSSS, or TSS UoC may write data with this interface; however, only an instance of the UoC that stored the data may retrieve it. The interface is modeled on the standard create, read, update, and delete interface for universal Data Stores.

```
//! Source file: FACE/TSS/CSP.idl

#ifndef __FACE_TSS_CSP
#define __FACE_TSS_CSP

#include <FACE/TSS/Common.idl>

module FACE {
  module TSS {
    module CSP {

      enum DATA_STORE_KIND_TYPE {
        PRIVATE_DATA_STORE,
        CHECKPOINT_DATA_STORE
      };

      typedef long long DATA_STORE_TOKEN_TYPE;

      typedef long long DATA_ID_TYPE;

        //! The CSP Interface
      interface CSP {

        //! The Initialize(CSP) function call allows for the PCS, PSSS, and
```

```
        //! TSSS UoC to trigger the initialization of the CSP Interface.
        void Initialize (
          in  CONFIGURATION_RESOURCE configuration,
          out RETURN_CODE_TYPE       return_code);

        //! The Open(CSP) function allows the PCS or PSSS UoC to open
        //! a data store that is associated with checkpoint or private data.
        //! The data store is associated with a configuration name and
        //! referenced by a token returned from the function.
        void Open (
          in  GUID_TYPE             uop_id,
          in  STRING_TYPE           configuration_name,
          in  DATA_STORE_KIND_TYPE  type,
          out DATA_STORE_TOKEN_TYPE token,
          out RETURN_CODE_TYPE      return_code);

        //! The Close(CSP) function allows the PCS or PSSS UoC to close
        //! a data store.
        void Close (
          in  GUID_TYPE             uop_id,
          in  DATA_STORE_TOKEN_TYPE token,
          out RETURN_CODE_TYPE      return_code);

        //! The Create(CSP) function allows the PCS or PSSS UoC to create
        //! a data store entry.
        void Create (
          in  GUID_TYPE             uop_id,
          in  DATA_STORE_TOKEN_TYPE token,
          out DATA_ID_TYPE          data_id,
          in  DATA_BUFFER_TYPE      data,
          out RETURN_CODE_TYPE      return_code);

        //! The Read(CSP) function allows the PCS or PSSS UoC to read
        //! a data store entry.
        void Read (
          in  GUID_TYPE             uop_id,
          in  DATA_STORE_TOKEN_TYPE token,
          in  DATA_ID_TYPE          data_id,
          out DATA_BUFFER_TYPE      data,
          out RETURN_CODE_TYPE      return_code);

        //! The Update(CSP) function allows the PCS or PSSS UoC to update
        //! a data store entry.
        void Update (
          in  GUID_TYPE             uop_id,
          in  DATA_STORE_TOKEN_TYPE token,
          in  DATA_ID_TYPE          data_id,
          in  DATA_BUFFER_TYPE      data,
          out RETURN_CODE_TYPE      return_code);

        //! The Delete(CSP) function allows the PCS or PSSS UoC to delete
        //! a data store entry.
        void Delete (
          in  GUID_TYPE             uop_id,
          in  DATA_STORE_TOKEN_TYPE token,
          in  DATA_ID_TYPE          data_id,
          out RETURN_CODE_TYPE      return_code);
      };
    };
  };
};

#endif // __FACE_TSS_CSP
```

E.3.5.1 Initialize(CSP) Function

The *Initialize*(CSP) function call allows for the PCS and PSSS UoC to trigger the initialization of the CSP interface.

```
/* IDL declaration */
module FACE {
  module TSS {
    module CSP {
      //! The CSP Interface
      interface CSP {
        //! The Initialize(CSP) function call allows for the PCS, PSSS,
        //! and TSSS UoC to trigger the initialization of the CSP
        //! Interface.
        void Initialize (
          in  CONFIGURATION_RESOURCE configuration,
          out RETURN_CODE_TYPE       return_code);

      };
    };
  };
};
```

The parameters to this method are as follows:

- *configuration* – specifies the name of the configuration for the CSP Interface
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Initialize*(CSP) is one of the following:

- NO_ERROR to indicate CSP was successfully initialized according to the configuration data
- NO_ACTION to indicate CSP has previously been successfully initialized
- NOT_AVAILABLE to indicate the configuration data is not accessible
- INVALID_CONFIG to indicate the configuration data has an error
- IN_PROGRESS to indicate the *initialize* is still in progress and the CSP has not yet transitioned to a normal state

Note: To support minimal blocking at startup, the initialize may return before it transitions to a nominal state. Subsequent calls return IN_PROGRESS until it transitions to its nominal state. Once the transition occurs, the next call to initialize returns NO_ACTION.

E.3.5.2 Open(CSP) Function

The *Open*(CSP) function call allows for the PCS and PSSS UoC open a data store associated with checkpoint and/or private data.

```
/* IDL declaration */
module FACE {
  module TSS {
    module CSP {
```

```
      //! The CSP Interface
      interface CSP {
        //! The Open(CSP) function allows the PCS or PSSS UoC to open
        //! a data store that is associated with checkpoint or private
        //! data. The data store is associated with a configuration name
        //! and referenced by a token returned from the function.
        void Open (
          in  GUID_TYPE             uop_id,
          in  STRING_ TYPE          configuration_name,
          in  DATA_STORE_KIND_TYPE  type,
          out DATA_STORE_TOKEN_TYPE token,
          out RETURN_CODE_TYPE      return_code);

      };
    };
  };
};
```

The parameters to this method are as follows:

- *uop_id* – identifier for the type of PCS/PSSS UoC using the CSP interface
- *configuration_name* – reference to a name for a data store matching the *configuration* of the CSP
- *type* – indication if the data store is to be used for private data or checkpoint data
- *token* – identifier for this data store which is returned by the CSP Interface; the identifier is used in subsequent interactions with the CSP pertaining to this data store
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Open*(CSP) is one of the following:

- NO_ERROR to indicate the UoC's data store was successfully opened
- NO_ACTION to indicate a failure due to unknown reasons
- NOT_AVAILABLE to indicate the CSP is not yet initialized or the underlying storage medium is unavailable
- INVALID_PARAM to indicate one or more parameters supplied is null or unknown
- INVALID_CONFIG to indicate the configuration data does not match one or more supplied parameter

E.3.5.3 Close(CSP) Function

The *Close*(CSP) function call allows for the PCS and PSSS UoC to close a Data Store.

```
/* IDL declaration */
module FACE {
  module TSS {
    module CSP {
      //! The CSP Interface
      interface CSP {
        //! The Close(CSP) function allows the PCS or PSSS UoC to close
```

```
        //! a data store.
        void Close (
          in  GUID_TYPE             uop_id,
          in  DATA_STORE_TOKEN_TYPE token,
          out RETURN_CODE_TYPE      return_code);

      };
    };
  };
};
```

The parameters to this method are as follows:

- *uop_id* – identifier for the PCS/PSSS UoC using the CSP interface
- *token* – identifier for the data store to close
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Close* is one of the following:

- NO_ERROR to indicate the UoC's data store was successfully closed
- NO_ACTION to indicate a failure due to unknown reasons
- NOT_AVAILABLE to indicate the CSP is not yet initialized or the underlying storage medium is unavailable
- INVALID_PARAM to indicate *token* supplied is null or unknown

E.3.5.4 Create(CSP) Function

The *Create*(CSP) function call allows for the PCS and PSSS UoC to create a Data Store entry.

```
/* IDL declaration */
module FACE {
  module TSS {
    module CSP {
      //! The CSP Interface
      interface CSP {
        //! The Create(CSP) function allows the PCS or PSSS UoC to create
        //! a data store entry.
        void Create (
          in  GUID_TYPE             uop_id,
          in  DATA_STORE_TOKEN_TYPE token,
          in  DATA_ID_TYPE          data_id,
          in  DATA_BUFFER_TYPE      data,
          out RETURN_CODE_TYPE      return_code);

      };
    };
  };
};
```

The parameters to this method are as follows:

- *uop_id* – identifier for the PCS/PSSS UoC using the CSP interface

- *token* – identifier for the CSP data store to create the entry; the token is provided by *Open*(CSP)
- *data_id* – identifier of the data entry created within the UoC's data store
- *data* – the PCS/PSSS data to store
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Create*(CSP) is one of the following:

- NO_ERROR to indicate the *Create* method was successful
- NO_ACTION to indicate a failure due to unknown reasons
- NOT_AVAILABLE to indicate the CSP is not yet initialized or the underlying storage medium is unavailable
- INVALID_PARAM to indicate one or more parameters supplied is null or unknown or the *token* does not exist
- CONNECTION_CLOSED to indicate the UoC's data store is not open

E.3.5.5 Read(CSP) Function

The *Read*(CSP) function call allows for the PCS and PSSS UoC to read a Data Store entry.

```
/* IDL declaration */
module FACE {
  module TSS {
    module CSP {
      //! The CSP Interface
      interface CSP {
        //! The Read(CSP) function allows the PCS or PSSS UoC to read
        //! a data store entry.
        void Read (
          in    GUID_TYPE             uop_id,
          in    DATA_STORE_TOKEN_TYPE token,
          in    DATA_ID_TYPE          data_id,
          inout DATA_BUFFER_TYPE      data,
          out   RETURN_CODE_TYPE      return_code);

      };
    };
  };
};
```

The parameters to this method are as follows:

- *uop_id* – identifier for the PCS/PSSS UoC using the CSP interface
- *token* – identifier for the CSP data store from which to read; the token is provided by *Open*(CSP)
- *data_id* – identifier of the data entry to read from the UoC's data store
- *data* – the data read being returned

- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Read*(CSP) is one of the following:

- NO_ERROR to indicate the *Read* method was successful
- NO_ACTION to indicate a failure due to unknown reasons
- NOT_AVAILABLE to indicate the CSP is not yet initialized or the underlying storage medium is unavailable
- INVALID_PARAM to indicate one or more parameters supplied is null or unknown or the *token* does not exist
- CONNECTION_CLOSED to indicate the UoC's data store is not open

E.3.5.6 Update(CSP) Function

The *Update*(CSP) function call allows for the PCS and PSSS UoC to update an existing Data Store entry.

```
/* IDL declaration */
module FACE {
  module TSS {
    module CSP {
      //! The CSP Interface
      interface CSP {
        //! The Update(CSP) function allows the PCS or PSSS UoC to update
        //! a data store entry.
        void Update (
          in  GUID_TYPE             uop_id,
          in  DATA_STORE_TOKEN_TYPE token,
          in  DATA_ID_TYPE          data_id,
          in  DATA_BUFFER_TYPE      data,
          out RETURN_CODE_TYPE      return_code);

      };
    };
  };
};
```

The parameters to this method are as follows:

- *uop_id* – identifier for the PCS/PSSS UoC using the CSP interface
- *token* – identifier for the CSP data store to update; the token is provided by *Open*(CSP)
- *data_id* – identifier of the data entry to update within the UoC's data store
- *data* – the PCS/PSSS data that will replace what is currently in the data store
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Update*(CSP) is one of the following:

- NO_ERROR to indicate the *Update* method was successful
- NO_ACTION to indicate a failure due to unknown reasons
- NOT_AVAILABLE to indicate the CSP is not yet initialized or the underlying storage medium is unavailable
- INVALID_PARAM to indicate one or more parameters supplied is null or unknown or the *token* does not exist
- CONNECTION_CLOSED to indicate the UoC's data store is not open

E.3.5.7 Delete(CSP) Function

The *Delete*(CSP) function call allows for the PCS and PSSS UoC to delete a Data Store entry.

```
/* IDL declaration */
module FACE {
  module TSS {
    module CSP {
      //! The CSP Interface
      interface CSP {
        //! The Delete(CSP) function allows the PCS or PSSS UoC to delete
        //! a data store entry.
        void Delete (
          in  GUID_TYPE             uop_id,
          in  DATA_STORE_TOKEN_TYPE token,
          in  DATA_ID_TYPE          data_id,
          out RETURN_CODE_TYPE      return_code);

      };
    };
  };
};
```

The parameters to this method are as follows:

- *uop_id* – identifier for the PCS/PSSS UoC using the CSP interface
- *token* – identifier for the CSP data store which contains the entry; the token is provided by *Open*(CSP)
- *data_id* – identifier of the data entry to delete within the UoC's data store
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Delete*(CSP) is one of the following:

- NO_ERROR to indicate the *Delete* method was successful
- NO_ACTION to indicate a failure due to unknown reasons
- NOT_AVAILABLE to indicate the CSP is not yet initialized or the underlying storage medium is unavailable

- INVALID_PARAM to indicate one or more parameters supplied is null or unknown or the *token* does not exist
- CONNECTION_CLOSED to indicate the UoC's data store is not open

E.4 TSS Intra-Segment Interfaces

E.4.1 Type Abstraction Interface Specification

The FACE Type Abstraction interface is optional. It provides a standard mechanism for TS portability across system implementations which can be conformed and verified as a software component. An adapter is used between the Transport Services interface and the Type Abstraction interface where the typed message in the *Send_Message*, *Receive_Message*, and *Callback_Handler* is re-cast to a general reference with a TS message identifier. The parameters and return codes are consistent with the Transport Services interface in Section E.3.1. MESSAGE_TYPE replaces DATATYPE_TYPE for message in the *Callback_Handler* (Section E.3.2.1), *Receive_Message* (Section E.3.2.2), *Send_Message* (Section E.3.2.3), *Send_Message_Blockin*g (Section E.3.4.2), and *Send_Message_Async* (Section E.3.4.3).

```
//! Source file: FACE/TSS/TypeAbstraction.idl

#ifndef __FACE_TSS_TYPEABSTRACTION
#define __FACE_TSS_TYPEABSTRACTION

#include <FACE/TSS/Common.idl>

module FACE {
  module TSS {
    module TypeAbstraction {

      //! The Read_Callback interface provides a callback prototype for the
      //! TS Capability and is used to support receiving periodic data
      //! without the TS Capability having to poll for data. If a callback
      //! is used by the TS capability it is registered using the
      //! Register_Callback.
      interface Read_Callback {

        void Callback_Handler (
          in  CONNECTION_ID_TYPE  connection_id,
          in  TRANSACTION_ID_TYPE transaction_id,
          in  MESSAGE_TYPE        message,
          in  HEADER_TYPE         header,
          in  QoS_EVENT_TYPE      qos_parameters,
          out RETURN_CODE_TYPE    return_code);
      };

      //! The Type Abstraction TA Interface
      interface TypeAbstractionTS {

        void Receive_Message (
          in     CONNECTION_ID_TYPE  connection_id,
          in     TIMEOUT_TYPE        timeout,
          inout  TRANSACTION_ID_TYPE transaction_id,
          in     MESSAGE_SIZE_TYPE   size_limit,
          inout  MESSAGE_TYPE        message,
          out    HEADER_TYPE         header,
          out    QoS_EVENT_TYPE      qos_parameters,
          out    RETURN_CODE_TYPE    return_code);
```

```
        void Send_Message (
          in    CONNECTION_ID_TYPE  connection_id,
          in    TIMEOUT_TYPE        timeout,
          inout TRANSACTION_ID_TYPE transaction_id,
          in    MESSAGE_TYPE        message,
          out   MESSAGE_SIZE_TYPE   size_sent,
          out   RETURN_CODE_TYPE    return_code);

        void Send_Message_Blocking (
          in  CONNECTION_ID_TYPE connection_id,
          in  TIMEOUT_TYPE       timeout,
          in  MESSAGE_SIZE_TYPE  size_limit,
          in  MESSAGE_TYPE       message,
          out MESSAGE_SIZE_TYPE  size_sent,
          out HEADER_TYPE        header,
          out QoS_EVENT_TYPE     qos_parameters,
          out MESSAGE_TYPE       return_data,
          out RETURN_CODE_TYPE   return_code);

        //! Note: The callback parameter is semantically an in parameter
        //!       but is inout to avoid an undesirable mapping in C++.
        void Send_Message_Async (
          in    CONNECTION_ID_TYPE  connection_id,
          in    TIMEOUT_TYPE        timeout,
          in    TRANSACTION_ID_TYPE transaction_id,
          in    MESSAGE_TYPE        message,
          inout Read_Callback       callback,
          out   MESSAGE_SIZE_TYPE   size_sent,
          out   RETURN_CODE_TYPE    return_code);

        //! The purpose of Register_Callback(TA) is to provide a mechanism
        //! to read data without polling. This is used for
        //! publish/subscribe transportation mechanisms.
        void Register_Callback (
          in     CONNECTION_ID_TYPE connection_id,
          inout  Read_Callback      data_callback,
          in     MESSAGE_SIZE_TYPE  max_message_size,
          out    RETURN_CODE_TYPE   return_code);
      };
    };
  };
};

#endif //! __FACE_TSS_TYPEABSTRACTION
```

E.4.2 Transport Protocol Module (TPM) Interface Specification

There may or may not be a TPM present in an implementation. The TPM provides a standard mechanism to access transports which may be external to a TSS implementation. A TSS can be extended to interface to a TPM using the TPM interface to promote interoperability between different TSS implementations. By isolating the transport protocol behind a standard interface, implementation of TSS UoCs may use any TPM Interface conformant module to provide the required transport functionality. A TSS UoC can also use a TPM Interface as part of its native design if desired.

```
//! Source file: FACE/TSS/TPM.idl

#ifndef __FACE_TSS_TPM
#define __FACE_TSS_TPM

#include <FACE/TSS/Common.idl>
```

```
module FACE {
  module TSS {
    module TPM {
      typedef UID_TYPE CHANNEL_ID_TYPE;

      enum EVENT_TYPE {
       INIT_COMPLETE,  //initialization has completed
       XPORT_DEGRADED, //a failure, such as being oversubscribed, is degrading
                       //transport performance
       CBIT_FAIL,      //continuous built-in-test failure
       IBIT_FAIL,      //initiated built-in-test failure
       CHANNEL_FAIL,   //a particular failure has occurred in a channel
       LOST_LINK,      //the transport link cannot detect wire activity
       TRANSMIT_COMPLETE //the last transmission has completed
      };

      //! Two kinds of callbacks are provided to receive data or events.
Callbacks
      //! registered to receive events get called when changes in the
channel(e.g. a
      //! disruption in the channel), transports (e.g. lost link, a degradation
in
      //! network performance, a cbit failure), as well as notification when
datagrams
      //! have completed transmission on the wire)
      interface TPM_Callback {
        typedef long EVENT_CODE_TYPE;
        typedef STRING_TYPE DIAGNOSTIC_MSG_TYPE;

        enum CALLBACK_KIND_TYPE{
          DATA,  //callback kind for a message incoming from the transport
          EVENT, //callback kind for a change in event status
          BOTH   //callback kind for both a message and change in event status
        };

        readonly attribute CALLBACK_KIND_TYPE callback_kind;

        void Data_Callback_Handler (
          in  CHANNEL_ID_TYPE     channel_id,
          in  TRANSACTION_ID_TYPE transaction_id,
          in  MESSAGE_TYPE        message,
          in  HEADER_TYPE         tss_header,
          in  QoS_EVENT_TYPE      qos_parameters,
          out RETURN_CODE_TYPE    return_code);

        void Event_Callback_Handler (
          in  CHANNEL_ID_TYPE     channel_id,
          in  TRANSACTION_ID_TYPE transaction_id,
          in  EVENT_TYPE          event,
          in  EVENT_CODE_TYPE     event_code,
          in  DIAGNOSTIC_MSG_TYPE diagnostic_msg,
          out RETURN_CODE_TYPE    return_code);
      };

      //! The TPM Interface
      interface TPMTS {

        typedef sequence<CHANNEL_ID_TYPE> CHANNEL_ID_SEQ_TYPE;

        enum TPM_STATE_TYPE{
          NORMAL,
          TEST,
```

```
          RESUME,
          PAUSE,
          SHUTDOWN,
          SECURE
        };

        enum LEVEL_OF_TEST_TYPE {
          CBIT,
          IBIT,
          PBIT
        };

        union STATE_CHANGE_DATA_TYPE switch(TPM_STATE_TYPE) {
        //!  Provide a list of channels which must have their associated
       //!  data cleared

        case SECURE:
          CHANNEL_ID_SEQ_TYPE channels_to_clear;

        //!  Different levels of test to allow for destructive and non-
destructive
        //!  Testing (CBIT, IBIT, etc)
        case TEST:
          LEVEL_OF_TEST_TYPE test_level;

        //the following cases don't have data associated with
        // them on a state change request
        //case NORMAL: //empty
        //case RESUME: //empty
        //case PAUSE: //empty
        //case SHUTDOWN: //empty

        };


        //! Initialize provides a method for use during startup to initialize
the
        //! transport hardware and the protocol. Initialize would be called
after each
        //! of the protocol binding module's functions are registered with the
service
        //! interface
        void Initialize (
          in  CONFIGURATION_RESOURCE configuration,
          out RETURN_CODE_TYPE       return_code);

        //! Open_Channel establishes an endpoint connection with another TS
domain.
        //! The primary TS can establish a contract with the underlying
        //! protocol and transport for security and quality of service.
        void Open_Channel (
          in  CONNECTION_NAME_TYPE endpoint_name,
          in  DATA_BUFFER_TYPE     transport_config,
          in  DATA_BUFFER_TYPE     security_config,
          out CHANNEL_ID_TYPE      channel_id,
          out RETURN_CODE_TYPE     return_code);

        void Close_Channel (
          in  CHANNEL_ID_TYPE  channel_id,
          out RETURN_CODE_TYPE return_code);

        //! State change is requested to control the transport, such as sending
the TPM
```

```
        //! into test or have the TPM temporarily suspend communications.  Data
may be
        //! associated with the state change, such as indicating the level of
test to
        //! perform
        void Request_TPM_State_Change (
          in  TPM_STATE_TYPE         new_state,
          in  STATE_CHANGE_DATA_TYPE data,
          out RETURN_CODE_TYPE       return_code);

        //! Is_Data_Available allows users to retrieve which channels have
activity and
        //! can subsequently be read without blocking.  NULL timeout allows the
user to
        //! block until there is any data received
        void Is_Data_Available (
          in  CHANNEL_ID_SEQ_TYPE channel_ids,
          in  TIMEOUT_TYPE        timeout,
          out CHANNEL_ID_SEQ_TYPE available_ids,
          out RETURN_CODE_TYPE    return_code);

        //! Used to monitor the health and availability of the transport.  The
status
        //! would allow the user to continue to use the transport or consider
it a
        //! BUS FAIL
        void Get_TPM_Status (
          out EVENT_TYPE       status,
          out RETURN_CODE_TYPE return_code);

        //! Read_From_Transport allows the primary TS to read incoming
datagrams.
        //! If a non-zero timeout is used, the call blocks until data is
received
        //! and processed by the protocol or the timeout is reached.
        //! If used with isDataAvailable, a timeout of 0 returns the datagram
already
        //! processed by the protocol otherwise if no data was received returns
0 bytes
        //! in the message
        void Read_From_Transport (
          in  CHANNEL_ID_TYPE     channel_id,
          in  TIMEOUT_TYPE        timeout,
          out TRANSACTION_ID_TYPE transaction_id,
          out MESSAGE_TYPE        message,
          out HEADER_TYPE         TSS_header,
          out QoS_EVENT_TYPE      qos_parameters,
          out RETURN_CODE_TYPE    return_code);

        //! Write_To_Transport provides the ability to write a datagram to the
binding
        //! module for protocol processing and transport.  The TPM transmits
the
        //! message within this method call with a maxDelay of 0 or establish a
pipeline
        //! of messages.  A send_TPM_event is used to release the buffers used
by the
        //! primary TS.
        void Write_To_Transport (
          in  CHANNEL_ID_TYPE     channel_id,
          in  TIMEOUT_TYPE        max_delay,
          in  MESSAGE_TYPE        message,
          in  HEADER_TYPE         TSS_header,
```

```
          out TRANSACTION_ID_TYPE transaction_id,
          out RETURN_CODE_TYPE    return_code);

        //! The callback functions are provided by the user and must be
registered for
        //! the binding module to call.
        void Register_TPM_Callback (
          in     CHANNEL_ID_TYPE    channel_id,
          inout  TPM_Callback       callback,
          out    RETURN_CODE_TYPE   return_code);

        void Unregister_TPM_Callback (
          in  CHANNEL_ID_TYPE             channel_id,
          in  TPM_Callback::CALLBACK_KIND_TYPE rcallback_kind,
          out RETURN_CODE_TYPE            return_code);
      };

      // Protocol-specific base type serialization
      // A serialization and deserialization method is provided for each of the
base
      // types
      interface Primitive_Marshalling {
        void Marshal_short (
          in    short            data,
          in    DATA_BUFFER_TYPE buffer,
          out   BYTE_SIZE_TYPE   bytes_consumed,
          out   RETURN_CODE_TYPE return_code);

        void Unmarshal_short (
          in    DATA_BUFFER_TYPE buffer,
          out   short            data,
          out   BYTE_SIZE_TYPE   bytes_consumed,
          out   RETURN_CODE_TYPE return_code);

        void Marshal_long (
          in    long             data,
          in    DATA_BUFFER_TYPE buffer,
          out   BYTE_SIZE_TYPE   bytes_consumed,
          out   RETURN_CODE_TYPE return_code);

        void Unmarshal_long (
          in    DATA_BUFFER_TYPE buffer,
          out   long             data,
          out   BYTE_SIZE_TYPE   bytes_consumed,
          out   RETURN_CODE_TYPE return_code);

        void Marshal_long_long (
          in    long long        data,
          in    DATA_BUFFER_TYPE buffer,
          out   BYTE_SIZE_TYPE   bytes_consumed,
          out   RETURN_CODE_TYPE return_code);

        void Unmarshal_long_long (
          in    DATA_BUFFER_TYPE buffer,
          out   long long        data,
          out   BYTE_SIZE_TYPE   bytes_consumed,
          out   RETURN_CODE_TYPE return_code);

        void Marshal_unsigned_short (
          in    unsigned short   data,
          in    DATA_BUFFER_TYPE buffer,
          out   BYTE_SIZE_TYPE   bytes_consumed,
          out   RETURN_CODE_TYPE return_code);
```

```
void Unmarshal_unsigned_short (
  in    DATA_BUFFER_TYPE buffer,
  out   unsigned short   data,
  out   BYTE_SIZE_TYPE   bytes_consumed,
  out   RETURN_CODE_TYPE return_code);

void Marshal_unsigned_long (
  in    unsigned long    data,
  in    DATA_BUFFER_TYPE buffer,
  out   BYTE_SIZE_TYPE   bytes_consumed,
  out   RETURN_CODE_TYPE return_code);

void Unmarshal_unsigned_long (
  in    DATA_BUFFER_TYPE buffer,
  out   unsigned long    data,
  out   BYTE_SIZE_TYPE   bytes_consumed,
  out   RETURN_CODE_TYPE return_code);

void Marshal_unsigned_long_long (
  in    unsigned long long data,
  in    DATA_BUFFER_TYPE   buffer,
  out   BYTE_SIZE_TYPE     bytes_consumed,
  out   RETURN_CODE_TYPE   return_code);

void Unmarshal_unsigned_long_long (
  in    DATA_BUFFER_TYPE   buffer,
  out   unsigned long long data,
  out   BYTE_SIZE_TYPE     bytes_consumed,
  out   RETURN_CODE_TYPE   return_code);

void Marshal_float (
  in    float            data,
  in    DATA_BUFFER_TYPE buffer,
  out   BYTE_SIZE_TYPE   bytes_consumed,
  out   RETURN_CODE_TYPE return_code);

void Unmarshal_float (
  in    DATA_BUFFER_TYPE buffer,
  out   float            data,
  out   BYTE_SIZE_TYPE   bytes_consumed,
  out   RETURN_CODE_TYPE return_code);

void Marshal_double (
  in    double           data,
  in    DATA_BUFFER_TYPE buffer,
  out   BYTE_SIZE_TYPE   bytes_consumed,
  out   RETURN_CODE_TYPE return_code);

void Unmarshal_double (
  in    DATA_BUFFER_TYPE buffer,
  out   double           data,
  out   BYTE_SIZE_TYPE   bytes_consumed,
  out   RETURN_CODE_TYPE return_code);

void Marshal_long_double (
  in    long double      data,
  in    DATA_BUFFER_TYPE buffer,
  out   BYTE_SIZE_TYPE   bytes_consumed,
  out   RETURN_CODE_TYPE return_code);

void Unmarshal_long_double (
  in    DATA_BUFFER_TYPE buffer,
```

```
            out   long double      data,
            out   BYTE_SIZE_TYPE   bytes_consumed,
            out   RETURN_CODE_TYPE return_code);

          void Marshal_char (
            in    char             data,
            in    DATA_BUFFER_TYPE buffer,
            out   BYTE_SIZE_TYPE   bytes_consumed,
            out   RETURN_CODE_TYPE return_code);

          void Unmarshal_char (
            in    DATA_BUFFER_TYPE buffer,
            out   char             data,
            out   BYTE_SIZE_TYPE   bytes_consumed,
            out   RETURN_CODE_TYPE return_code);

          void Marshal_boolean (
            in    boolean          data,
            in    DATA_BUFFER_TYPE buffer,
            out   BYTE_SIZE_TYPE   bytes_consumed,
            out   RETURN_CODE_TYPE return_code);

          void Unmarshal_boolean (
            in    DATA_BUFFER_TYPE buffer,
            out   boolean          data,
            out   BYTE_SIZE_TYPE   bytes_consumed,
            out   RETURN_CODE_TYPE return_code);

          void Marshal_octet (
            in    octet            data,
            in    DATA_BUFFER_TYPE buffer,
            out   BYTE_SIZE_TYPE   bytes_consumed,
            out   RETURN_CODE_TYPE return_code);

          void Unmarshal_octet (
            in    DATA_BUFFER_TYPE buffer,
            out   octet            data,
            out   BYTE_SIZE_TYPE   bytes_consumed,
            out   RETURN_CODE_TYPE return_code);

          void Marshal_string (
            in    UNBOUNDED_STRING_TYPE data,
            in    DATA_BUFFER_TYPE      buffer,
            out   BYTE_SIZE_TYPE        bytes_consumed,
            out   RETURN_CODE_TYPE      return_code);

          void Unmarshal_string (
            in    DATA_BUFFER_TYPE      buffer,
            out   UNBOUNDED_STRING_TYPE data,
            out   BYTE_SIZE_TYPE        bytes_consumed,
            out   RETURN_CODE_TYPE      return_code);
        };
      };
    };
  };

  #endif // __FACE_TSS_TPM
```

E.4.2.1 Initialize(TPM) Function

The *Initialize*(TPM) function call allows for the TSS UoC to trigger the initialization of the TPM software component. It provides the entry point for the TPM at startup.

```
/* IDL declaration */
module FACE {
  module TSS {
    module TPM {
      //! The TPM Interface
      interface TPMTS {
        //! Initialize provides a method for use during startup to
        //! initialize the transport hardware and the protocol.
        //! Initialize would be called after each of the protocol binding
        //! module's functions are registered with the service interface.
        void Initialize (
          in  CONFIGURATION_RESOURCE configuration,

out RETURN_CODE_TYPE       return_code);

      };
    };
  };
};
```

The parameters to this method are as follows:

- *configuration* – specifies the name of the configuration for the Transport Protocol Module Interface
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Initialize*(TPM) is one of the following:

- NO_ERROR to indicate TPM was successfully initialized according to the configuration data
- NO_ACTION to indicate TPM has been initialized successfully previously
- NOT_AVAILABLE to indicate the *configuration* data is not accessible
- INVALID_CONFIG to indicate the *configuration* data has an error
- IN_PROGRESS to indicate the *initialize* is still in progress and the TPM has not yet transitioned to a normal state

Note: To support minimal blocking at startup, the initialize may return before it transitions from an initialize state to a normal state. Subsequent calls to initialize returns IN_PROGRESS until it transitions out of its initial state. Once the transition occurs, the next call to initialize returns NO_ACTION.

E.4.2.2 Open Channel(TPM) Function

The *openChannel*(TPM) call allows for a TSS UoC to establish a connection for the transport managed by the TPM using attributes required by the TSS for the transport, security and QoS.

```
/* IDL declaration */
module FACE {
  module TSS {
    module TPM {
      //! The TPM Interface
      interface TPMTS {
```

```
        //! Open_Channel establishes an endpoint connection with another
        //! TS domain. The primary TS can establish a contract with the
        //! underlying protocol and transport for security and quality of
        //! service.
        void Open_Channel (
          in  CONNECTION_NAME_TYPE endpoint_name,
          in  DATA_BUFFER_TYPE     transport_config,
          in  DATA_BUFFER_TYPE     security_config,
          out CHANNEL_ID_TYPE      channel_id,
          out RETURN_CODE_TYPE     return_code);

      };
    };
  };
};
```

The parameters to this method are as follows:

- *endpoint_name* – reference to a name in the *configuration* with which the TS would like to communicate
- *transport_config* – reference to implementation-defined transport metadata, which can be managed at run-time *versus* configuration, specific to the channel being opened
- *security_config* – reference to implementation-defined security metadata such as certificates, public keys, or enabling encryption for a channel
- *channel_id* – returned by the TPM Interface indicating the reference to use for this channel in subsequent TPM calls
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *openChannel*(TPM) is one of the following:

- NO_ERROR to indicate the TS-TPM channel was successfully created
- NO_ACTION to indicate a failure due to unknown reasons
- NOT_AVAILABLE to indicate the TPM is not yet initialized or the underlying technology is unavailable
- INVALID_PARAM to indicate one or more parameters supplied is null or not in range
- INVALID_CONFIG to indicate the configuration data does not match one or more supplied parameter
- TIMED_OUT to indicate a timeout was specified and exceeded

E.4.2.3 Close Channel(TPM) Function

The *closeChannel*(TPM) function frees up any resources allocated to the channel. This can be an empty function if no cleanup is required.

```
/* IDL declaration */
module FACE {
  module TSS {
```

```
    module TPM {
      //! The TPM Interface
      interface TPMTS {
        void Close_Channel (
          in  CHANNEL_ID_TYPE  channel_id,
          out RETURN_CODE_TYPE return_code);

      };
    };
  };
};
```

The parameters to this method are as follows:

- *channel_id* – identifier for the channel to destroy
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *closeChannel*(TPM) is one of the following:

- NO_ERROR to indicate the TPM channel was successfully destroyed
- NO_ACTION to indicate a failure due to unknown reasons
- NOT_AVAILABLE to indicate the TPM is not yet initialized or the underlying technology is unavailable
- INVALID_PARAM to indicate *channelToken* supplied is null or not in range

E.4.2.4 Request_TPM_State_Change(TPM) Function

The *requestTPMStateChange*(TPM) function frees up any resources allocated to the channel. This can be an empty function if no cleanup is required.

```
/* IDL declaration */
module FACE {
  module TSS {
    module TPM {
      //! The TPM Interface
      interface TPMTS {
        //! State change is requested to control the transport, such as
        //! sending the TPM into test or have the TPM temporarily suspend
        //! communications. Data may be associated with the state change,
        //! such as indicating the level of test to perform.
        void Request_TPM_State_Change (
          in  TPM_STATE_TYPE         new_state,
          in  STATE_CHANGE_DATA_TYPE data,
          out RETURN_CODE_TYPE       return_code);

      };
    };
  };
};
```

The parameters to this method are as follows:

- *new_state* – the state to which the TPM is being requested to change

- *data* – a reference to data associated with requests to change to the test or secure states

  For the TEST state, this indicates a level of test to perform. For the SECURE state, it provides the TS credentials required to secure the transport.

- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *requestTPMStateChange*(TPM) is one of the following:

- NO_ERROR to indicate the TPM channel was successfully changed states
- NO_ACTION to indicate a failure due to unknown reasons
- NOT_AVAILABLE to indicate the TPM is not yet initialized or the underlying technology is unavailable

E.4.2.5 is_Data_Available(TPM) Function

The *isDataAvailable*(TPM) call provides a single function to indicate which previously created channels have data ready to read. Typically used to simplify buffer management and avoid race conditions.

```
/* IDL declaration */
module FACE {
  module TSS {
    module TPM {
      //! The TPM Interface
      interface TPMTS {
        //! Is_Data_Available allows users to retrieve which channels have
        //! activity and can subsequently be read without blocking.  NULL
        //! timeout allows the user to block until there is any data
        //! received.
        void Is_Data_Available (
          in  CHANNEL_ID_SEQ_TYPE channel_ids,
          in  TIMEOUT_TYPE        timeout,
          out CHANNEL_ID_SEQ_TYPE available_ids,
          out RETURN_CODE_TYPE    return_code);

      };
    };
  };
};
```

The parameters to this method are as follows:

- *channel_ids* – a list of channel token identifiers which the TS is querying for activity
- *timeout* – an upper limit on the blocking time a TS is willing to wait for the *isDataAvailable* method to return control to the TS; *isDataAvailable* can return earlier, but no later than the timeout provided
- *available_ids* – a list of channel token identifiers from the *tokenList* which have activity
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *isDataAvailable*(TPM) is one of the following:

- NO_ERROR to indicate the TPM connection was successfully destroyed
- NO_ACTION to indicate a failure due to unknown reasons
- NOT_AVAILABLE to indicate the TS is not yet initialized or the underlying technology is unavailable
- INVALID_PARAM to indicate *connection_id* supplied is null or not in range

E.4.2.6 Get_TPM_Status(TPM) Function

The *Get_TPM_Status*(TPM) function is used to monitor the health and availability of the transport, get the current state of the TPM, or receive information regarding the channel and whether it remains active and can communicate to its remote endpoint.

```
/* IDL declaration */
module FACE {
  module TSS {
    module TPM {
      //! The TPM Interface
      interface TPMTS {
        //! Used to monitor the health and availability of the transport.
        //! The status would allow the user to continue to use the
        //! transport or consider it a BUS FAIL.
        void Get_TPM_Status (
          out EVENT_TYPE       status,
          out RETURN_CODE_TYPE return_code);

      };
    };
  };
};
```

The parameters to this method are as follows:

- *status* – current state of the TPM, transport, and/or information regarding a channel
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Get_TPM_Status*(TPM) is one of the following:

- NO_ERROR to indicate the *GetTPMStatus* method was successful
- NOT_AVAILABLE to indicate the TPM is not yet initialized or the underlying technology is unavailable

E.4.2.7 Read_From_Transport(TPM) Function

The *Read_From_Transport*(TPM) function provides an interface for a TSS UoC to read a message from the TPM that has been received on the transport. This can be used as a non-blocking read with or without making use of the *Is_Data_Available* function or a blocking read

using the timeout. The TSS and TPM use the same message UID and TSS header as defined by the TS Interface. The TPM can re-construct them if the protocol optimizes them.

```
/* IDL declaration */
module FACE {
  module TSS {
    module TPM {
      //! The TPM Interface
      interface TPMTS {
        //! Read_From_Transport allows the primary TS to read incoming
        //! datagrams. If a non-zero timeout is used, the call blocks
        //! until data is received and processed by the protocol or the
        //! timeout is reached. If used with isDataAvailable, a timeout
        //! of 0 returns the datagram already processed by the protocol
        //! otherwise if no data was received returns 0 bytes in the
        //! message.
        void Read_From_Transport (
          in  CHANNEL_ID_TYPE     channel_id,
          in  TIMEOUT_TYPE        timeout,
          out TRANSACTION_ID_TYPE transaction_id,
          out MESSAGE_TYPE        message,
          out HEADER_TYPE         TSS_header,
          out QoS_EVENT_TYPE      qos_parameters,
          out RETURN_CODE_TYPE    return_code);

      };
    };
  };
};
```

The parameters to this method are as follows:

- *channel_id* – identifier for the connection on which to send data
- *timeout* – an upper limit on the blocking time a TS is willing to wait for the *Read_From_Transport* method to return control to the TS

  Read_From_Transport can return earlier, but no later than the timeout provided. A non-zero timeout waits for a message to be received up to the timeout value.
- *transaction_id* – a transaction identifier provided by the TPM to the TS used to associate messages in the request/response message pattern
- *message* – a reference to the PCS/PSSS data to read which is passed on by the TS; messages are deserialized before the message is returned to the TS
- *TSS_header* – a reference to the header instance for this Message Instance
- *qos_parameters* – a reference to the QoS attribute values the TPM accomplished for this Message Instance
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *read_From_Transport*(TPM) is one of the following:

- NO_ERROR to indicate the *readFromTransport* method was successful

- NO_ACTION to indicate a failure due to unknown reasons
- NOT_AVAILABLE to indicate the TPM is not yet initialized or the connection for the underlying technology is unavailable
- INVALID_PARAM to indicate one or more parameters supplied is null or not in range or the *channelToken* does not exist
- INVALID_MODE to indicate a callback function has been registered for this connection
- TIMED_OUT to indicate a timeout was specified and exceeded
- MESSAGE_STALE to indicate the message lifespan has been exceeded
- CONNECTION_CLOSED to indicate the TS-TPM channel is not open/available
- DATA_BUFFER_TOO_SMALL to indicate the message received exceeds the message size given on a single transaction
- DATA_OVERFLOW to indicate the rate of incoming messages exceeds the rate of messages being read

E.4.2.8 Write_To_Transport(TPM) Function

The *Write_To_Transport*(TPM) function provides an interface for a TSS UoC to write a message to the TPM for protocol processing and transmission on the transport. Max delay of zero indicates immediate processing by the TPM. Otherwise, the TSS can establish a pipeline of datagrams and track they were successfully transmitted from the TPM notification. Once notification is received by the TSS the message has completed transmission, the TSS can free the message buffer. The TSS and TPM use the same message UID and TSS header as defined by the TS Interface. The TPM may optimize them for transmission.

```
/* IDL declaration */
module FACE {
  module TSS {
    module TPM {
      //! The TPM Interface
      interface TPMTS {
        //! Write_To_Transport provides the ability to write a datagram
        //! to the binding module for protocol processing and transport.
        //! The TPM transmits the message within this method call with a
        //! maxDelay of 0 or establish a pipeline of messages.
        //! A send_TPM_event is used to release the buffers used by the
        //! primary TS.
        void Write_To_Transport (
          in  CHANNEL_ID_TYPE     channel_id,
          in  TIMEOUT_TYPE        max_delay,
          in  MESSAGE_TYPE        message,
          in  HEADER_TYPE         TSS_header,
          out TRANSACTION_ID_TYPE transaction_id,
          out RETURN_CODE_TYPE    return_code);

      };
    };
  };
};
```

The parameters to this method are as follows:

- *channel_id* – identifier for the connection on which to send data
- *maxDelay* – an upper limit on the blocking time a TS is willing to wait for the *writeToTransport* method to return control to the TS

  writeToTransport can return earlier, but no later than the timeout provided. Transmission within the context of this message is indicated with a *maxDelay* equal to 0 (immediate). A *maxDelay* > 0 queues messages within the TPM assigning it a lifespan equal to the *maxDelay* value to establish a pipeline of messages. An *Event_Callback_Handler* notifies the TS of message transmission status and is used to release TS resources.
- *message* – a reference to the PCS/PSSS data to send as passed on by the TS; message encoding is not performed prior to sending a message to the TPM
- *TSS_header* – a reference to the header instance for this Message Instance
- *transaction_id* – a transaction identifier provided by the TPM to the TS to facilitate buffer management

  If a message is not sent within the *maxDelay* or a *maxDelay* of 0 is used, the TS can receive a notification from the *send_TPM_event* when the transaction is complete to release TS resources.
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *write_To_Transport*(TPM) is one of the following:

- NO_ERROR to indicate the *writeToTransport* method was successful
- NO_ACTION to indicate a failure due to unknown reasons
- NOT_AVAILABLE to indicate the TPM is not yet initialized or the connection for the underlying technology is unavailable
- INVALID_PARAM to indicate one or more parameters supplied is null or not in range or the *channelToken* does not exist
- TIMED_OUT to indicate a timeout was specified and exceeded
- CONNECTION_CLOSED to indicate the TS-TPM channel is not open/available
- DATA_OVERFLOW to indicate the rate of messages to send exceeds the ability of the transport to send the message

E.4.2.9 Data_Callback_Handler(TPM) Functions

Data callback handlers are provided by users of the TPM interface. The handler is supplied to the TPM through the *Register_TPM_Callback* function. Once a callback is registered for a channel, the *Data_Callback_Handler* is invoked by the TPM on receipt of data.

```
/* IDL declaration */
module FACE {
```

```
  module TSS {
    module TPM {
      interface TPM_Callback {
        void Data_Callback_Handler (
          in  CHANNEL_ID_TYPE     channel_id,
          in  TRANSACTION_ID_TYPE transaction_id,
          in  MESSAGE_TYPE        message,
          in  HEADER_TYPE         tss_header,
          in  QoS_EVENT_TYPE      qos_parameters,
          out RETURN_CODE_TYPE    return_code);

      };
    };
  };
};
```

The parameters to this method are as follows:

- *channel_id* – identifier for the channel on which data was received
- *transaction_id* – a transaction identifier provided by the TPM to the TS used to associate messages in the request/response message pattern
- *message* – a reference to the PCS/PSSS data to read which is passed on by the TS; messages are deserialized before the message is returned to the TS
- *header* – a reference to the header instance for this Message Instance
- *qos_parameters* – a reference to the QoS attribute values the TPM accomplished for this Message Instance
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Data_Callback_Handler* is one of the following:

- NO_ERROR to indicate the *Data_Callback_Handler* method was successful
- DATA_OVERFLOW to indicate the rate of received messages sent via callback exceeds the ability of the UoP to process those messages

E.4.2.10 Event_Callback_Handler(TPM) Functions

Event callback handlers are provided by users of the TPM interface. The handler is supplied to the TPM through the *Register_TPM_Callback* function. Once a callback is registered for a channel, the *Event_Callback_Handler* is invoked by the TPM on changes in the TPM state or channel state.

```
/* IDL declaration */
module FACE {
  module TSS {
    module TPM {
      interface TPM_Callback {
        void Event_Callback_Handler (
          in  CHANNEL_ID_TYPE     channel_id,
          in  TRANSACTION_ID_TYPE transaction_id,
          in  EVENT_TYPE          event,
          in  EVENT_CODE_TYPE     event_code,
```

```
            in  DIAGNOSTIC_MSG_TYPE diagnostic_msg,
            out RETURN_CODE_TYPE    return_code);

      };
    };
  };
};
```

The parameters to this method are as follows:

- *channel_id* – identifier for the channel on which data was received
- *transaction_id* – a transaction identifier provided by the TPM to the TS used to associate messages in the request/response message pattern
- *event* – a notification of a change in TPM state or a TPM channel state such as initialization complete, built-in-test failure, or a channel transmission has completed
- *event_code* – an implementation-defined code for diagnostic purposes
- *diagnostic_message* – an implementation-defined string of human readable information for diagnostic purposes
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Event_Callback_Handler* is one of the following:

- NO_ERROR to indicate the *Event_Callback_Handler* method was successful
- DATA_OVERFLOW to indicate the rate of received messages sent via callback exceeds the ability of the UoP to process those messages

E.4.2.11 Register_TPM_Callback(TPM) Functions

The purpose of *Register_TPM_Callback*(TPM) is to provide a mechanism to read data without polling. Once the TS registers its *Read_TPM_Callback* interface with the TPM, the TPM invokes the *Read_TPM_Callback* method associated with the channel's messages received.

```
/* IDL declaration */
module FACE {
  module TSS {
    module TPM {
      //! The TPM Interface
      interface TPMTS {
        //! The callback functions are provided by the user and must be
        //! registered for the binding module to call.
        void Register_TPM_Callback (
          in     CHANNEL_ID_TYPE    channel_id,
          inout  TPM_Callback       callback,
          out    RETURN_CODE_TYPE   return_code);
      };
    };
  };
};
```

The parameters to this method are as follows:

- *channel_id* – connection identifier to associate the read callback
- *callback* – a reference to a *TPM_Callback* method to handle received messages
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Register_Callback* is one of the following:

- NO_ERROR to indicate the *Register_TPM_Callback* method was successful
- NO_ACTION to indicate a callback has already been registered for the *connection_id* given
- NOT_AVAILABLE to indicate the TPM is not yet initialized
- INVALID_PARAM to indicate one or more parameters supplied is null or not in range or the *connection_id* does not exist
- CONNECTION_CLOSED to indicate the TPM-TS connection is not open/available
- DATA_BUFFER_TOO_SMALL to indicate the message configured is greater than the message size given

E.4.2.12 Unregister_TPM_Callback(TPM) Functions

The purpose of *Unregister_TPM_Callback*(TPM) is to provide a mechanism to unregister the data or event callback associated with a *connection_id*.

```
/* IDL declaration */
module FACE {
  module TSS {
    module TPM {
      //! The TPM Interface
      interface TPMTS {
        void Unregister_TPM_Callback (
          in  CHANNEL_ID_TYPE             channel_id,
          in  TPM_Callback::CALLBACK_KIND_TYPE rcallback_kind,
          out RETURN_CODE_TYPE            return_code);

      };
    };
  };
};
```

The parameters to this method are as follows:

- *channel_id* – identifier for the connection to unregister a callback
- *rcallback_kind* – unregisters a data callback or event callback for a channel
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Unregister_TPM_Callback* is one of the following:

- NO_ERROR to indicate the Unr*egister_TPM_Callback* method was successful
- NOT_AVAILABLE to indicate the TPM is not yet initialized
- INVALID_PARAM to indicate the *connection_id* supplied is null or not in range

E.4.2.13 Primitive_Marshalling(TPM) Functions

Protocol-specific marshalling and unmarshalling functions are provided for each primitive type of data such as short, long, long long, unsigned short, etc. The *Primitive_Marshalling* interface is provided to the implementation of the *Message_Serialization* interface. Each marshalling and unmarshalling method follows the same pattern, for each primitive type. Only one example for marshalling is provided.

```
/* IDL declaration */
module FACE {
  module TSS {
    module TPM {
      //! Protocol-specific base type serialization
      //! A serialization and deserialization method is provided for each
      //! of the base types.
      interface Primitive_Marshalling {
        void Marshal_long (
          in    long             data,
          in    DATA_BUFFER_TYPE buffer,
          out   BYTE_SIZE_TYPE   bytes_consumed,
          out   RETURN_CODE_TYPE return_code);

      };
    };
  };
};
```

The parameters to this method are as follows:

- *data* – a reference to an instance of a data element of type long
- *buffer* – a reference to the buffer location used to place the encoded long data element
- *bytes_consumed* – the number of bytes within the data buffer consumed by the encoded long data element
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Marshal_long*(TPM) is one of the following:

- NO_ERROR to indicate *Marshal_long* was successful
- NO_ACTION to indicate a failure due to unknown reasons
- INVALID_PARAM to indicate one or more parameter supplied is null or not in range

F FACE OSS HMFM Interfaces

F.1 Introduction

The Health Monitoring and Fault Management (HMFM) Services API provides a normalized interface to manage and respond to faults, and report them in a portable manner. The HMFM Services API is part of the OSS. The FACE HMFM API is designed to directly map to ARINC 653 services when those are available. This direct mapping is only possible if the API is defined in C.

Note: The code in this document is formatted to align with the formatting constraints of the printed document.

F.2 HMFM Services API and Message Definitions

```
//! Source file: FACE/HMFM.h

#ifndef _FACE_HMFM_H
#define _FACE_HMFM_H

#include <stdint.h>

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

typedef int32_t    FACE_HMFM_long;
typedef uint32_t   FACE_HMFM_unsigned_long;
typedef int8_t     FACE_HMFM_char;

typedef enum {
    FACE_HMFM_NO_ERROR,
    FACE_HMFM_NO_ACTION,
    FACE_HMFM_NOT_AVAILABLE,
    FACE_HMFM_INVALID_PARAM,
    FACE_HMFM_INVALID_CONFIG,
    FACE_HMFM_INVALID_MODE,
    FACE_HMFM_TIMED_OUT,
    FACE_HMFM_ADDR_IN_USE,
    FACE_HMFM_PERMISSION_DENIED,
    FACE_HMFM_MESSAGE_STALE,
    FACE_HMFM_CONNECTION_IN_PROGRESS,
    FACE_HMFM_CONNECTION_CLOSED,
    FACE_HMFM_DATA_BUFFER_TOO_SMALL
} FACE_HMFM_RETURN_CODE_TYPE;

typedef void * FACE_HMFM_SYSTEM_ADDRESS_TYPE;

typedef FACE_HMFM_long FACE_HMFM_FAULT_MESSAGE_SIZE_TYPE;

typedef void * FACE_HMFM_FAULT_MESSAGE_ADDRESS_TYPE;
```

```
typedef uintptr_t FACE_HMFM_THREAD_ID_TYPE;

#define FACE_HMFM_FAULT_MESSAGE_MAXIMUM_SIZE \
  ((FACE_HMFM_FAULT_MESSAGE_SIZE_TYPE) 128)

typedef FACE_HMFM_unsigned_long FACE_HMFM_STACK_SIZE_TYPE;

typedef FACE_HMFM_char
  FACE_HMFM_FAULT_MESSAGE_TYPE[FACE_HMFM_FAULT_MESSAGE_MAXIMUM_SIZE];

typedef enum {
    FACE_HMFM_DEADLINE_MISSED,
    FACE_HMFM_APPLICATION_ERROR,
    FACE_HMFM_NUMERIC_ERROR,
    FACE_HMFM_ILLEGAL_REQUEST,
    FACE_HMFM_STACK_OVERFLOW,
    FACE_HMFM_MEMORY_VIOLATION,
    FACE_HMFM_HARDWARE_FAULT,
    FACE_HMFM_POWER_FAIL
} FACE_HMFM_FAULT_CODE_TYPE;

typedef struct {
    FACE_HMFM_FAULT_CODE_TYPE CODE;
    FACE_HMFM_FAULT_MESSAGE_SIZE_TYPE LENGTH;
    FACE_HMFM_THREAD_ID_TYPE FAILED_THREAD_ID;
    FACE_HMFM_SYSTEM_ADDRESS_TYPE FAILED_ADDRESS;
    FACE_HMFM_FAULT_MESSAGE_TYPE MESSAGE;

} FACE_HMFM_FAULT_STATUS_TYPE;

typedef void (*FACE_HMFM_FAULT_HANDLER_ENTRY_TYPE) (void);

void FACE_HMFM_Initialize (
        /*    out */ FACE_HMFM_RETURN_CODE_TYPE *return_code
    );

void FACE_HMFM_Create_Fault_Handler (
        /* in     */ FACE_HMFM_FAULT_HANDLER_ENTRY_TYPE entry_point,
        /* in     */ FACE_HMFM_STACK_SIZE_TYPE stack_size,
        /*    out */ FACE_HMFM_RETURN_CODE_TYPE *return_code
    );

void FACE_HMFM_Report_Application_Message (
        /* in     */ FACE_HMFM_FAULT_MESSAGE_ADDRESS_TYPE fault,
        /* in     */ FACE_HMFM_FAULT_MESSAGE_SIZE_TYPE length,
        /*    out */ FACE_HMFM_RETURN_CODE_TYPE *return_code
    );

void FACE_HMFM_Get_Fault_Status (
        /*    out */ FACE_HMFM_FAULT_STATUS_TYPE *fault,
        /*    out */ FACE_HMFM_RETURN_CODE_TYPE *return_code
    );

void FACE_HMFM_Raise_Application_Fault (
        /* in     */ FACE_HMFM_FAULT_CODE_TYPE code,
        /* in     */ FACE_HMFM_FAULT_MESSAGE_ADDRESS_TYPE message,
        /* in     */ FACE_HMFM_FAULT_MESSAGE_SIZE_TYPE length,
        /*    out */ FACE_HMFM_RETURN_CODE_TYPE *return_code
    );

#ifdef __cplusplus
}
#endif /* __cplusplus */
```

F.2.1 Initialize(HMFM) Function

The *Initialize*(HMFM) method allows the component to initialize the HMFM implementation.

```
void FACE_HMFM_Initialize (
        /*    out */ FACE_HMFM_RETURN_CODE_TYPE *return_code    );
```

The parameters to this method are as follows:

- *return_code* – upon return contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *FACE_HMFM_Initialize* is one of the following:

- FACE_HMFM_NO_ERROR to indicate successful completion of the operation
- FACE_HMFM_INVALID_CONFIG to indicate that an underlying operating system API call failed

F.2.2 Report_Application_Message(HMFM) Function

The *Report_Application_Message*(HMFM) method allows for a component to send a message to the HM fault handler which invokes the registered fault handler to process the message.

```
void FACE_HMFM_Report_Application_Message (
        /* in     */ FACE_HMFM_FAULT_MESSAGE_ADDRESS_TYPE fault,
        /* in     */ FACE_HMFM_FAULT_MESSAGE_SIZE_TYPE length,
        /*    out */ FACE_HMFM_RETURN_CODE_TYPE *return_code
    );
```

The *FACE_HMFM_Report_Application_Message* method is used by the software component to send a message to the HM function if it detects an erroneous behavior. This service may also be used to record an event for logging purposes. The response to the message is determined by the fault handler installed and the system configuration. The parameters to this method are as follows:

- *fault* – a message describing the fault
- *length* – the length of the fault message in bytes
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *FACE_HMFM_Report_Application_Message* is one of the following:

- FACE_HMFM_NO_ERROR to indicate successful completion of the operation
- FACE_HMFM_INVALID_PARAM to indicate the *length* parameter is invalid

F.2.3 Create_Fault_Handler(HMFM) Function

The *Create_Fault_Handler*(HMFM) method allows for a component to register a POSIX process- specific fault handler which is invoked in the event of POSIX process-level faults detected by the OS or component.

```
void FACE_HMFM_Create_Fault_Handler (
        /* in     */ FACE_HMFM_FAULT_HANDLER_ENTRY_TYPE entry_point,
        /* in     */ FACE_HMFM_STACK_SIZE_TYPE stack_size,
        /*    out */ FACE_HMFM_RETURN_CODE_TYPE *return_code
    );
```

The *FACE_HMFM_Create_Fault_Handler* method is used to create a fault handler thread. This thread may not be accessible by normal thread methods. The fault handler thread is an aperiodic thread with the highest priority and its priority cannot be modified. It cannot be suspended or stopped by other threads. The fault handler thread preempts any running thread independent of its priority or preemption mode.

The parameters to this method are as follows:

- *entry_point* – the entry point of the fault handler thread
- *stack_size* – the size of the fault handler's stack in bytes
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *FACE_HMFM_Create_Fault_Handler* is one of the following:

- FACE_HMFM_NO_ERROR to indicate successful completion of the operation
- FACE_HMFM_NO_ACTION to indicate that a fault handler has already been created
- FACE_HMFM_INVALID_CONFIG to indicate that the thread could not be created
- FACE_HMFM_INVALID_CONFIG to indicate that the stack_size parameter is invalid
- FACE_HMFM_INVALID_MODE to indicate that the system is in the incorrect mode to perform this operation

F.2.4 Get_Fault_Status(HMFM) Function

The *Get_Fault_Status*(HMFM) function allows for the fault handler registered by a component to obtain information regarding the current fault.

```
void FACE_HMFM_Get_Fault_Status (
        /*    out */ FACE_HMFM_FAULT_STATUS_TYPE *fault,
        /*    out */ FACE_HMFM_RETURN_CODE_TYPE *return_code
    );
```

The *FACE_HMFM_Get_Fault_Status* method is used by the fault handler to determine the fault type, faulty thread, the address at which the fault occurred, and the message associated with the fault. The parameters to this method are as follows:

- *fault* – upon return, contains a message describing the fault

- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *FACE_HMFM_Get_Fault_Status* is one of the following:

- FACE_HMFM_NO_ERROR to indicate successful completion of the operation
- FACE_HMFM_INVALID_CONFIG to indicate that the current thread is not the fault handler
- FACE_HMFM_NO_ACTION to indicate that there are no current faults

F.2.5 Raise_Application_Fault(HMFM) Function

The *Raise_Application_Fault*(HMFM) function allows for a component to indicate that a fault has occurred.

```
void FACE_HMFM_Raise_Application_Fault (
        /* in     */ FACE_HMFM_FAULT_CODE_TYPE code,
        /* in     */ FACE_HMFM_FAULT_MESSAGE_ADDRESS_TYPE message,
        /* in     */ FACE_HMFM_FAULT_MESSAGE_SIZE_TYPE length,
        /*    out */ FACE_HMFM_RETURN_CODE_TYPE *return_code
    );
```

The *FACE_HMFM_Raise_Application_Fault* method allows the current software component to indicate that a fault has occurred. The code, message, and length parameters are eventually passed to the installed fault handler for processing. The parameters to this method are as follows:

- *code* – contains an indication of the fault type
- *message* – references a message describing the fault
- *length* – the length of the fault message in bytes
- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *FACE_HMFM_Raise_Application_Fault* is one of the following:

- FACE_HMFM_NO_ERROR to indicate successful completion of the operation
- FACE_HMFM_INVALID_PARAM to indicate the length parameter is invalid
- FACE_HMFM_INVALID_PARAM when error code parameter is not FACE_HMFM_APPLICATION_ERROR

G FACE Configuration Interface

G.1 Introduction

The Configuration Services API provides a normalized interface to obtain configuration information from either a local or centralized configuration service in a portable manner. The Configuration Services API is part of the OSS.

Declarations are provided using an IDL syntax that is mapped to a Programming Language, as described in Section 4.14.

Note: The code in this document is formatted to align with the formatting constraints of the printed document.

G.2 Configuration Services API

FACE/Configuration.idl

```
//! Source file: FACE/Configuration.idl

#ifndef __FACE_CONFIGURATION
#define __FACE_CONFIGURATION

#include <FACE/Common.idl>

module FACE {

 //! This defines the Configuration Application Programming Interface.
 interface Configuration {

  //! This type is used to represent the handle used during a session
  //! with a configuration container.
  typedef long HANDLE_TYPE;

  //! This type is used to pass implementation specific initialization
  //! information to a Configuration API implementation.
  typedef STRING_TYPE INITIALIZATION_TYPE;

  //! This type is used to represent the name of a configuration
  //! container.
  //! The contents of a configuration container are accessed during a
  //! session.
  typedef STRING_TYPE CONTAINER_NAME_TYPE;

  //! This type is used to represent the name of a configuration set
  //! within a configuration container.
  typedef STRING_TYPE SET_NAME_TYPE;

  //! This type is used to represent the length of the buffer or
  //! amount of data returned by Read().
  typedef long BUFFER_SIZE_TYPE;
```

```
//! This type is used to represent the desired offset used with
//! Seek().
typedef long OFFSET_TYPE;

//! This type is used to represent the parameter to
//! Seek(). It indicates the manner in which the offset is to be
//! interpreted.
enum WHENCE_TYPE {
 //! This indicates that the offset value is to be interpreted as an
 //! offset from the beginning of the file. The offset should be a
 //! positive number and represent a position in the file.
 SEEK_FROM_START,
 //! This indicates that the offset value is to be interpreted as an
 //! offset from the current position in the file. The offset may be
 //! a positive or negative number to seek backward or forward in the
 //! in the file.
 SEEK_FROM_CURRENT,
 //! This indicates that the offset value is to be interpreted as an
 //! offset from the end of the file. The offset should be a
 //! negative number and represent how many bytes to backup.
 SEEK_FROM_END
};

//! The Initialize method is used to initialize the Configuration
//! implementation.
//!
//! @param[in] initialization_information provides implementation
//!   specific information which assists in the initialization of
//!   a Configuration API implementation.
//! @param[out] return_code contains a status code indicating success
//!
//! @return NO_ERROR is returned in @a return_code to indicate
//!   successful completion of the operation.
//! @return INVALID_PARAM to indicate that the @a return_code pointer
//!   (in appropriate languages) is invalid.
void Initialize
( in INITIALIZATION_TYPE  initialization_information,
 out RETURN_CODE_TYPE    return_code);

//! The @a Open method is used to establish a session with the
//! Configuration implementation.
//!
//! @param[in] container_name is the name of the configuration
//!   container to open a session with.
//! @param[out] handle contains a handle to be used on subsequent
//!   calls during this session.
//! @param[out] return_code contains a status code indicating success
//!   or failure
//!
//! @return NO_ERROR is returned in @a return_code to indicate
//!   successful completion of the operation.
//! @return INVALID_CONFIG is returned in @a return_code to indicate
//!   that the @a configuration container specified is invalid.
//! @return INVALID_PARAM is returned in @a return_code to indicate
//!   that the @a handle pointer (in appropriate languages) is
//!   invalid.
//! @return INVALID_MODE is returned in @a return_code to indicate
//!   the caller does not have permission to access the
//!   @a configuration container.
void Open
( in CONTAINER_NAME_TYPE container_name,
 out HANDLE_TYPE     handle,
 out RETURN_CODE_TYPE   return_code);
```

```
//! The @a Get_Size method is used to obtain the size of a particular
//! configuration set from the configuration container associated with
//! this session.
//!
//! @param[in] handle indicates the current session.
//! @param[in] set_name indicates the name of the configuration
//!     set to obtain the value of.
//! @param[out] size contains the size in bytes of the set.
//! @param[out] return_code contains a status code indicating success
//!     or failure.
//!
//! @return NO_ERROR is returned in @a return_code to indicate
//!     successful completion of the operation.
//! @return INVALID_CONFIG is returned in @a return_code to indicate
//!     that the @a handle is invalid.
//! @return INVALID_PARAM is returned in @a return_code to indicate
//!     that one of the pointer arguments (in the appropriate languages)
//!     is invalid.
//! @return NOT_AVAILABLE is returned in @a return_code to indicate
//!     that the size of the set is not available based on the backend
//!     media adapter used for this configuration information.
//!
//! @note For streaming configuration information sources, the
//!    @a set_name parameter should be set to "" or the empty string.
void Get_Size
( in  HANDLE_TYPE        handle,
  in  SET_NAME_TYPE      set_name,
  out BUFFER_SIZE_TYPE   size,
  out RETURN_CODE_TYPE   return_code);

//! The @a Read method is used to obtain configuration information
//! from the configuration container associated with this session.
//!
//! @param[in] handle indicates the current session.
//! @param[in] set_name indicates the name of the configuration
//!   set to obtain the value of.
//! @param[inout] buffer points to the buffer to be filled in with
//!   configuration information.
//! @param[in] buffer_size indicates the size of the @a buffer and
//!   the maximum number of bytes which can be returned.
//! @param[out] bytes_read contains the number of bytes read on
//!   a successful read.
//! @param[out] return_code contains a status code indicating success
//!   or failure
//!
//! @return NO_ERROR is returned in @a return_code to indicate
//!   successful completion of the operation.
//! @return INVALID_CONFIG is returned in @a return_code to indicate
//!   that the @a handle is invalid.
//! @return INVALID_PARAM is returned in @a return_code to indicate
//!   that one of the pointer arguments (in the appropriate
//!   languages) is invalid.
//! @return NOT_AVAILABLE is returned in @a return_code to indicate
//!   that the entire data stream associated with this configuration
//!   set has been read.
//!
//! @note For streaming configuration information sources, the
//!  @a set_name parameter should be set to "all"
void Read
( in  HANDLE_TYPE          handle,
  in  SET_NAME_TYPE        set_name,
  in  SYSTEM_ADDRESS_TYPE  buffer,
```

```
    in  BUFFER_SIZE_TYPE     buffer_size,
    out BUFFER_SIZE_TYPE     bytes_read,
    out RETURN_CODE_TYPE     return_code);

  //! The @a Seek method is used to set the current position indicator
  //! in the configuration session.
  //!
  //! @param[in] handle indicates the current session.
  //! @param[in] whence indicates how to interpret the offset parameter.
  //! @param[in] offset indicates the desired offset.
  //! @param[out] return_code contains a status code indicating success
  //!   or failure
  //!
  //! @return NO_ERROR is returned in @a return_code to indicate
  //!   successful completion of the operation.
  //! @return INVALID_PARAM is returned in @a return_code to indicate
  //!   that the whence @a parameter is invalid, the offset is invalid
  //!   for the specified value of @a whence, or that the
  //!   @a return_code pointer (in the appropriate languages)
  //!   is invalid.
  //!
  //! @note For some configuration media implementations, the @a Seek
  //!   operation may not be applicable.
  void Seek
  ( in HANDLE_TYPE    handle,
   in WHENCE_TYPE    whence,
   in OFFSET_TYPE    offset,
   out RETURN_CODE_TYPE return_code);

  //! The Close method is used to conclude a sequence of operations
  //! on a configuration handle. It frees any resources allocated
  //! during open based on the specific configuration being accessed.
  //!
  //! @param[in] handle is the session to close
  //! @param[out] return_code indicates success or failure
  //!
  //! @return NO_ERROR is returned in @a return_code to indicate
  //!   successful completion of the operation.
  //! @return INVALID_CONFIG is returned in @a return_code to indicate
  //!   that the @a handle is invalid.
  void Close
  ( in HANDLE_TYPE       handle,
   out RETURN_CODE_TYPE     return_code);

 };
};

#endif //! __FACE_CONFIGURATION
```

G.2.1 Initialize(CONFIG) Function

The *Initialize*(CONFIG) function is used to initialize the Configuration implementation.

```
/* IDL declaration */
module FACE
{
 interface Configuration
 {
  void Initialize
  ( in INITIALIZATION_TYPE  initialization_information,
   out RETURN_CODE_TYPE    return_code); };
};
```

The parameters to this method are as follows:

- *initialization_information* – provides implementation-specific information which assists in the initialization of a Configuration API implementation
- *return_code* – upon return contains a status code indicating success or failure

The *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Initialize*(CONFIG) is one of the following:

- NO_ERROR to indicate successful completion of the operation
- INVALID_CONFIG to indicate that the initialization_information parameter(in appropriate languages) is invalid, or does not identify a known configuration

G.2.2 Open(CONFIG) Function

The *Open*(CONFIG) function is used to establish a session with the Configuration implementation.

```
/* IDL declaration */
module FACE
{
 interface Configuration
 {
  void Open
  ( in CONTAINER_NAME_TYPE container_name,
   out HANDLE_TYPE     handle,
   out RETURN_CODE_TYPE   return_code);

 };
};
```

The parameters to this method are as follows:

- *container_name* – the name of the configuration container with which to open a session
- *handle* – upon return, contains an identifier to be used in future configuration operations upon this configuration container
- *return_code* – upon return contains a status code indicating success or failure

The *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Open*(CONFIG) is one of the following:

- NO_ERROR to indicate successful completion of the operation
- INVALID_CONFIG to indicate that the configuration container specified is invalid
- INVALID_PARAM to indicate that the handle or return_code pointer (in appropriate languages) is invalid
- INVALID_MODE to indicate that the caller does not have permission to access the configuration container

G.2.3 Get_Size(CONFIG) Function

The *Get_Size*(CONFIG) function is used to obtain used to obtain the size of a particular configuration set from the configuration container associated with this session.

```
/* IDL declaration */
module FACE
{
 interface Configuration
 {
  void Get_Size
  ( in  HANDLE_TYPE        handle,
    in  SET_NAME_TYPE      set_name,
    out BUFFER_SIZE_TYPE   size,
    out RETURN_CODE_TYPE   return_code);
 };
};
```

The parameters to this method are as follows:

- *handle* – specifies the configuration session
- *set_name* – indicates the name of the configuration set of which to obtain the size
- *bytes_read* – contains the number of bytes which would be read on a successful read
- *return_code* – upon return contains a status code indicating success or failure

The *handle* parameter contains the session handle returned by a previous call to *Open*(CONFIG).

The *set_name* parameter contains the name of the configuration element to obtain the value of or one of the following special configuration element names:

- "all" to indicate that the intent is to read all data from the configuration container as a stream

Upon successful return, the length of the requested configuration *set_name* is returned in *bytes_read* and indicates the number of octets in length.

The *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Get_Size*(CONFIG) is one of the following:

- NO_ERROR to indicate successful completion of the operation
- INVALID_CONFIG to indicate that the *handle* specified is invalid
- INVALID_PARAM to indicate that either the *buffer* or *return_status* pointer (in appropriate languages) is invalid
- NOT_AVAILABLE to indicate that the size of the set is not available based on the backend media adapter used for this configuration information

G.2.4 Read(CONFIG) Function

The *Read*(CONFIG) function is used to obtain configuration information from the configuration container associated with this session.

```
/* IDL declaration */
module FACE
{
 interface Configuration
 {
  void Read
  ( in  HANDLE_TYPE          handle,
    in  SET_NAME_TYPE        set_name,
    in  SYSTEM_ADDRESS_TYPE  buffer,
    in  BUFFER_SIZE_TYPE     buffer_size,
    out BUFFER_SIZE_TYPE     bytes_read,
    out RETURN_CODE_TYPE     return_code);
  };
};
```

The parameters to this method are as follows:

- *handle* – specifies the configuration session
- *set_name* – indicates the name of the configuration set of which to obtain the value
- *buffer* – points to the buffer to be filled in with configuration information
- *buffer_size* – indicates the size of the buffer and the maximum number of bytes which can be returned
- *bytes_read* – contains the number of bytes read on a successful read
- *return_code* – upon return contains a status code indicating success or failure

The *handle* parameter contains the session handle returned by a previous call to *Open*(CONFIG).

The *element* parameter contains the name of the configuration element to obtain the value of or one of the following special configuration element names:

- "all" to indicate that the intent is to read all data from the configuration container as a stream

Upon successful return, the memory specified by *buffer* contains the contents of the requested configuration *element*. This value is *bytes_read* octets in length. When reading the special *set_name* "all" to indicate that the data is to be read as a stream, it is possible that the entire contents cannot be read into the buffer provided. In the event, the size of the buffer provided is not large enough to contain the entire value and *bytes_read* is equal to *buffer_size* and subsequent *Read*() operations may be performed to obtain the remainder of the data. When there is no more data to obtain, *bytes_read* may contain zero or larger up to *buffer_size* and the *return_code* is set to FACE_NOT_AVAILABLE.

The *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Read*(CONFIG) is one of the following:

- NO_ERROR to indicate successful completion of the operation
- INVALID_CONFIG to indicate that the *handle* specified is invalid
- INVALID_PARAM to indicate that either the *buffer* or *return_status* pointer (in appropriate languages) is invalid

- NOT_AVAILABLE to indicate that there is no more data to be read in the configuration stream “all”; there may be zero or more bytes returned in this case

G.2.5 Seek(CONFIG) Function

The *Seek*(CONFIG) function is used to set the current position indicator for the specified configuration session.

```
/* IDL declaration */
module FACE
{
 interface Configuration
 {
  void Seek
  ( in HANDLE_TYPE    handle,
   in WHENCE_TYPE    whence,
   in OFFSET_TYPE    offset,
   out RETURN_CODE_TYPE return_code);
 };
};
```

The parameters to this method are as follows:

- *handle* – specifies the configuration session
- *whence* – indicates how the *offset* parameter is to be interpreted
- *offset* – indicates the desired offset within the configuration session
- *return_code* – upon return contains a status code indicating success or failure

The *whence* parameter is used to indicate whether the *offset* parameter is to be relative to the beginning of the configuration session, an arbitrary position, or relative to the end of the configuration session. The *whence* parameter is of an enumerated type which can have the following values:

- SEEK_FROM_START – indicates that the *offset* value is to be interpreted as an offset from the beginning of the file – the *offset* should be a positive number and represent a position in the file
- SEEK_FROM_CURRENT – indicates that the *offset* value is to be interpreted as an offset from the current position in the file – the *offset* may be a positive or negative number to seek backward or forward in the file
- SEEK_FROM_END – indicates that the *offset* value is to be interpreted as an offset from the end of the file – the *offset* should be a negative number and represent how many bytes to backup

The *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Seek*(CONFIG) is one of the following:

- NO_ERROR to indicate successful completion of the operation
- INVALID_CONFIG to indicate that the *handle* specified is invalid

- INVALID_PARAM to indicate that the *whence* parameter is invalid, the *offset* is invalid for the specified value of *whence*, or that the *return_code* pointer (in the appropriate languages) is invalid

Note: This method may not be supported by all underlying configuration media.

G.2.6 Close(CONFIG) Function

The *Close*(CONFIG) function is used to terminate a session with the Configuration implementation.

```
/* IDL declaration */
module FACE
{
 interface Configuration
 {
  void Close
  ( in HANDLE_TYPE       handle,
   out RETURN_CODE_TYPE     return_code);
 };
};
```

The parameters to this method are as follows:

- *handle* – upon return, contains an identifier to be used in future configuration operations upon this configuration container
- *return_code* – upon return contains a status code indicating success or failure

The *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Close*(CONFIG) is one of the following:

- NO_ERROR to indicate successful completion of the operation
- INVALID_CONFIG to indicate that the *handle* specified is invalid

H Graphics

H.1 Introduction

Graphics Services provide normalized interfaces for PSSS Graphics UoCs and other UoCs providing graphics capabilities. Composition of multiple graphics contexts within a multi-threaded Embedded Graphics Library (EGL) system is enabled by the compositor extension. Cockpit Display Systems (CDS) and User Applications (UA) use the ARINC 661 standardized interface protocols. Adding Graphics UoCs with minimal integration is enabled by Display Management.

H.2 Graphics – A661_Conformance.xsd

The following XSD defines the style data configuration schema for ARINC 661.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">

  <xs:simpleType name="a661_byte">
   <xs:restriction base="xs:integer">
    <xs:minInclusive value="0"/>
    <xs:maxInclusive value="255"/>
   </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="a661_ushort">
   <xs:restriction base="xs:integer">
    <xs:minInclusive value="0"/>
    <xs:maxInclusive value="65535"/>
   </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="colorReference">
   <xs:union memberTypes="a661_byte xs:string" />
  </xs:simpleType>

 <xs:complexType name="textureEntry" mixed="true">
  <xs:attribute name="name"  type="xs:string" />
  <xs:attribute name="ID"   type="a661_ushort" use="required"/>
  <xs:attribute name="stride" type="xs:float"   use="required" />
 </xs:complexType>

 <xs:complexType name="stippleReference">
  <xs:attribute name="name"  type="xs:string" />
  <xs:attribute name="index" type="xs:integer"  use="required"/>
  <xs:attribute name="scale" type="xs:float"   use="required"/>
 </xs:complexType>

 <xs:complexType name="colorTableEntry">
  <xs:sequence>
   <xs:element name="RGBA">
    <xs:complexType>
```

```
    <xs:attribute name="r" type="a661_byte" use="required"/>
    <xs:attribute name="g" type="a661_byte" use="required"/>
    <xs:attribute name="b" type="a661_byte" use="required"/>
    <xs:attribute name="a" type="a661_byte"/>
   </xs:complexType>
  </xs:element>
  <xs:element name="intent"     type="xs:string"    minOccurs="0"/>
 </xs:sequence>
 <xs:attribute name="ID"       type="a661_byte"    use="required"/>
 <xs:attribute name="intent"     type="xs:string"/>
 <xs:attribute name="name"      type="xs:string"/>
</xs:complexType>

<xs:complexType name="fillStyle">
 <xs:sequence>
  <xs:element name="lineStyle"   type="lineStyle"/>
  <!--integrator only-->
  <xs:element name="textureFlags"  type="textureEntry"/>
  <xs:element name="fillHints"   type="xs:string"    minOccurs="0"/>
 </xs:sequence>
 <xs:attribute name="ID"       type="a661_ushort"/>
 <xs:attribute name="name"      type="xs:string"/>
</xs:complexType>

<xs:complexType name="lineStyle">
 <xs:sequence>
  <xs:element name="stippleRef"   type="stippleReference" minOccurs="0"/>
  <xs:element name="textureFlags"  type="textureEntry"   minOccurs="0"/>
  <xs:element name="lineHints"   type="xs:string"   minOccurs="0"/>
 </xs:sequence>
 <xs:attribute name="ID"       type="a661_ushort"   use="required"/>
 <xs:attribute name="name"      type="xs:string"/>
 <xs:attribute name="width"     type="xs:float"     use="required"/>
 <xs:attribute name="endCap"     type="xs:boolean"    default="false" />
</xs:complexType>

<xs:complexType name="textureTableEntry">
 <xs:sequence>
  <xs:element name="filepath"    type="xs:string"/>
  <xs:element name="description"  type="xs:string" minOccurs="0"/>
 </xs:sequence>
 <xs:attribute name="ID"       type="xs:integer" use="required"/>
 <xs:attribute name="name"      type="xs:string"/>
 <xs:attribute name="width"     type="xs:integer" use="required"/>
 <xs:attribute name="height"     type="xs:integer" use="required"/>
</xs:complexType>

<xs:complexType name="pictureTableEntry">
 <xs:sequence>
  <xs:element name="filepath"    type="xs:string"/>
  <xs:element name="description"  type="xs:string"  minOccurs="0"/>
 </xs:sequence>
 <xs:attribute name="ID"       type="a661_ushort" use="required"/>
 <xs:attribute name="name"      type="xs:string"/>
</xs:complexType>

<xs:complexType name="lineStippleTableEntry">
 <xs:attribute name="ID"       type="xs:integer" use="required"/>
 <xs:attribute name="name"      type="xs:string"/>
 <xs:attribute name="pattern"    type="xs:string" use="required"/>
 <xs:attribute name="halo"      type="xs:boolean" default="false"/>
</xs:complexType>
```

```
 <xs:complexType name="labelStyleSet">
   <xs:attribute name="ID"       type="a661_ushort"/>
   <xs:attribute name="name"      type="xs:string"/>
   <xs:attribute name="backgroundColor" type="colorReference"/>
 </xs:complexType>

 <xs:complexType name="fontTableEntry" mixed="true">
  <xs:attribute name="ID"       type="a661_byte"/>
  <xs:attribute name="name"      type="xs:string"/>
  <xs:attribute name="size"      type="xs:integer"/>
 </xs:complexType>

 <xs:complexType name="mapItemStyleEntry">
  <xs:attribute name="ID"       type="xs:integer"   use="required"/>
  <xs:attribute name="name"      type="xs:string"   use="required"/>
  <xs:attribute name="fontRef"    type="a661_byte"   use="required"/>
  <xs:attribute name="lineStyleRef"  type="a661_ushort"  use="required"/>
  <xs:attribute name="fillStyleRef"  type="a661_ushort"  use="required"/>
  <xs:attribute name="colorRef"    type="colorReference" use="required"/>
  <xs:attribute name="labelStyleRef" type="a661_ushort"  use="required"/>
  <xs:attribute name="halo"      type="xs:boolean"   default="false"/>
 </xs:complexType>

 <xs:element name="configuration">
  <xs:complexType>
   <xs:all>
    <xs:element name="metadata" minOccurs="0">
     <xs:complexType>
     <xs:sequence>
      <xs:any minOccurs="0" />
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="constants">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="constant" maxOccurs="unbounded">
        <xs:complexType>
         <xs:attribute name="name" type="xs:string"/>
         <xs:attribute name="value" type="xs:string"/>
        </xs:complexType>
       </xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="colortable" minOccurs="0">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="color" type="colorTableEntry" minOccurs="0"
maxOccurs="256"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="fontTable" minOccurs="0">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="font" type="fontTableEntry" minOccurs="0"
maxOccurs="256"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="fillStyleTable" minOccurs="0">
     <xs:complexType>
```

```
      <xs:sequence>
       <xs:element name="fillStyle" type="fillStyle" minOccurs="0"
maxOccurs="65536"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="lineStyleTable" minOccurs="0">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="lineStyle" type="lineStyle" minOccurs="0"
maxOccurs="65536"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="textureTable" minOccurs="0">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="texture" type="textureTableEntry" minOccurs="0"
maxOccurs="65536"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="lineStippleTable" minOccurs="0">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="lineStipple" type="lineStippleTableEntry"
minOccurs="0" maxOccurs="65536"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="labelStyleTable" minOccurs="0">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="labelStyle" type="labelStyleSet" minOccurs="0"
maxOccurs="65535" />
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="mapItemStyleTable" minOccurs="0">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="itemStyle" type="mapItemStyleEntry" minOccurs="0"
maxOccurs="65536"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="pictureTable" minOccurs="0">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="picture" type="pictureTableEntry" minOccurs="0"
maxOccurs="65536"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:all>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

H.3 Graphics – DisplayManagement.xsd

The following XSD defines the configuration schema for display management.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">

  <xs:simpleType name="a661_byte">
   <xs:restriction base="xs:integer">
    <xs:minInclusive value="0"/>
    <xs:maxInclusive value="255"/>
   </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="a661_ushort">
   <xs:restriction base="xs:integer">
    <xs:minInclusive value="0"/>
    <xs:maxInclusive value="65535"/>
   </xs:restriction>
  </xs:simpleType>

 <xs:simpleType name="Units">
  <xs:restriction base="xs:string">
  <xs:enumeration value="inch"/>
  <xs:enumeration value="mm"/>
  <xs:enumeration value="screen"/>
  <xs:enumeration value="pixel"/>
  </xs:restriction>
 </xs:simpleType>

 <xs:complexType name="Rectangle">
  <xs:attribute name="x"         type="xs:integer" use="required"/>
  <xs:attribute name="y"         type="xs:integer" use="required"/>
  <xs:attribute name="width"       type="xs:integer" use="required"/>
  <xs:attribute name="height"       type="xs:integer" use="required"/>
 </xs:complexType>

 <xs:complexType name="Scale">
  <xs:attribute name="xScale"       type="xs:decimal" use="required"/>
  <xs:attribute name="yScale"       type="xs:decimal" use="required"/>
  <xs:attribute name="baseUnit"      type="Units"   default="screen" />
  <xs:attribute name="perUnit"      type="Units"   default="pixel" />
 </xs:complexType>

 <xs:complexType name="Size">
  <xs:attribute name="width"       type="xs:integer" use="required"/>
  <xs:attribute name="height"       type="xs:integer" use="required"/>
  <xs:attribute name="units"       type="Units"   default="pixel"/>
 </xs:complexType>

 <xs:complexType name="UserApplication">
  <xs:anotation>

 </xs:anotation>
  <xs:attribute name="applicationId"   type="xs:integer" use="required" />
  <xs:attribute name="dFPath"       type="xs:string" use="required" />
  <xs:attribute name="mapSourceLayer"   type="xs:boolean" default="false"/>
  <xs:attribute name="styleFilePath"   type="xs:string" />
  <xs:attribute name="visible"      type="xs:boolean" default="false"/>
  <xs:attribute name="window"       type="xs:integer" user="required"/>
 </xs:complexType>
```

```
<xs:complexType name="Window">
 <xs:anotation>

 </xs:anotation>
 <xs:sequence>
  <xs:element name="description"    type="xs:string" />
  <xs:element name="pixelArea"     type="Rectangle" />
  <xs:element name="scalingFactor"   type="Scale" />
 </xs:sequence>
 <xs:attribute name="id"         type="xs:integer" use="required"/>
 <xs:attribute name="name"        type="xs:string"/>
</xs:complexType>

<xs:complexType name="Screen">
 <xs:anotation>

 </xs:anotation>
 <xs:sequence>
  <xs:element name="pixelSize"      type="Size" >

  </xs:element>
  <xs:element name="physicalDimensions"  type="Size" >

  </xs:element>
  <xs:element name="layout"        maxOccurs="65535" >
   <xs:anotation>

   </xs:anotation>
   <xs:complexType>
    <xs:sequence>
     <xs:element name="window"     type="Window"  maxOccurs="65535" />
    </xs:sequence>
    <xs:attribute name="id"       type="xs:integer" use="required"/>
   </xs:complexType>
  </xs:element>
 </xs:sequence>
 <xs:attribute name="id"         type="xs:integer" use="required"/>
</xs:complexType>

<xs:complexType name="ExternalSource" >
 <xs:anotation>

 </xs:anotation>
 <xs:sequence>
  <xs:any minOccurs="0" maxOccurs="65535"/>
 </xs:sequence>
 <xs:attribute name="id"         type="a661_ushort" use="required"/>
 <xs:attribute name="name"        type="xs:string" />
 <xs:attribute name="type"        type="xs:string" />
</xs:complexType>

<xs:element name="size" type="Size" />
<xs:element name="properties">
 <xs:anotation>

 </xs:anotation>
 <xs:complexType>
  <xs:attribute name="path"        type="xs:string" />
  <xs:attribute name="input"       type="xs:string" />
 </xs:complexType>
</xs:element>

<xs:element name="configuration">
```

```
  <xs:anotation>
  This is the root element of the configuration file.
  </xs:anotation>
  <xs:complexType>
   <xs:sequence>
    <xs:element name="metaData"     type="xs:string" minOccurs="0" />
    <xs:element name="screen"      type="Screen" maxOccurs="65535"/>
    <xs:element name="ua"        type="UserApplication" maxOccurs="65535" />
    <xs:element name="externalSource"  type="ExternalSource" maxOccurs="65535"
/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

H.3.1 UserApplication

The UserApplication element is used to define which Definition Files (DF) or UAs are placed into which windows. It also defines which DFs are visible by default and the configuration file which defines the style tables used for that DF file, including color tables, style set parameters, and other similar attributes. The application ID attribute is used to specify the UA ID as specified in the ARINC 661 standard, and it helps to make sure that the UA ID is unique in a complex system.

H.3.2 Window

The Window element defines a section of the screen in which one or more DFs are placed with their origin at the bottom left of the window area. More information on the concept of a window is in the ARINC 661 standard.

H.3.3 Screen

The Screen element defines a physical display surface which has both physical dimensions and a defined pixel size. Within a screen a number of windows are defined in the ARINC 661 server's screen space. Within each screen space an ARINC 661 DF is used to define what is in that area of the screen.

The "id" attribute is used by the EGL layer to define to which display output this information refers.

H.3.4 pixelSize

The pixelSize element is used to let the ARINC 661 server know how many pixels are in the area. It simply has a width and height attribute of the rectangular area of the screen. The assumption that the screen is rectangular is being made here as a general case. If the physical screen can be expressed as a set of rectangles it is recommended that multiple screen elements are used.

H.3.5 physicalDimensions

The physicalDimensions element defines the physical size of the screen. This is used to determine the default scaling factor of all ARINC 661 applications to work as specified by the ARINC 661 standard which defined the 0.01 mm = 1 ARINC 661 display unit. It can also be used to supply the physical dimensions to the EGL layer used by the OpenGL applications for their use.

H.3.6 Layout

The Layout element is used to define a layout of windows. This allows for the server to be able to change between different layouts as needed.

H.3.7 ExternalSource

The ExternalSource element defines the external sources which are in the system. The "id" attribute maps directly to the ARINC 661 external source reference attribute of the *ExternalSourceWidget*. The type attribute defines which external source driver is used for the external source; these values are specific to the hardware being used. The name attribute is simply a free string to help the writer identify the external source.

H.3.8 Properties

The Properties element is a set of name, value pairs that are used by the external source driver to configure the driver for proper use to provide the requested external source data.

I Injectable Interface

I.1 Introduction

During startup, each UoC in a memory address space requires a reference to the IDL interfaces it uses during run-time. The Injectable Interface provides a basic *Set_Reference* interface for an external executive to provide references to UoCs. The instantiations make each *Set_Reference* function unique by the type of Injectable Interface. If more than one reference to an interface is used, the *Set_Reference* is called for each instance required.

Declarations are provided using an IDL syntax that is mapped to a Programming Language, as described in Section 4.14.

Note: The code in this document is formatted to align with the formatting constraints of the printed document.

I.2 FACE_Injectable Interface Specification

```
//! Source file: FACE/Injectable.idl

#ifndef __FACE_INJECTABLE
#include <FACE/Common.idl>

module FACE {
  module Injectable<interface INTERFACE_TYPE> {
    interface Injectable {
      //! The Set_Reference(Injectable) function assigns an instance
      //! of an interface provider specified by interface_reference.
      //!
      //! Note: The interface_reference parameter is semantically an
      //!       in parameter but is inout to avoid an undesirable
      //!       mapping in C++.
      void Set_Reference (
        in    STRING_TYPE      interface_name,
        inout INTERFACE_TYPE   interface_reference,
        in    GUID_TYPE        id,
        out   RETURN_CODE_TYPE return_code);
    };
  };
};
```

The parameters to *Set_Reference* are as follows:

- *interface_name* – a human-readable name for the instance of the interface being provided – supports configuration of different instances of interfaces used by a component
- *interface_reference* – a reference to the specific interface appropriate for the UoC
- *id* – a UUID; to delineate the interface associated with a specific data modeled message or provide computer readable ID for an interface instance

- *return_code* – upon return, contains a status code indicating success or failure

Upon return, the *return_code* output parameter contains a value indicating that the method executed successfully or failed for a specific reason.

The return code value returned from *Set_Reference* is one of the following:

- NO_ERROR to indicate the operation was successful
- INVALID_PARAM to indicate the *reference_interface* parameter supplied is null or not in range
- NO_ACTION to indicate the reference is a duplicate to one already set
- INVALID_MODE to indicate a *Set_Reference* call was received while in steady state
- INVALID_CONFIG to indicate a new reference is being set that has one or more parameters that creates an invalid request, such as a new name is provided for a reference that was previously provided
- NOT_AVAILABLE to indicate the calling function cannot set more than one reference of this interface type for this UoC

I.3 Explicit Injectable Declarations

This section provides template module instantiations for FACE Interfaces that do not vary by type to ensure that the corresponding Injectable Interface signatures are consistent for all UoCs.

Three FACE Interfaces do vary by type and thus cannot be listed: TS Typed Interface, TS Typed-Extended Interface, and LCM Stateful Interface.

I.3.1 FACE::Configuration_Injectable

```
// Source file: FACE/Configuration_Injectable.idl

#ifndef __FACE_CONFIGURATION_INJECTABLE
#define __FACE_CONFIGURATION_INJECTABLE

#include <FACE/Injectable.idl>
#include <FACE/Configuration.idl>

module FACE {

  // Template module instantiation for Configuration Interface
  module Injectable<
    FACE::Configuration
  > Configuration_Injectable;

}; // module FACE

#endif // __FACE_CONFIGURATION_INJECTABLE
```

I.3.2 FACE::IOSS::Analog::IO_Service_Injectable

```
// Source file: FACE/IOSS/Analog_Injectable.idl

#ifndef __FACE_IOSS_ANALOG_INJECTABLE
```

```
#define __FACE_IOSS_ANALOG_INJECTABLE

#include <FACE/Injectable.idl>
#include <FACE/IOSS/Analog.idl>

module FACE {
  module IOSS {
    module Analog {

      // Template module instantiation for Analog I/O Service Interface
      module Injectable<
        FACE::IOSS::Analog::IO_Service
      > IO_Service_Injectable;

    }; // module Analog
  }; // module IOSS
}; // module FACE

#endif // __FACE_IOSS_ANALOG_INJECTABLE
```

I.3.3 FACE::IOSS::ARINC429::IO_Service_Injectable

```
// Source file: FACE/IOSS/ARINC429_Injectable.idl

#ifndef __FACE_IOSS_ARINC429_INJECTABLE
#define __FACE_IOSS_ARINC429_INJECTABLE

#include <FACE/Injectable.idl>
#include <FACE/IOSS/ARINC429.idl>

module FACE {
  module IOSS {
    module ARINC429 {

      // Template module instantiation for ARINC 429 I/O Service Interface
      module Injectable<
        FACE::IOSS::ARINC429::IO_Service
      > IO_Service_Injectable;

    }; // module ARINC429
  }; // module IOSS
}; // module FACE

#endif // __FACE_IOSS_ARINC429_INJECTABLE
```

I.3.4 FACE::IOSS::ARINC825::IO_Service_Injectable

```
// Source file: FACE/IOSS/ARINC825_Injectable.idl

#ifndef __FACE_IOSS_ARINC825_INJECTABLE
#define __FACE_IOSS_ARINC825_INJECTABLE

#include <FACE/Injectable.idl>
#include <FACE/IOSS/ARINC825.idl>

module FACE {
  module IOSS {
    module ARINC825 {

      // Template module instantiation for ARINC 825 I/O Service Interface
      module Injectable<
        FACE::IOSS::ARINC825::IO_Service
      > IO_Service_Injectable;
```

```
    }; // module ARINC825
  }; // module IOSS
}; // module FACE

#endif // __FACE_IOSS_ARINC825_INJECTABLE
```

I.3.5 FACE::IOSS::Discrete::IO_Service_Injectable

```
// Source file: FACE/IOSS/Discrete_Injectable.idl

#ifndef __FACE_IOSS_DISCRETE_INJECTABLE
#define __FACE_IOSS_DISCRETE_INJECTABLE

#include <FACE/Injectable.idl>
#include <FACE/IOSS/Discrete.idl>

module FACE {
  module IOSS {
    module Discrete {

      // Template module instantiation for Discrete I/O Service Interface
      module Injectable<
        FACE::IOSS::Discrete::IO_Service
      > IO_Service_Injectable;

    }; // module Discrete
  }; // module IOSS
}; // module FACE

#endif // __FACE_IOSS_DISCRETE_INJECTABLE
```

I.3.6 FACE::IOSS:Generic::IO_Service_Injectable

```
// Source file: FACE/IOSS/Generic_Injectable.idl

#ifndef __FACE_IOSS_GENERIC_INJECTABLE
#define __FACE_IOSS_GENERIC_INJECTABLE

#include <FACE/Injectable.idl>
#include <FACE/IOSS/Generic.idl>

module FACE {
  module IOSS {
    module Generic {

      // Template module instantiation for Generic I/O Service Interface
      module Injectable<
        FACE::IOSS::Generic::IO_Service
      > IO_Service_Injectable;

    }; // module Generic
  }; // module IOSS
}; // module FACE

#endif // __FACE_IOSS_GENERIC_INJECTABLE
```

I.3.7 FACE::IOSS::I2C::Combined_RW_IO_Service_Injectable

```
// Source file: FACE/IOSS/I2C_Injectable.idl

#ifndef __FACE_IOSS_I2C_INJECTABLE
```

```
#define __FACE_IOSS_I2C_INJECTABLE

#include <FACE/Injectable.idl>
#include <FACE/IOSS/I2C.idl>

module FACE {
  module IOSS {
    module I2C {

      // Template module instantiation for I2C I/O Service Interface
      module Injectable<
        FACE::IOSS::I2C::Combined_RW_IO_Service
      > Combined_RW_IO_Service_Injectable;

    }; // module I2C
  }; // module IOSS
}; // module FACE

#endif // __FACE_IOSS_I2C_INJECTABLE
```

I.3.8 FACE::IOSS::M1553::IO_Service_Injectable

```
// Source file: FACE/IOSS/M1553_Injectable.idl

#ifndef __FACE_IOSS_M1553_INJECTABLE
#define __FACE_IOSS_M1553_INJECTABLE

#include <FACE/Injectable.idl>
#include <FACE/IOSS/M1553.idl>

module FACE {
  module IOSS {
    module M1553 {

      // Template module instantiation for MIL-STD-1553 I/O Service Interface
      module Injectable<
        FACE::IOSS::M1553::IO_Service
      > IO_Service_Injectable;

    };
  }; // module IOSS
}; // module FACE

#endif // __FACE_IOSS_M1553_INJECTABLE
```

I.3.9 FACE::IOSS::PrecisionSynchro::IO_Service_Injectable

```
// Source file: FACE/IOSS/PrecisionSynchro_Injectable.idl

#ifndef __FACE_IOSS_PRECISIONSYNCHRO_INJECTABLE
#define __FACE_IOSS_PRECISIONSYNCHRO_INJECTABLE

#include <FACE/Injectable.idl>
#include <FACE/IOSS/PrecisionSynchro.idl>

module FACE {
  module IOSS {
    module PrecisionSynchro {

      // Template module instantiation for Precision Synchro
      // I/O Service Interface
      module Injectable<
        FACE::IOSS::PrecisionSynchro::IO_Service
```

```
      > IO_Service_Injectable;

    }; // module PrecisionSynchro
  }; // module IOSS
}; // module FACE

#endif // __FACE_IOSS_PRECISIONSYNCHRO_INJECTABLE
```

I.3.10 FACE::IOSS::Serial::IO_Service_Injectable

```
// Source file: FACE/IOSS/Serial_Injectable.idl

#ifndef __FACE_IOSS_SERIAL_INJECTABLE
#define __FACE_IOSS_SERIAL_INJECTABLE

#include <FACE/Injectable.idl>
#include <FACE/IOSS/Serial.idl>

module FACE {
  module IOSS {
    module Serial {

      // Template module instantiation for Serial I/O Service Interface
      module Injectable<
        FACE::IOSS::Serial::IO_Service
      > IO_Service_Injectable;

    }; // module Serial
  }; // module IOSS
}; // module FACE

#endif // __FACE_IOSS_SERIAL_INJECTABLE
```

I.3.11 FACE::IOSS::Synchro::IO_Service_Injectable

```
// Source file: FACE/IOSS/Synchro_Injectable.idl

#ifndef __FACE_IOSS_SYNCHRO_INJECTABLE
#define __FACE_IOSS_SYNCHRO_INJECTABLE

#include <FACE/Injectable.idl>
#include <FACE/IOSS/Synchro.idl>

module FACE {
  module IOSS {
    module Synchro {

      // Template module instantiation for Synchro I/O Service Interface
      module Injectable<
        FACE::IOSS::Synchro::IO_Service
      > IO_Service_Injectable;

    }; // module Synchro
  }; // module IOSS
}; // module FACE

#endif // __FACE_IOSS_SYNCHRO_INJECTABLE
```

I.3.12 FACE::LCM::Configurable::ConfigurableInstance_Injectable

```
// Source file: FACE/LCM/Configurable_Injectable.idl
```

```
#ifndef __FACE_CONFIGURABLE_INJECTABLE
#define __FACE_CONFIGURABLE_INJECTABLE

#include <FACE/Injectable.idl>
#include <FACE/LCM/Configurable.idl>

module FACE {
  module LCM {
    module Configurable {

      // Template module instantiation for Configurable Interface
      module Injectable<
        FACE::LCM::Configurable::ConfigurableInstance
      > ConfigurableInstance_Injectable;

    }; // module Configurable
  }; // module LCM
}; // module FACE

#endif // __FACE_CONFIGURABLE_INJECTABLE
```

I.3.13 FACE::LCM::Connectable::ConnectableInstance_Injectable

```
// Source file: FACE/LCM/Connectable_Injectable.idl

#ifndef __FACE_LCM_CONNECTABLE_INJECTABLE
#define __FACE_LCM_CONNECTABLE_INJECTABLE

#include <FACE/Injectable.idl>
#include <FACE/LCM/Connectable.idl>

module FACE {
  module LCM {
    module Connectable {

      // Template module instantiation for Connectable Interface
      module Injectable<
        FACE::LCM::Connectable::ConnectableInstance
      > ConnectableInstance_Injectable;

    }; // module Connectable
  }; // module LCM
}; // module FACE

#endif // __FACE_LCM/CONNECTABLE_INJECTABLE
```

I.3.14 FACE::LCM::Initializable::InitializableInstance_Injectable

```
// Source file: FACE/LCM/Initializable_Injectable.idl

#ifndef __FACE_LCM_INITIALIZABLE_INJECTABLE
#define __FACE_LCM_INITIALIZABLE_INJECTABLE

#include <FACE/Injectable.idl>
#include <FACE/LCM/Initializable.idl>

module FACE {
  module LCM {
    module Initializable {

      // Template module instantiation for Initializable Interface
      module Injectable<
        FACE::LCM::Initializable::InitializableInstance
```

```
      > InitializableInstance_Injectable;

    }; // module Initializable
  }; // module LCM
}; // module FACE

#endif // __FACE_LCM/INITIALIZABLE_INJECTABLE
```

I.3.15 FACE::TSS::Base_Injectable

```
// Source file: FACE/TSS/Base_Injectable.idl

#ifndef __FACE_TSS_BASE_INJECTABLE
#define __FACE_TSS_BASE_INJECTABLE

#include <FACE/Injectable.idl>
#include <FACE/TSS/Base.idl>

module FACE {
  module TSS {

    // Template module instantiation for Base Interface
    module Injectable<
        FACE::TSS::Base
    > Base_Injectable;

  }; // module TSS
}; // module FACE

#endif // __FACE_TSS_BASE_INJECTABLE
```

I.3.16 FACE::TSS::CSP::CSP_Injectable

```
// Source file: FACE/TSS/CSP_Injectable.idl

#ifndef __FACE_TSS_CSP_INJECTABLE
#define __FACE_TSS_CSP_INJECTABLE

#include <FACE/Injectable.idl>
#include <FACE/TSS/CSP.idl>

module FACE {
  module TSS {
    module CSP {

      // Template module instantiation for Component State Persistence
Interface
      module Injectable<
        FACE::TSS::CSP::CSP
      > CSP_Injectable;

    }; // module CSP
  }; // module TSS
}; // module FACE

#endif // __FACE_TSS_CSP_INJECTABLE
```

I.3.17 FACE::TSS::Serialization_Injectable

```
// Source file: FACE/TSS/Serialization_Injectable.idl

#ifndef __FACE_TSS_SERIALIZATION_INJECTABLE
```

```
#define __FACE_TSS_SERIALIZATION_INJECTABLE

#include <FACE/Injectable.idl>
#include <FACE/TSS/Serialization.idl>

module FACE {
  module TSS {

    // Template module instantiation for Serialization Interface
    module Injectable<
        FACE::TSS::Serialization
      > Serialization_Injectable;

  }; // module TSS
}; // module FACE

#endif // __FACE_TSS_SERIALIZATION_INJECTABLE
```

I.3.18 FACE::TSS::TPM::TPMTS_Injectable

```
// Source file: FACE/TSS/TPM_Injectable.idl

#ifndef __FACE_TSS_TPM_INJECTABLE
#define __FACE_TSS_TPM_INJECTABLE

#include <FACE/Injectable.idl>
#include <FACE/TSS/TPM.idl>

module FACE {
  module TSS {
    module TPM {

      // Template module instantiation for Transport Protocol Module Interface
      module Injectable<
        FACE::TSS::TPM::TPMTS
      > TPMTS_Injectable;

    }; // module TPM
  }; // module TSS
}; // module FACE

#endif // __FACE_TSS_TPM_INJECTABLE
```

I.3.19 FACE::TSS::TypeAbstraction::TypeAbstractionTS_Injectable

```
// Source file: FACE/TSS/TypeAbstraction_Injectable.idl

#ifndef __FACE_TSS_TYPEABSTRACTION_INJECTABLE
#define __FACE_TSS_TYPEABSTRACTION_INJECTABLE

#include <FACE/Injectable.idl>
#include <FACE/TSS/TypeAbstraction.idl>

module FACE {
  module TSS {
    module TypeAbstraction {

      // Template module instantiation for Type Abstraction Interface
      module Injectable<
        FACE::TSS::TypeAbstraction::TypeAbstractionTS
      > TypeAbstractionTS_Injectable;

    }; // module TypeAbstraction
```

```
  }; // module TSS
}; // module FACE
```

J FACE Data Model Language

J.1 Introduction

The FACE Data Model Language is defined by a Meta-Object Facility (MOF) metamodel. This appendix includes the EMOF XMI representation of the metamodel and serves as the normative version of the FACE Data Model Language. In addition to the EMOF XMI, this appendix includes normative OCL constraints which also define the FACE Data Model Language. The FACE Data Model Language is described in detail in Section 3.3.2, Section 4.9.1, and Section J.2. Section J.2 includes descriptions of the elements in the metamodel. These descriptions have been removed from the included EMOF XMI.

J.2 Language Description

The following sections provide descriptive detail to aid in understanding the normative specification of the metamodel in Section J.4.

J.2.1 Meta-Package: face

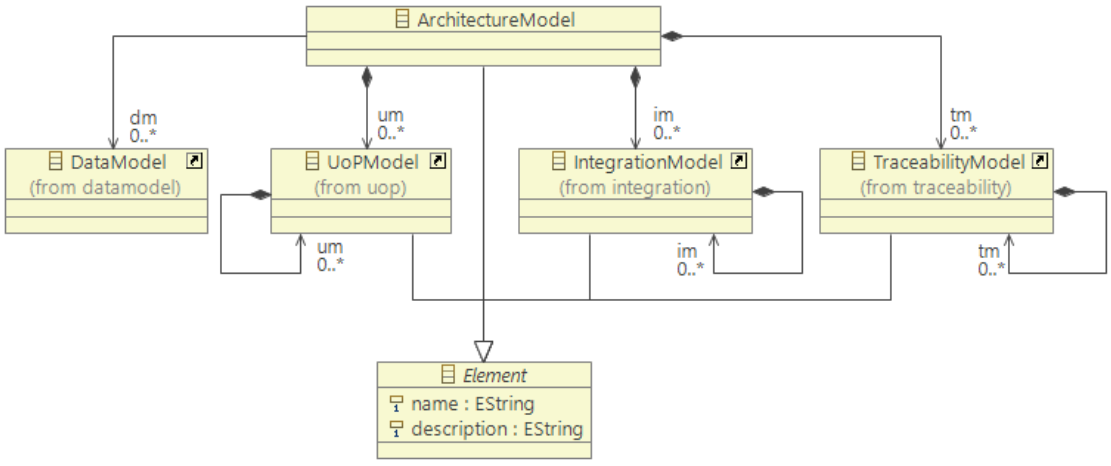

Figure 27: FACE Metamodel "face" Package

J.2.1.1 Meta-Class: face.ArchitectureModel

Description

An ArchitectureModel is a container for DataModels, UoPModels, IntegrationModels, and TraceabilityModels. The relationships for the ArchitectureModel meta-class are listed in Table 42.

Table 42: face.ArchitectureModel Relationships

Type	Name	Target	Multiplicity
Composition	dm	datamodel.DataModel	0..*
Composition	um	face.uop.UoPModel	0..*
Composition	im	face.integration.IntegrationModel	0..*
Composition	tm	face.traceability.TraceabilityModel	0..*
Generalization		Element	

J.2.1.2 Meta-Class: face.Element

Description

An Element is the root type for defining all named elements in the ArchitectureModel. The "name" attribute captures the name of the Element in the model. The "description" attribute captures a description for the element. The attributes for the Element meta-class are listed in Table 43.

Table 43: face.Element Attributes

Name	Type	Multiplicity
name	string	1
description	string	1

J.2.2 Meta-Package: face.uop

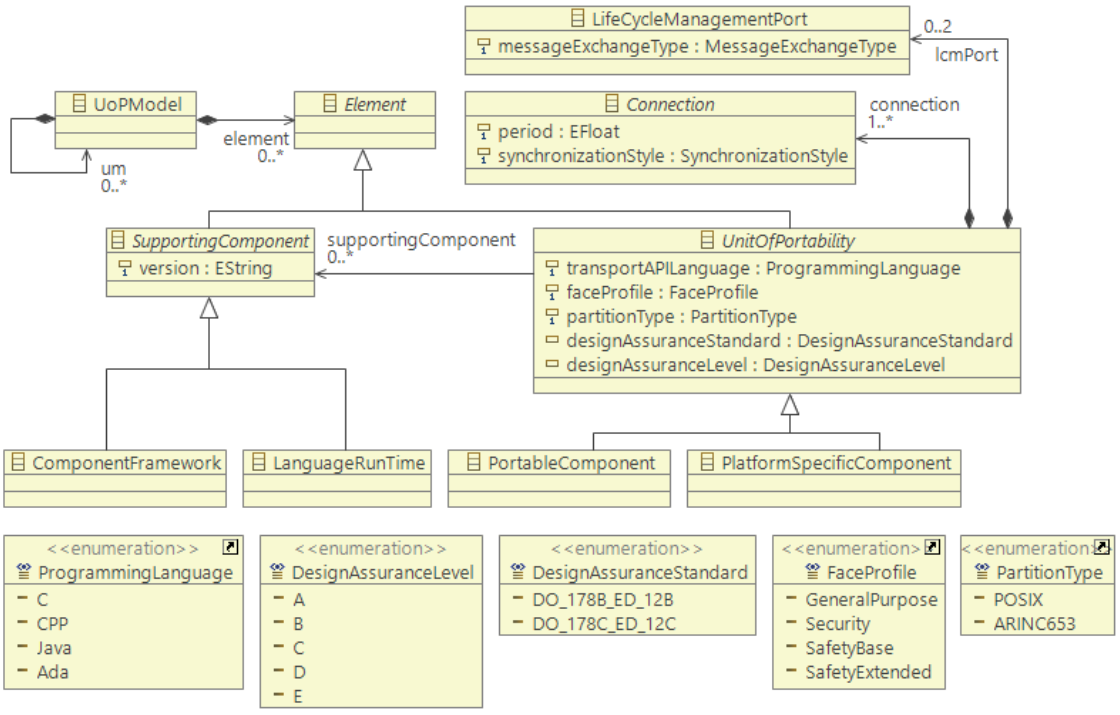

Figure 28: FACE Metamodel "face.uop" Package

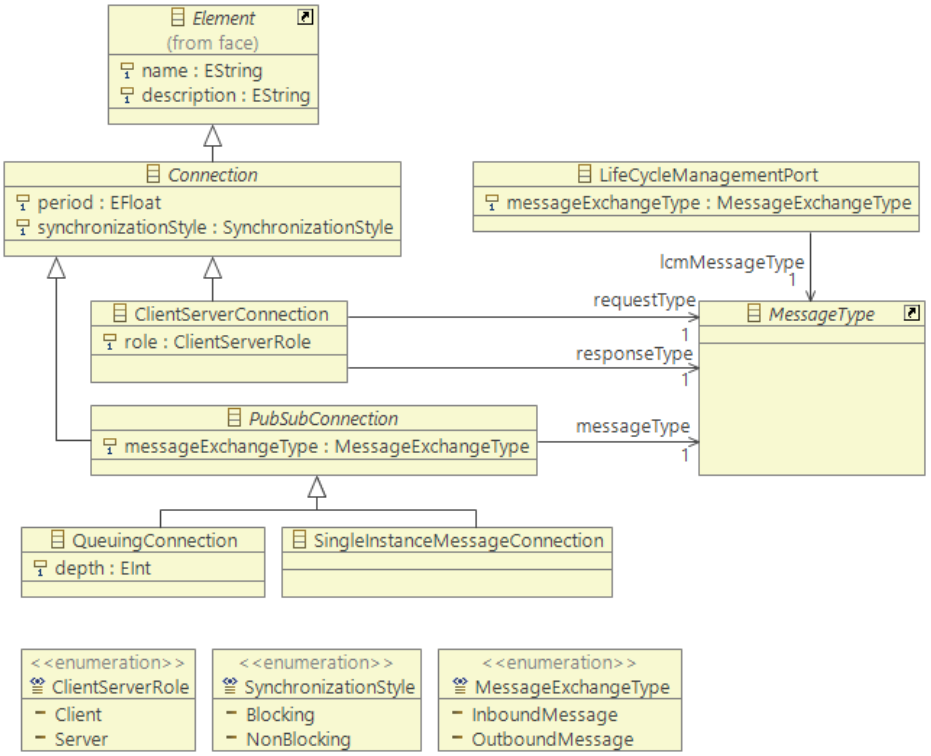

Figure 29: FACE Metamodel "face.uop" Package: UoP Connections

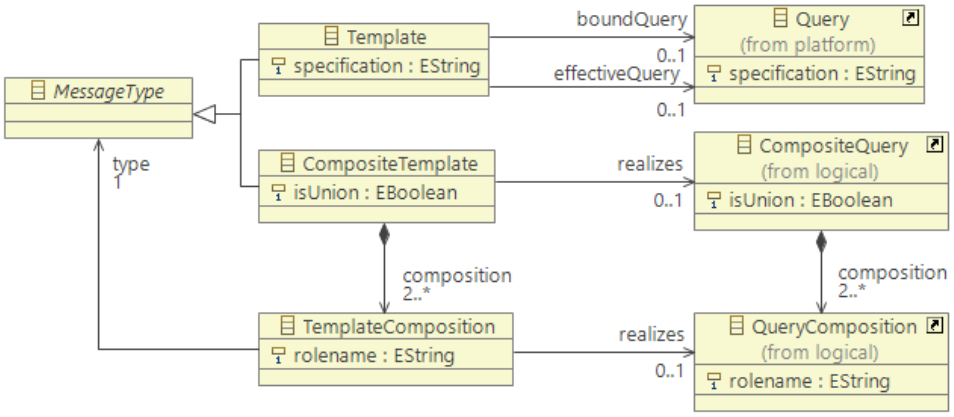

Figure 30: FACE Metamodel "face.uop" Package: Message Types

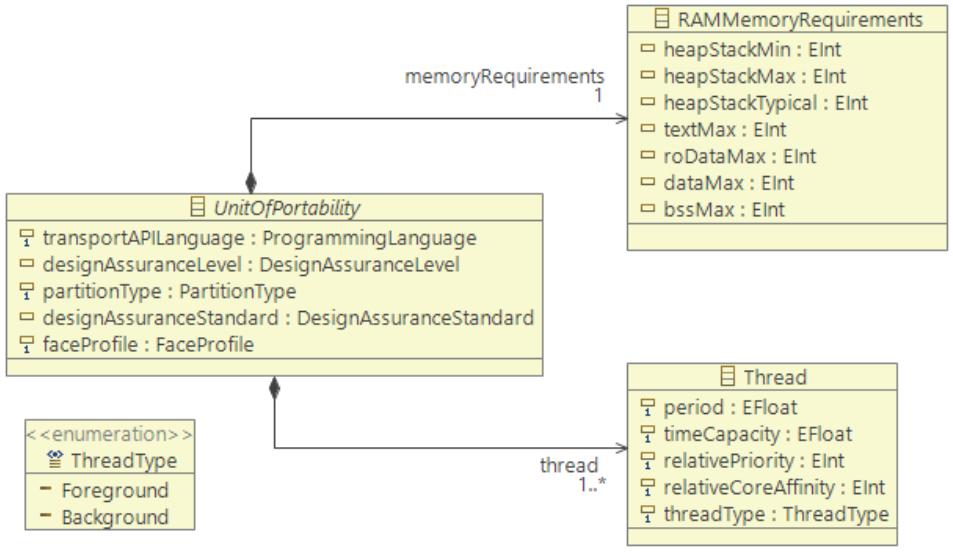

Figure 31: FACE Metamodel "face.uop" Package: UoP Characterization

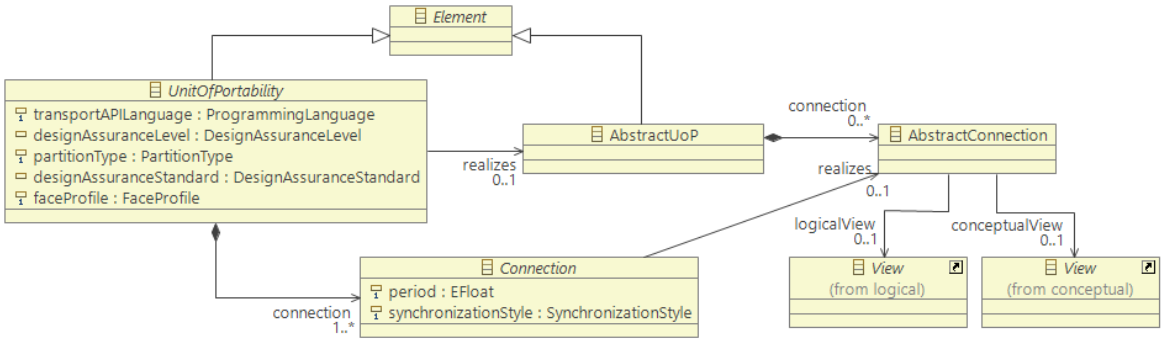

Figure 32: FACE Metamodel "face.uop" Package: Abstract UoP

J.2.2.1 Meta-Enumeration: face.uop.ClientServerRole

Description

The literals for the ClientServerRole enumeration are listed in Table 44.

Table 44: face.uop.ClientServerRole Literals

Name
Client
Server

J.2.2.2 Meta-Enumeration: face.uop.FaceProfile

Description

The FaceProfile enumeration captures the OS API subsets for a UoP as defined by the OSS. The literals for the FaceProfile enumeration are listed in Table 45.

Table 45: face.uop.FaceProfile Literals

Name
GeneralPurpose
Security
SafetyBase
SafetyExtended

J.2.2.3 Meta-Enumeration: face.uop.DesignAssuranceLevel

Description

The literals for the DesignAssuranceLevel enumeration are listed in Table 46.

Table 46: face.uop.DesignAssuranceLevel Literals

Name
A
B
C
D
E

J.2.2.4 Meta-Enumeration: face.uop.DesignAssuranceStandard

Description

The literals for the DesignAssuranceStandard enumeration are listed in Table 47.

Table 47: face.uop.DesignAssuranceStandard Literals

Name
DO_178B_ED_12B
DO_178C_ED_12C

J.2.2.5 Meta-Enumeration: face.uop.MessageExchangeType

Description

The MessageExchangeType enumeration captures the options for the message exchange type of a UoP port as defined by the TS Interface. The literals for the MessageExchangeType enumeration are listed in Table 48.

Table 48: face.uop.MessageExchangeType Literals

Name
InboundMessage
OutboundMessage

J.2.2.6 Meta-Enumeration: face.uop.PartitionType

Description

The PartitionType enumeration captures the OS API types for a UoP as defined by the OSS. The literals for the PartitionType enumeration are listed in Table 49.

Table 49: face.uop.PartitionType Literals

Name
POSIX
ARINC653

J.2.2.7 Meta-Enumeration: face.uop.ProgrammingLanguage

Description

The ProgrammingLanguage enumeration captures the options for programming language API bindings as defined by Section 4.14. The literals for the ProgrammingLanguage enumeration are listed in Table 50.

Table 50: face.uop.ProgrammingLanguage Literals

Name
C
CPP
Java
Ada

J.2.2.8 Meta-Enumeration: face.uop.SynchronizationStyle

Description

The SynchronizationStyle enumeration captures the options for the synchronization style of a UoP port as defined by the TS Interface. The literals for the SynchronizationStyle enumeration are listed in Table 51.

Table 51: face.uop.SynchronizationStyle Literals

Name
Blocking
NonBlocking

J.2.2.9 Meta-Enumeration: face.uop.ThreadType

Description

The literals for the ThreadType enumeration are listed in Table 52.

Table 52: face.uop.ThreadType Literals

Name
Foreground
Background

J.2.2.10 Meta-Class: face.uop.UoPModel

Description

A UoPModel is a container for UoC Elements. The relationships for the UoPModel meta-class are listed in Table 53.

Table 53: face.uop.UoPModel Relationships

Type	Name	Target	Multiplicity
Composition	element	Element	0..*

Type	Name	Target	Multiplicity
Composition	um	UoPModel	0..*
Generalization		face.Element	

J.2.2.11 Meta-Class: face.uop.Element

Description

A uop Element is the root type for defining the component elements of the ArchitectureModel. The relationships for the Element meta-class are listed in Table 54.

Table 54: face.uop.Element Relationships

Type	Name	Target	Multiplicity
Generalization		face.Element	

J.2.2.12 Meta-Class: face.uop.SupportingComponent

Description

A SupportingComponent is a LanguageRunTime or ApplicationFramework. The "version" attribute is the version of the SupportingComponent. The attributes for the SupportingComponent meta-class are listed in Table 55, and its relationships are shown in Table 56.

Table 55: face.uop.SupportingComponent Attributes

Name	Type	Multiplicity
version	string	1

Table 56: face.uop.SupportingComponent Relationships

Type	Name	Target	Multiplicity
Generalization		Element	

J.2.2.13 Meta-Class: face.uop.LanguageRunTime

Description

A LanguageRunTime is a language run-time as defined in Section 4.2.3. The relationships for the LanguageRunTime meta-class are listed in Table 57.

Table 57: face.uop.LanguageRunTime Relationships

Type	Name	Target	Multiplicity
Generalization		SupportingComponent	

J.2.2.14 Meta-Class: face.uop.ComponentFramework

Description

An ComponentFramework is an application framework as defined in Section 4.2.4. The relationships for the ComponentFramework meta-class are listed in Table 58.

Table 58: face.uop.ComponentFramework Relationships

Type	Name	Target	Multiplicity
Generalization		SupportingComponent	

J.2.2.15 Meta-Class: face.uop.AbstractUoP

Description

An AbstractUoP is used to capture the specification of a UoP. The relationships for the AbstractUoP meta-class are listed in Table 59.

Table 59: face.uop.AbstractUoP Relationships

Type	Name	Target	Multiplicity
Composition	connection	AbstractConnection	0..*
Generalization		Element	
Generalization		face.traceability.TraceableElement	

J.2.2.16 Meta-Class: face.uop.AbstractConnection

Description

An AbstractConnection captures the input and output characteristics of an AbstractUoP by specifying data at a Logical or Conceptual level. The relationships for the AbstractConnection meta-class are listed in Table 60.

Table 60: face.uop.AbstractConnection Relationships

Type	Name	Target	Multiplicity
Association	conceptualView	datamodel.conceptual.View	0..1
Association	logicalView	datamodel.logical.View	0..1
Generalization		face.Element	
Generalization		face.traceability.TraceableElement	

J.2.2.17 Meta-Class: face.uop.UnitOfPortability

Description

A UnitOfPortability is a FACE PlatformSpecificComponent or PortableComponent. The attributes for the UnitOfPortability meta-class are listed in Table 61, and its relationships are shown in Table 62.

Table 61: face.uop.UnitOfPortability Attributes

Name	Type	Multiplicity
transportAPILanguage	ProgrammingLanguage	1
designAssuranceLevel	DesignAssuranceLevel	0..1
partitionType	PartitionType	1
designAssuranceStandard	DesignAssuranceStandard	0..1
faceProfile	FaceProfile	1

Table 62: face.uop.UnitOfPortability Relationships

Type	Name	Target	Multiplicity
Association	supportingComponent	SupportingComponent	0..*
Composition	thread	Thread	1..*
Composition	memoryRequirements	RAMMemoryRequirements	1
Association	realizes	AbstractUoP	0..1
Composition	connection	Connection	1..*
Composition	lcmPort	LifeCycleManagementPort	0..2
Generalization		Element	
Generalization		face.traceability.TraceableElement	

J.2.2.18 Meta-Class: face.uop.PortableComponent

Description

A PortableComponent is a software component as defined by the PCS. The relationships for the PortableComponent meta-class are listed in Table 63.

Table 63: face.uop.PortableComponent Relationships

Type	Name	Target	Multiplicity
Generalization		UnitOfPortability	

J.2.2.19 Meta-Class: face.uop.PlatformSpecificComponent

Description

A PlatformSpecificComponent is a software component as defined by the PSSS. The relationships for the PlatformSpecificComponent meta-class are listed in Table 64.

Table 64: face.uop.PlatformSpecificComponent Relationships

Type	Name	Target	Multiplicity
Generalization		UnitOfPortability	

J.2.2.20 Meta-Class: face.uop.Thread

Description

A Thread defines the properties for the scheduling of a thread. The attributes for the Thread meta-class are listed in Table 65.

Table 65: face.uop.Thread Attributes

Name	Type	Multiplicity
period	float	1
timeCapacity	float	1
relativePriority	int	1
relativeCoreAffinity	int	1
threadType	ThreadType	1

J.2.2.21 Meta-Class: face.uop.RAMMemoryRequirements

Description

A RAMMemoryRequirements defines memory resources required by a UoP. The attributes for the RAMMemoryRequirements meta-class are listed in Table 66.

Table 66: face.uop.RAMMemoryRequirements Attributes

Name	Type	Multiplicity
heapStackMin	int	0..1
heapStackMax	int	0..1
heapStackTypical	int	0..1
textMax	int	0..1
roDataMax	int	0..1
dataMax	int	0..1

Name	Type	Multiplicity
bssMax	int	0..1

J.2.2.22 Meta-Class: face.uop.Connection

Description

A Connection is a communication endpoint on a FACE UoP. A Connection is either a Publisher, Subscriber, Client, or Server. The "messageType" specifies the platform View that is transmitted through the endpoint. If "period" is not specified, the endpoint is aperiodic. If "period" is specified, the value is the period of the endpoint in seconds. The attributes for the Connection meta-class are listed in Table 67, and its relationships are shown in Table 68.

Table 67: face.uop.Connection Attributes

Name	Type	Multiplicity
period	float	1
synchronizationStyle	SynchronizationStyle	1

Table 68: face.uop.Connection Relationships

Type	Name	Target	Multiplicity
Association	realizes	AbstractConnection	0..1
Generalization		face.traceability.TraceableElement	
Generalization		face.Element	

J.2.2.23 Meta-Class: face.uop.ClientServerConnection

Description

A ClientServerConnection is a Request/Reply Connection as defined in Section 4.7. The attributes for the ClientServerConnection meta-class are listed in Table 69, and its relationships are shown in Table 70.

Table 69: face.uop.ClientServerConnection Attributes

Name	Type	Multiplicity
role	ClientServerRole	1

Table 70: face.uop.ClientServerConnection Relationships

Type	Name	Target	Multiplicity
Association	requestType	MessageType	1
Association	responseType	MessageType	1

Type	Name	Target	Multiplicity
Generalization		Connection	

J.2.2.24 Meta-Class: face.uop.PubSubConnection

Description

A PubSubConnection is a QueuingConnection or a SingleInstanceMessageConnection. The "messageExchangeType" attribute defines the direction of the message relative to the UoP. The attributes for the PubSubConnection meta-class are listed in Table 71, and its relationships are shown in Table 72.

Table 71: face.uop.PubSubConnection Attributes

Name	Type	Multiplicity
messageExchangeType	MessageExchangeType	1

Table 72: face.uop.PubSubConnection Relationships

Type	Name	Target	Multiplicity
Association	messageType	MessageType	1
Generalization		Connection	

J.2.2.25 Meta-Class: face.uop.QueuingConnection

Description

A QueuingConnection is a PubSubConnection that supports buffering/queuing as defined in Section 4.8. The attributes for the QueuingConnection meta-class are listed in Table 73, and its relationships are shown in Table 74.

Table 73: face.uop.QueuingConnection Attributes

Name	Type	Multiplicity
depth	int	1

Table 74: face.uop.QueuingConnection Relationships

Type	Name	Target	Multiplicity
Generalization		PubSubConnection	

J.2.2.26 Meta-Class: face.uop.SingleInstanceMessageConnection

Description

A SingleInstanceMessageConnection is a PubSubConnection that supports single instance messaging as defined in Section 4.8. The relationships for the SingleInstanceMessageConnection meta-class are listed in Table 75.

Table 75: face.uop.SingleInstanceMessageConnection Relationships

Type	Name	Target	Multiplicity
Generalization		PubSubConnection	

J.2.2.27 Meta-Class: face.uop.LifeCycleManagementPort

Description

A LifeCycleManagementPort is used to define the life-cycle interface for the component. The "messageExchangeType" attribute defines the direction of the life-cycle message relative to the UoP. The attributes for the LifeCycleManagementPort meta-class are listed in Table 76, and its relationships are shown in Table 77.

Table 76: face.uop.LifeCycleManagementPort Attributes

Name	Type	Multiplicity
messageExchangeType	MessageExchangeType	1

Table 77: face.uop.LifeCycleManagementPort Relationships

Type	Name	Target	Multiplicity
Association	lcmMessageType	MessageType	1

J.2.2.28 Meta-Class: face.uop.MessageType

Description

A platform View is a Template or a CompositeTemplate. The relationships for the MessageType meta-class are listed in Table 78.

Table 78: face.uop.MessageType Relationships

Type	Name	Target	Multiplicity
Generalization		Element	
Generalization		face.traceability.TraceableElement	

J.2.2.29 Meta-Class: face.uop.CompositeTemplate

Description

A CompositeTemplate is a collection of two or more Templates. The "isUnion" attribute specifies whether the composed Templates are to be represented as cases in an IDL union or as members of an IDL struct. The attributes for the CompositeTemplate meta-class are listed in Table 79, and its relationships are shown in Table 80.

Table 79: face.uop.CompositeTemplate Attributes

Name	Type	Multiplicity
isUnion	boolean	1

Table 80: face.uop.CompositeTemplate Relationships

Type	Name	Target	Multiplicity
Composition	composition	TemplateComposition	2..* {Ordered}
Association	realizes	datamodel.logical.CompositeQuery	0..1
Generalization		Element	
Generalization		MessageType	

J.2.2.30 Meta-Class: face.uop.TemplateComposition

Description

A TemplateComposition is the mechanism that allows a CompositeTemplate to be constructed from Templates and other CompositeTemplates. The "rolename" attribute defines the name of the composed platform View within the scope of the composing CompositeTemplate. The "type" of a TemplateComposition is the platform View being used to construct the CompositeTemplate. The attributes for the TemplateComposition meta-class are listed in Table 81, and its relationships are shown in Table 82.

Table 81: face.uop.TemplateComposition Attributes

Name	Type	Multiplicity
rolename	string	1

Table 82: face.uop.TemplateComposition Relationships

Type	Name	Target	Multiplicity
Association	realizes	datamodel.logical.QueryComposition	0..1
Association	type	MessageType	1

J.2.2.31 Meta-Class: face.uop.Template

Description

A Template is a specification that defines a structure for Characteristics projected by its "boundQuery" or its "effectiveQuery". The "specification" attribute captures the specification of a Template as defined by the Template grammar in Section J.4. The attributes for the Template meta-class are listed in Table 83, and its relationships are shown in Table 84.

Table 83: face.uop.Template Attributes

Name	Type	Multiplicity
specification	string	1

Table 84: face.uop.Template Relationships

Type	Name	Target	Multiplicity
Association	boundQuery	datamodel.platform.Query	0..1
Association	effectiveQuery	datamodel.platform.Query	0..1
Generalization		MessageType	

J.2.3 Meta-Package: face.integration

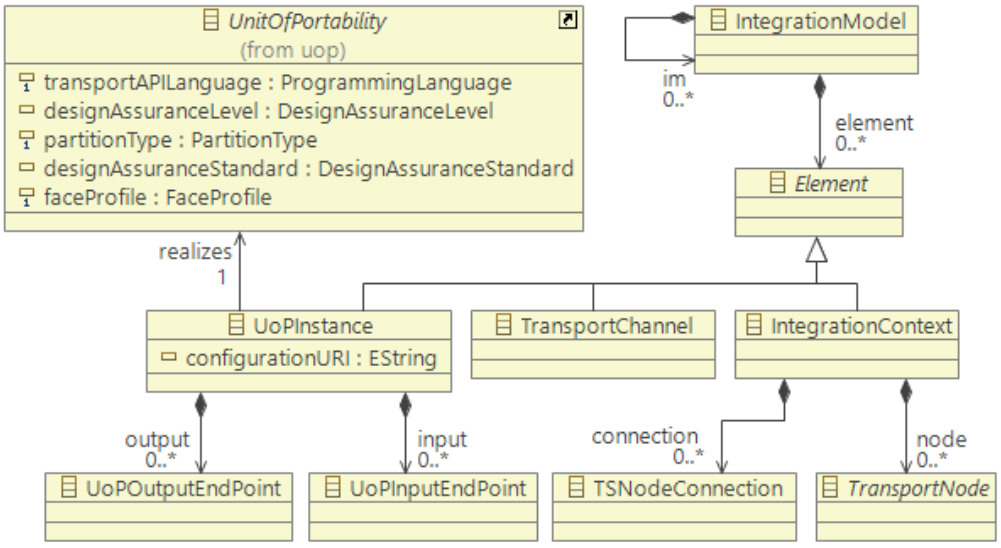

Figure 33: FACE Metamodel "face.integration" Package

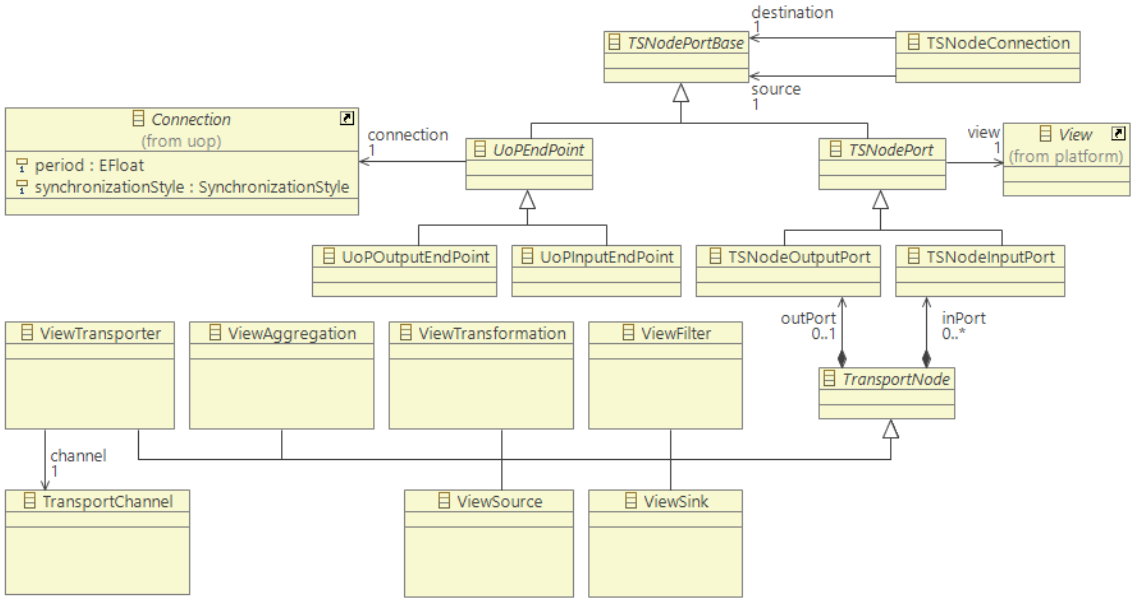

Figure 34: FACE Metamodel "face.integration" Package: Transport

J.2.3.1 Meta-Class: face.integration.IntegrationModel

Description

An IntegrationModel is a container for integration Elements. The relationships for the IntegrationModel meta-class are listed in Table 85.

Table 85: face.integration.IntegrationModel Relationships

Type	Name	Target	Multiplicity
Composition	im	IntegrationModel	0..*
Composition	element	Element	0..*
Generalization		face.Element	

J.2.3.2 Meta-Class: face.integration.Element

Description

An integration Element is the root type for defining the integration elements of the ArchitectureModel. The relationships for the Element meta-class are listed in Table 86.

Table 86: face.integration.Element Relationships

Type	Name	Target	Multiplicity
Generalization		face.Element	

J.2.3.3 Meta-Class: face.integration.IntegrationContext

Description

An IntegrationContext is a container used to group a set of TransportNodes and TSNodeConnections related to each other by a common, integrator defined context (e.g., collection and distribution of navigation data). The relationships for the IntegrationContext meta-class are listed in Table 87.

Table 87: face.integration.IntegrationContext Relationships

Type	Name	Target	Multiplicity
Composition	connection	TSNodeConnection	0..*
Composition	node	TransportNode	0..*
Generalization		Element	

J.2.3.4 Meta-Class: face.integration.TSNodeConnection

Description

A TSNodeConnection represents a connection between two TransportNodes. The relationships for the TSNodeConnection meta-class are listed in Table 88.

Table 88: face.integration.TSNodeConnection Relationships

Type	Name	Target	Multiplicity
Association	source	TSNodePortBase	1
Association	destination	TSNodePortBase	1

J.2.3.5 Meta-Class: face.integration.TSNodePortBase

Description

A TSNodePortBase is a port that can be used to connect a TransportNode and a UoPEndPoint together using a TSNodeConnection.

J.2.3.6 Meta-Class: face.integration.UoPInstance

Description

A UoPInstance represents an instance of a specific UoP within the system bounded by an integration model. An integration model can contain multiple instances of the same UoP. The attributes for the UoPInstance meta-class are listed in Table 89, and its relationships are shown in Table 90.

Table 89: face.integration.UoPInstance Attributes

Name	Type	Multiplicity
configurationURI	string	0..1

Table 90: face.integration.UoPInstance Relationships

Type	Name	Target	Multiplicity
Association	realizes	face.uop.UnitOfPortability	1
Composition	output	UoPOutputEndPoint	0..*
Composition	input	UoPInputEndPoint	0..*
Generalization		Element	

J.2.3.7 Meta-Class: face.integration.UoPEndPoint

Description

A UoPEndPoint is a specialization of a TSNodePortBase that allows connections in a UoPInstance to be part of a TSNodeConnection. This supports connecting UOP input and output endpoints to each other and to transport node input and output ports. The relationships for the UoPEndPoint meta-class are listed in Table 91.

Table 91: face.integration.UoPEndPoint Relationships

Type	Name	Target	Multiplicity
Association	connection	face.uop.Connection	1
Generalization		TSNodePortBase	

J.2.3.8 Meta-Class: face.integration.UoPInputEndPoint

Description

A UoPInputEndPoint is a specialization of a UoPEndPoint providing an endpoint which is used to input data to a UoP. The relationships for the UoPInputEndPoint meta-class are listed in Table 92.

Table 92: face.integration.UoPInputEndPoint Relationships

Type	Name	Target	Multiplicity
Generalization		UoPEndPoint	

J.2.3.9 Meta-Class: face.integration.UoPOutputEndPoint

Description

A UoPOutputEndPoint is a specialization of a UoPEndPoint providing an endpoint which is used to output data from a UoP. The relationships for the UoPOutputEndPoint meta-class are listed in Table 93.

Table 93: face.integration.UoPOutputEndPoint Relationships

Type	Name	Target	Multiplicity
Generalization		UoPEndPoint	

J.2.3.10 Meta-Class: face.integration.TransportNode

Description

A TransportNode is an abstraction of a node that performs a function along a path of communication from source UoPs to destination UoPs. The relationships for the TransportNode meta-class are listed in Table 94.

Table 94: face.integration.TransportNode Relationships

Type	Name	Target	Multiplicity
Composition	outPort	TSNodeOutputPort	0..1
Composition	inPort	TSNodeInputPort	0..*
Generalization		face.Element	

J.2.3.11 Meta-Class: face.integration.TSNodePort

Description

A TSNodePort is a port that provides a connection point to a TransportNode. A TSNodePort is typed by the "view" it references. The relationships for the TSNodePort meta-class are listed in Table 95.

Table 95: face.integration.TSNodePort Relationships

Type	Name	Target	Multiplicity
Association	view	face.datamodel.platform.View	1
Generalization		TSNodePortBase	

J.2.3.12 Meta-Class: face.integration.TSNodeInputPort

Description

A TSNodeInputPort is a specialization of a TSNodePort providing an endpoint which is used to input data to a TransportNode. The relationships for the TSNodeInputPort meta-class are listed in Table 96.

Table 96: face.integration.TSNodeInputPort Relationships

Type	Name	Target	Multiplicity
Generalization		TSNodePort	

J.2.3.13 Meta-Class: face.integration.TSNodeOutputPort

Description

A TSNodeOutputPort is a specialization of a TSNodePort providing an endpoint which is used to output data from a TransportNode. The relationships for the TSNodeOutputPort meta-class are listed in Table 97.

Table 97: face.integration.TSNodeOutputPort Relationships

Type	Name	Target	Multiplicity
Generalization		TSNodePort	

J.2.3.14 Meta-Class: face.integration.ViewAggregation

Description

A ViewAggregation represents of an instance of aggregation of data from multiple incoming views into a single outgoing view type, including transformation of input data to that required by the output view type. The relationships for the ViewAggregation meta-class are listed in Table 98.

Table 98: face.integration.ViewAggregation Relationships

Type	Name	Target	Multiplicity
Generalization		TransportNode	

J.2.3.15 Meta-Class: face.integration.ViewFilter

Description

A ViewFilter represents of an instance of a filter of data allowing a view to either pass through a filter, or to be filtered out (i.e., not passed through). A ViewFilter performs no transformation of data. The relationships for the ViewFilter meta-class are listed in Table 99.

Table 99: face.integration.ViewFilter Relationships

Type	Name	Target	Multiplicity
Generalization		TransportNode	

J.2.3.16 Meta-Class: face.integration.ViewSource

Description

A ViewSource is a TransportNode that only provides a View. The relationships for the ViewSource meta-class are listed in Table 100.

Table 100: face.integration.ViewSource Relationships

Type	Name	Target	Multiplicity
Generalization		TransportNode	

J.2.3.17 Meta-Class: face.integration.ViewSink

Description

A ViewSink is a TransportNode that only receives a View. The relationships for the ViewSink meta-class are listed in Table 101.

Table 101: face.integration.ViewSink Relationships

Type	Name	Target	Multiplicity
Generalization		TransportNode	

J.2.3.18 Meta-Class: face.integration.ViewTransformation

Description

A ViewTransformation represents an instance of transformation of data from one view type to another. The relationships for the ViewTransformation meta-class are listed in Table 102.

Table 102: face.integration.ViewTransformation Relationships

Type	Name	Target	Multiplicity
Generalization		TransportNode	

J.2.3.19 Meta-Class: face.integration.ViewTransporter

Description

A ViewTransporter represents the use of a TransportChannel with the intent of moving a view over it. The relationships for the ViewTransporter meta-class are listed in Table 103.

Table 103: face.integration.ViewTransporter Relationships

Type	Name	Target	Multiplicity
Association	channel	TransportChannel	1
Generalization		TransportNode	

J.2.3.20 Meta-Class: face.integration.TransportChannel

Description

A TransportChannel is a placeholder for an integrator supplied configuration between transport endpoints. The relationships for the TransportChannel meta-class are listed in Table 104.

Table 104: face.integration.TransportChannel Relationships

Type	Name	Target	Multiplicity
Generalization		Element	

J.2.4 Meta-Package: face.traceability

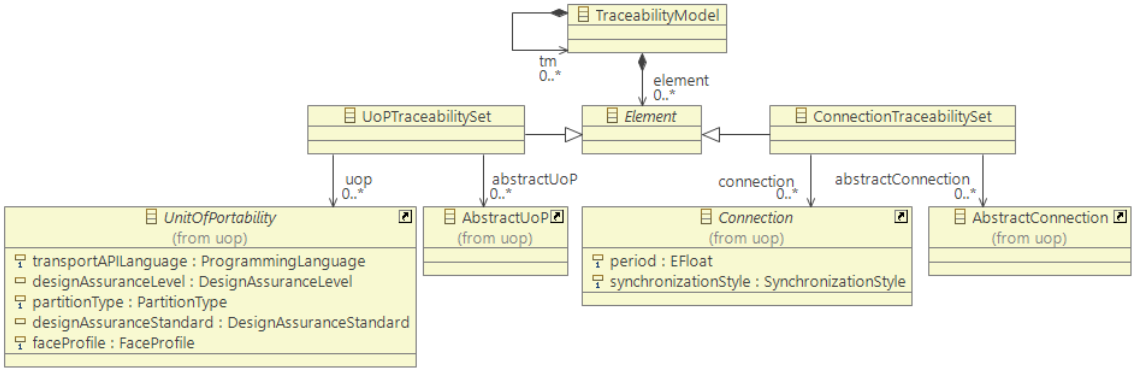

Figure 35: FACE Metamodel "face.traceability" Package

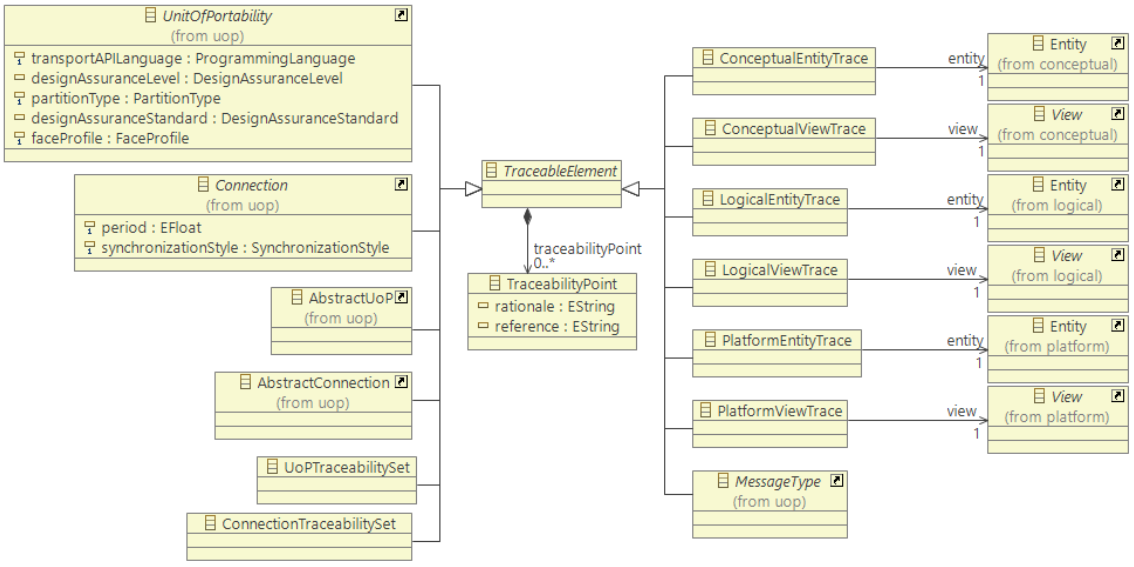

Figure 36: FACE Metamodel "face.traceability" Package: Traceable Elements

J.2.4.1 Meta-Class: face.traceability.TraceabilityModel

Description

A TraceabilityModel is a container for traceability Elements. The relationships for the TraceabilityModel meta-class are listed in Table 105.

Table 105: face.traceability.TraceabilityModel Relationships

Type	Name	Target	Multiplicity
Composition	element	Element	0..*
Composition	tm	TraceabilityModel	0..*
Generalization		face.Element	

J.2.4.2 Meta-Class: face.traceability.Element

Description

A traceability Element is the root type for defining the traceability elements of the FACE Architecture Model. The relationships for the Element meta-class are listed in Table 106.

Table 106: face.traceability.Element Relationships

Type	Name	Target	Multiplicity
Generalization		face.Element	

J.2.4.3 Meta-Class: face.traceability.TraceableElement

Description

A TraceableElement is used to capture traceability to other models. The relationships for the TraceableElement meta-class are listed in Table 107.

Table 107: face.traceability.TraceableElement Relationships

Type	Name	Target	Multiplicity
Composition	traceabilityPoint	TraceabilityPoint	0..*

J.2.4.4 Meta-Class: face.traceability.TraceabilityPoint

Description

A TraceabilityPoint is used to document the relationship between a TraceableElement and an external model. The "reference" attribute is a reference to the external model. The "rationale" attribute is used to document the reasoning behind the Trace. The attributes for the TraceabilityPoint meta-class are listed in Table 108.

Table 108: face.traceability.TraceabilityPoint Attributes

Name	Type	Multiplicity
rationale	string	0..1
reference	string	0..1

J.2.4.5 Meta-Class: face.traceability.UoPTraceabilitySet

Description

A UoPTraceabilitySet is used to relate a set of UoPs and/or AbstractUoPs to a set of TraceabilityPoints. The relationships for the UoPTraceabilitySet meta-class are listed in Table 109.

Table 109: face.traceability.UoPTraceabilitySet Relationships

Type	Name	Target	Multiplicity
Association	uop	face.uop.UnitOfPortability	0..*
Association	abstractUoP	face.uop.AbstractUoP	0..*
Generalization		Element	
Generalization		TraceableElement	

J.2.4.6 Meta-Class: face.traceability.ConnectionTraceabilitySet

Description

A ConnectionTraceabilitySet is used to relate a set of Connections and/or AbstractConnections to a set of TraceabilityPoints. The relationships for the ConnectionTraceabilitySet meta-class are listed in Table 110.

Table 110: face.traceability.ConnectionTraceabilitySet Relationships

Type	Name	Target	Multiplicity
Association	connection	face.uop.Connection	0..*
Association	abstractConnection	face.uop.AbstractConnection	0..*
Generalization		Element	
Generalization		TraceableElement	

J.2.4.7 Meta-Class: face.traceability.ConceptualEntityTrace

Description

The relationships for the ConceptualEntityTrace meta-class are listed in Table 111.

Table 111: face.traceability.ConceptualEntityTrace Relationships

Type	Name	Target	Multiplicity
Association	entity	datamodel.conceptual.Entity	1
Generalization		Element	
Generalization		TraceableElement	

J.2.4.8 Meta-Class: face.traceability.ConceptualViewTrace

Description

The relationships for the ConceptualViewTrace meta-class are listed in Table 112.

Table 112: face.traceability.ConceptualViewTrace Relationships

Type	Name	Target	Multiplicity
Association	view	datamodel.conceptual.View	1
Generalization		Element	
Generalization		TraceableElement	

J.2.4.9 Meta-Class: face.traceability.LogicalEntityTrace

Description

The relationships for the LogicalEntityTrace meta-class are listed in Table 113.

Table 113: face.traceability.LogicalEntityTrace Relationships

Type	Name	Target	Multiplicity
Association	entity	datamodel.logical.Entity	1
Generalization		Element	
Generalization		TraceableElement	

J.2.4.10 Meta-Class: face.traceability.LogicalViewTrace

Description

The relationships for the LogicalViewTrace meta-class are listed in Table 114.

Table 114: face.traceability.LogicalViewTrace Relationships

Type	Name	Target	Multiplicity
Association	view	datamodel.logical.View	1
Generalization		Element	
Generalization		TraceableElement	

J.2.4.11 Meta-Class: face.traceability.PlatformEntityTrace

Description

The relationships for the PlatformEntityTrace meta-class are listed in Table 115.

Table 115: face.traceability.PlatformEntityTrace Relationships

Type	Name	Target	Multiplicity
Association	entity	datamodel.platform.Entity	1
Generalization		Element	
Generalization		TraceableElement	

J.2.4.12 Meta-Class: face.traceability.PlatformViewTrace

Description

The relationships for the PlatformViewTrace meta-class are listed in Table 116.

Table 116: face.traceability.PlatformViewTrace Relationships

Type	Name	Target	Multiplicity
Association	view	datamodel.platform.View	1
Generalization		Element	
Generalization		TraceableElement	

J.2.5 Meta-Package: face.uop

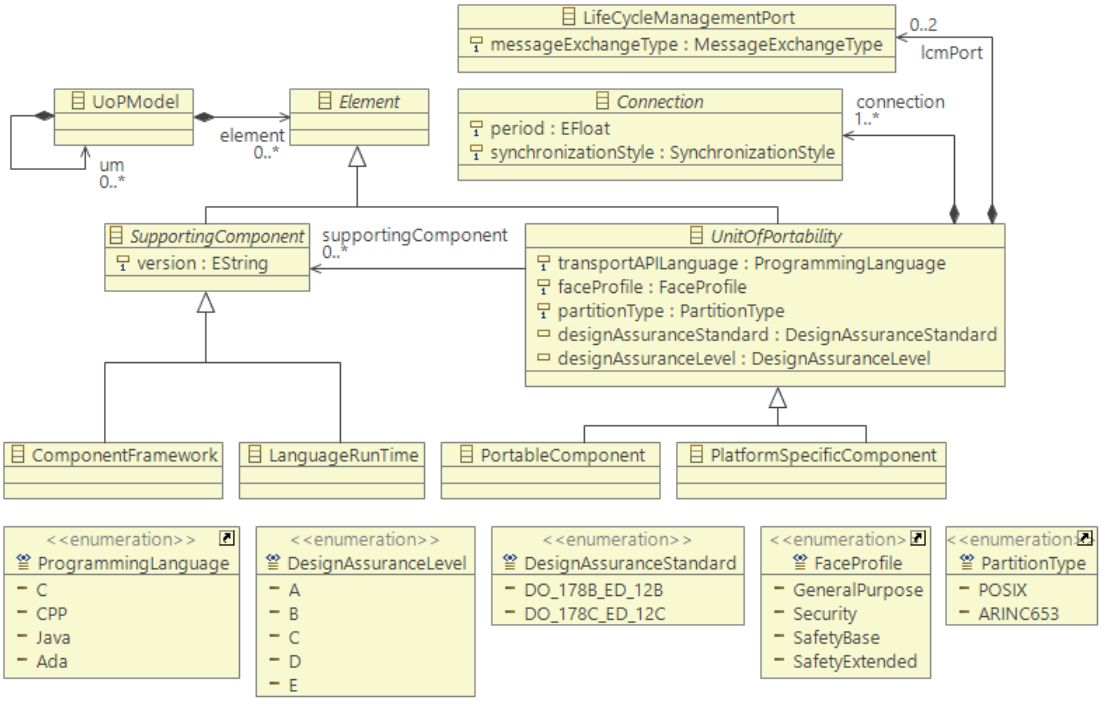

Figure 37: FACE Metamodel "face.uop" Package

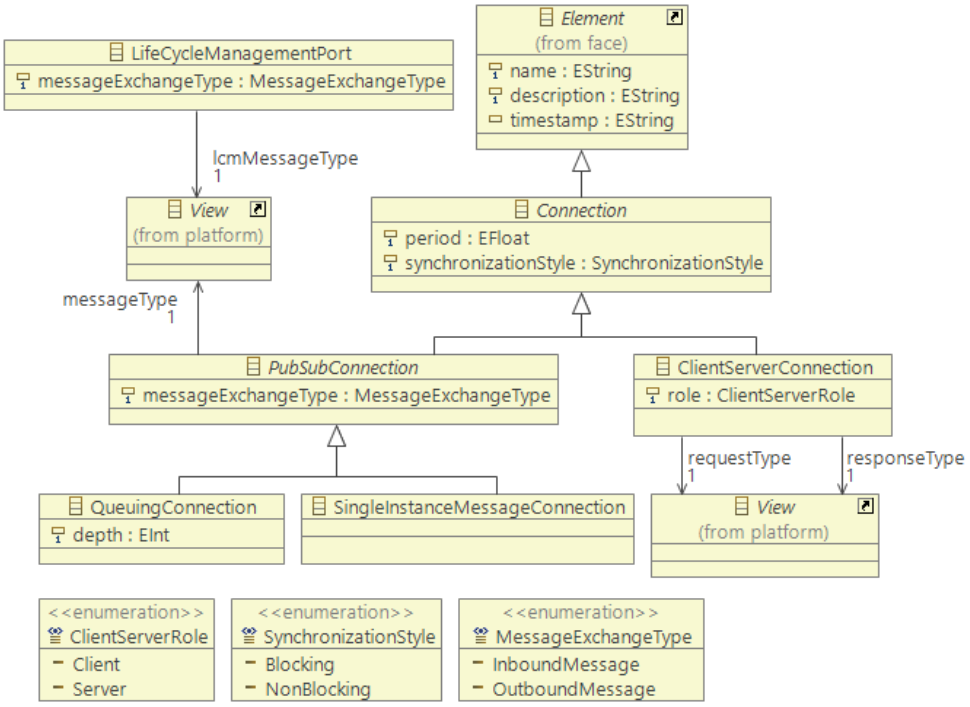

Figure 38: FACE Metamodel "face.uop" Package: UoP Connections

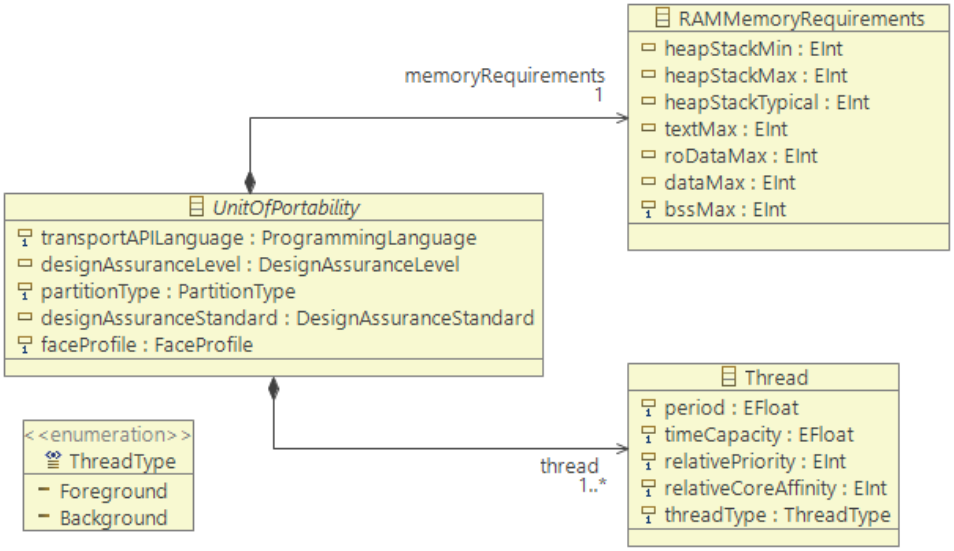

Figure 39: FACE Metamodel "face.uop" Package: UoP Characterization

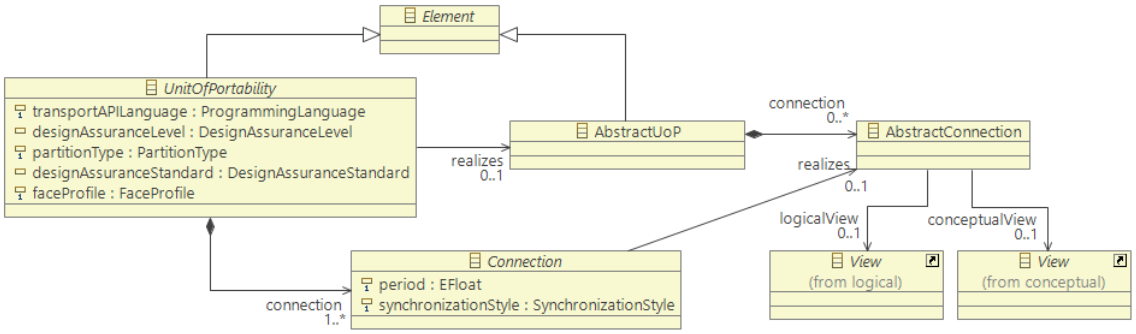

Figure 40: FACE Metamodel "face.uop" Package: Abstract UoP

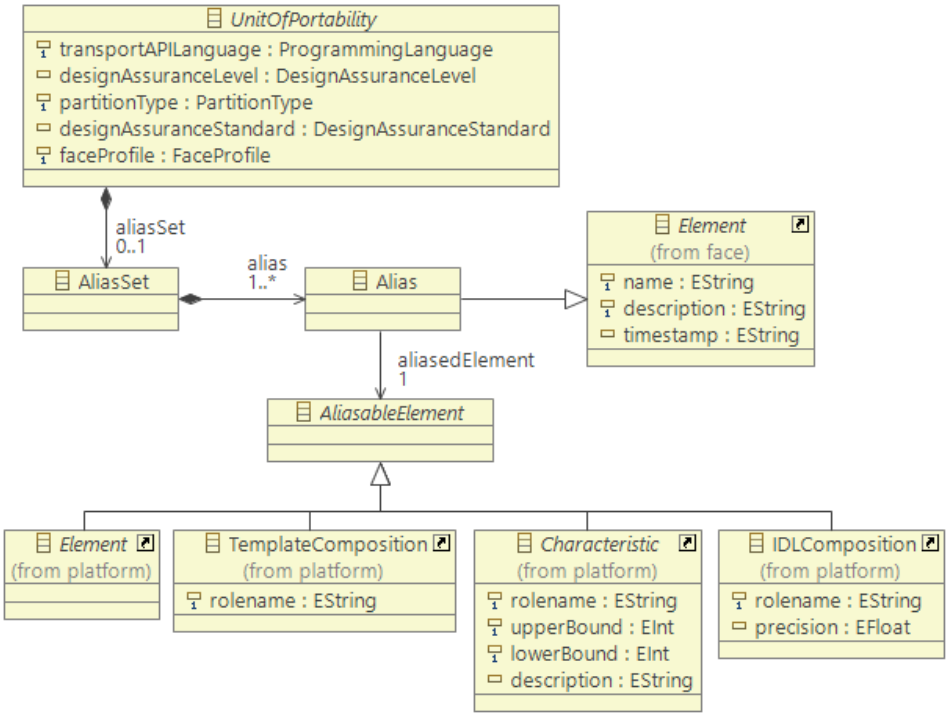

Figure 41: FACE Metamodel "face.uop" Package: Aliases

J.2.5.1 Meta-Enumeration: face.uop.ClientServerRole

Description

The literals for the ClientServerRole enumeration are listed in Table 117.

Table 117: face.uop.ClientServerRole Literals

Name
Client
Server

J.2.5.2 Meta-Enumeration: face.uop.FaceProfile

Description

The FaceProfile enumeration captures the OS API subsets for a UoP as defined by the OSS. The literals for the FaceProfile enumeration are listed in Table 118.

Table 118: face.uop.FaceProfile Literals

Name
GeneralPurpose
Security
SafetyBase
SafetyExtended

J.2.5.3 Meta-Enumeration: face.uop.DesignAssuranceLevel

Description

The literals for the DesignAssuranceLevel enumeration are listed in Table 119.

Table 119: face.uop.DesignAssuranceLevel Literals

Name
A
B
C
D
E

J.2.5.4 Meta-Enumeration: face.uop.DesignAssuranceStandard

Description

The literals for the DesignAssuranceStandard enumeration are listed in Table 120.

Table 120: face.uop.DesignAssuranceStandard Literals

Name
DO_178B_ED_12B
DO_178C_ED_12C

J.2.5.5 Meta-Enumeration: face.uop.MessageExchangeType

Description

The MessageExchangeType enumeration captures the options for the message exchange type of a UoP port as defined by the TS Interface. The literals for the MessageExchangeType enumeration are listed in Table 121.

Table 121: face.uop.MessageExchangeType Literals

Name
InboundMessage
OutboundMessage

J.2.5.6 Meta-Enumeration: face.uop.PartitionType

Description

The PartitionType enumeration captures the OS API types for a UoP as defined by the OSS. The literals for the PartitionType enumeration are listed in Table 122.

Table 122: face.uop.PartitionType Literals

Name
POSIX
ARINC653

J.2.5.7 Meta-Enumeration: face.uop.ProgrammingLanguage

Description

The ProgrammingLanguage enumeration captures the options for programming language API bindings as defined by Section 4.14. The literals for the ProgrammingLanguage enumeration are listed in Table 123.

Table 123: face.uop.ProgrammingLanguage Literals

Name
C
CPP
Java
Ada

J.2.5.8 Meta-Enumeration: face.uop.SynchronizationStyle

Description

The SynchronizationStyle enumeration captures the options for the synchronization style of a UoP port as defined by the TS Interface. The literals for the SynchronizationStyle enumeration are listed in Table 124.

Table 124: face.uop.SynchronizationStyle Literals

Name
Blocking
NonBlocking

J.2.5.9 Meta-Enumeration: face.uop.ThreadType

Description

The literals for the ThreadType enumeration are listed in Table 125.

Table 125: face.uop.ThreadType Literals

Name
Foreground
Background

J.2.5.10 Meta-Class: face.uop.UoPModel

Description

A UoPModel is a container for UoC Elements. The relationships for the UoPModel meta-class are listed in Table 126.

Table 126: face.uop.UoPModel Relationships

Type	Name	Target	Multiplicity
Composition	element	Element	0..*
Composition	um	UoPModel	0..*
Generalization		face.Element	

J.2.5.11 Meta-Class: face.uop.Element

Description

A uop Element is the root type for defining the component elements of the ArchitectureModel. The relationships for the Element meta-class are listed in Table 127.

Table 127: face.uop.Element Relationships

Type	Name	Target	Multiplicity
Generalization		face.Element	

J.2.5.12 Meta-Class: face.uop.SupportingComponent

Description

A SupportingComponent is a LanguageRunTime or ApplicationFramework. The "version" attribute is the version of the SupportingComponent. The attributes for the SupportingComponent meta-class are listed in Table 128, and its relationships are shown in Table 129.

Table 128: face.uop.SupportingComponent Attributes

Name	Type	Multiplicity
version	string	1

Table 129: face.uop.SupportingComponent Relationships

Type	Name	Target	Multiplicity
Generalization		Element	

J.2.5.13 Meta-Class: face.uop.LanguageRunTime

Description

A LanguageRunTime is a language run-time as defined in Section 4.2.3. The relationships for the LanguageRunTime meta-class are listed in Table 130.

Table 130: face.uop.LanguageRunTime Relationships

Type	Name	Target	Multiplicity
Generalization		SupportingComponent	

J.2.5.14 Meta-Class: face.uop.ComponentFramework

Description

An ComponentFramework is an application framework as defined in Section 4.2.4. The relationships for the ComponentFramework meta-class are listed in Table 131.

Table 131: face.uop.ComponentFramework Relationships

Type	Name	Target	Multiplicity
Generalization		SupportingComponent	

J.2.5.15 Meta-Class: face.uop.AbstractUoP

Description

An AbstractUoP is used to capture the specification of a UoP. The relationships for the AbstractUoP meta-class are listed in Table 132.

Table 132: face.uop.AbstractUoP Relationships

Type	Name	Target	Multiplicity
Composition	connection	AbstractConnection	0..*
Generalization		Element	
Generalization		face.traceability.TraceableElement	

J.2.5.16 Meta-Class: face.uop.AbstractConnection

Description

An AbstractConnection captures the input and output characteristics of an AbstractUoP by specifying data at a Logical or Conceptual level. The relationships for the AbstractConnection meta-class are listed in Table 133.

Table 133: face.uop.AbstractConnection Relationships

Type	Name	Target	Multiplicity
Association	conceptualView	face.datamodel.conceptual.View	0..1
Association	logicalView	face.datamodel.logical.View	0..1
Generalization		face.Element	
Generalization		face.traceability.TraceableElement	

J.2.5.17 Meta-Class: face.uop.UnitOfPortability

Description

A UnitOfPortability is a FACE PlatformSpecificComponent or PortableComponent. The attributes for the UnitOfPortability meta-class are listed in Table 134, and its relationships are shown in Table 135.

Table 134: face.uop.UnitOfPortability Attributes

Name	Type	Multiplicity
transportAPILanguage	ProgrammingLanguage	1
designAssuranceLevel	DesignAssuranceLevel	0..1
partitionType	PartitionType	1

Name	Type	Multiplicity
designAssuranceStandard	DesignAssuranceStandard	0..1
faceProfile	FaceProfile	1

Table 135: face.uop.UnitOfPortability Relationships

Type	Name	Target	Multiplicity
Association	supportingComponent	SupportingComponent	0..*
Composition	thread	Thread	1..*
Composition	memoryRequirements	RAMMemoryRequirements	1
Association	realizes	AbstractUoP	0..1
Composition	connection	Connection	1..*
Composition	lcmPort	LifeCycleManagementPort	0..2
Generalization		Element	
Generalization		face.traceability.TraceableElement	

J.2.5.18 Meta-Class: face.uop.PortableComponent

Description

A PortableComponent is a software component as defined by the PCS. The relationships for the PortableComponent meta-class are listed in Table 136.

Table 136: face.uop.PortableComponent Relationships

Type	Name	Target	Multiplicity
Generalization		UnitOfPortability	

J.2.5.19 Meta-Class: face.uop.PlatformSpecificComponent

Description

A PlatformSpecificComponent is a software component as defined by the PSSS. The relationships for the PlatformSpecificComponent meta-class are listed in Table 137.

Table 137: face.uop.PlatformSpecificComponent Relationships

Type	Name	Target	Multiplicity
Generalization		UnitOfPortability	

J.2.5.20 Meta-Class: face.uop.Thread

Description

A Thread defines the properties for the scheduling of a thread. The attributes for the Thread meta-class are listed in Table 138.

Table 138: face.uop.Thread Attributes

Name	Type	Multiplicity
period	float	1
timeCapacity	float	1
relativePriority	int	1
relativeCoreAffinity	int	1
threadType	ThreadType	1

J.2.5.21 Meta-Class: face.uop.RAMMemoryRequirements

Description

A RAMMemoryRequirements defines memory resources required by a UoP. The attributes for the RAMMemoryRequirements meta-class are listed in Table 139.

Table 139: face.uop.RAMMemoryRequirements Attributes

Name	Type	Multiplicity
heapStackMin	int	0..1
heapStackMax	int	0..1
heapStackTypical	int	0..1
textMax	int	0..1
roDataMax	int	0..1
dataMax	int	0..1
bssMax	int	0..1

J.2.5.22 Meta-Class: face.uop.Connection

Description

A Connection is a communication endpoint on a FACE UoP. A Connection is either a Publisher, Subscriber, Client, or Server. The “messageType” specifies the platform View that is transmitted through the endpoint. If “period” is not specified, the endpoint is aperiodic. If “period” is specified, the value is the period of the endpoint in seconds. The attributes for the Connection meta-class are listed in Table 140, and its relationships are shown in Table 141.

Table 140: face.uop.Connection Attributes

Name	Type	Multiplicity
period	float	1
synchronizationStyle	SynchronizationStyle	1

Table 141: face.uop.Connection Relationships

Type	Name	Target	Multiplicity
Association	realizes	AbstractConnection	0..1
Generalization		face.traceability.TraceableElement	
Generalization		face.Element	

J.2.5.23 Meta-Class: face.uop.ClientServerConnection

Description

A ClientServerConnection is a Request/Reply Connection as defined in Section 4.7. The attributes for the ClientServerConnection meta-class are listed in Table 142, and its relationships are shown in Table 143.

Table 142: face.uop.ClientServerConnection Attributes

Name	Type	Multiplicity
role	ClientServerRole	1

Table 143: face.uop.ClientServerConnection Relationships

Type	Name	Target	Multiplicity
Association	requestType	face.datamodel.platform.View	1
Association	responseType	face.datamodel.platform.View	1
Generalization		Connection	

J.2.5.24 Meta-Class: face.uop.PubSubConnection

Description

A PubSubConnection is a QueuingConnection or a SingleInstanceMessageConnection. The “messageExchangeType” attribute defines the direction of the message relative to the UoP. The attributes for the PubSubConnection meta-class are listed in Table 144, and its relationships are shown in Table 145.

Table 144: face.uop.PubSubConnection Attributes

Name	Type	Multiplicity
messageExchangeType	MessageExchangeType	1

Table 145: face.uop.PubSubConnection Relationships

Type	Name	Target	Multiplicity
Association	messageType	face.datamodel.platform.View	1
Generalization		Connection	

J.2.5.25 Meta-Class: face.uop.QueuingConnection

Description

A QueuingConnection is a PubSubConnection that supports buffering/queuing as defined in Section 4.8. The attributes for the QueuingConnection meta-class are listed in Table 146, and its relationships are shown in Table 147.

Table 146: face.uop.QueuingConnection Attributes

Name	Type	Multiplicity
depth	int	1

Table 147: face.uop.QueuingConnection Relationships

Type	Name	Target	Multiplicity
Generalization		PubSubConnection	

J.2.5.26 Meta-Class: face.uop.SingleInstanceMessageConnection

Description

A SingleInstanceMessageConnection is a PubSubConnection that supports single instance messaging as defined in Section 4.8. The relationships for the SingleInstanceMessageConnection meta-class are listed in Table 148.

Table 148: face.uop.SingleInstanceMessageConnection Relationships

Type	Name	Target	Multiplicity
Generalization		PubSubConnection	

J.2.5.27 Meta-Class: face.uop.LifeCycleManagementPort

Description

A LifeCycleManagementPort is used to define the life-cycle interface for the component. The "messageExchangeType" attribute defines the direction of the life-cycle message relative to the UoP. The attributes for the LifeCycleManagementPort meta-class are listed in Table 149, and its relationships are shown in Table 150.

Table 149: face.uop.LifeCycleManagementPort Attributes

Name	Type	Multiplicity
messageExchangeType	MessageExchangeType	1

Table 150: face.uop.LifeCycleManagementPort Relationships

Type	Name	Target	Multiplicity
Association	lcmMessageType	face.datamodel.platform.View	1

J.2.6 Meta-Package: face.integration

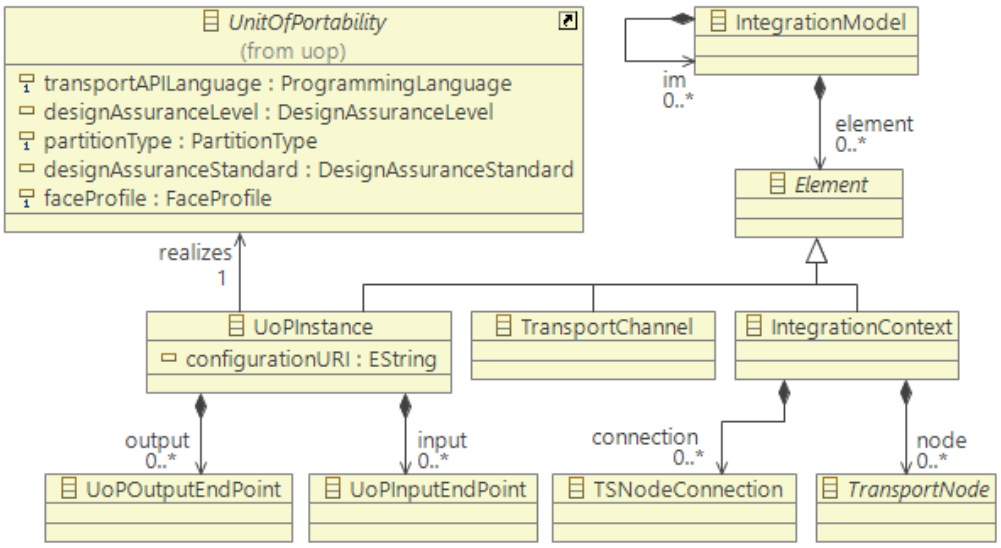

Figure 42: FACE Metamodel "face.integration" Package

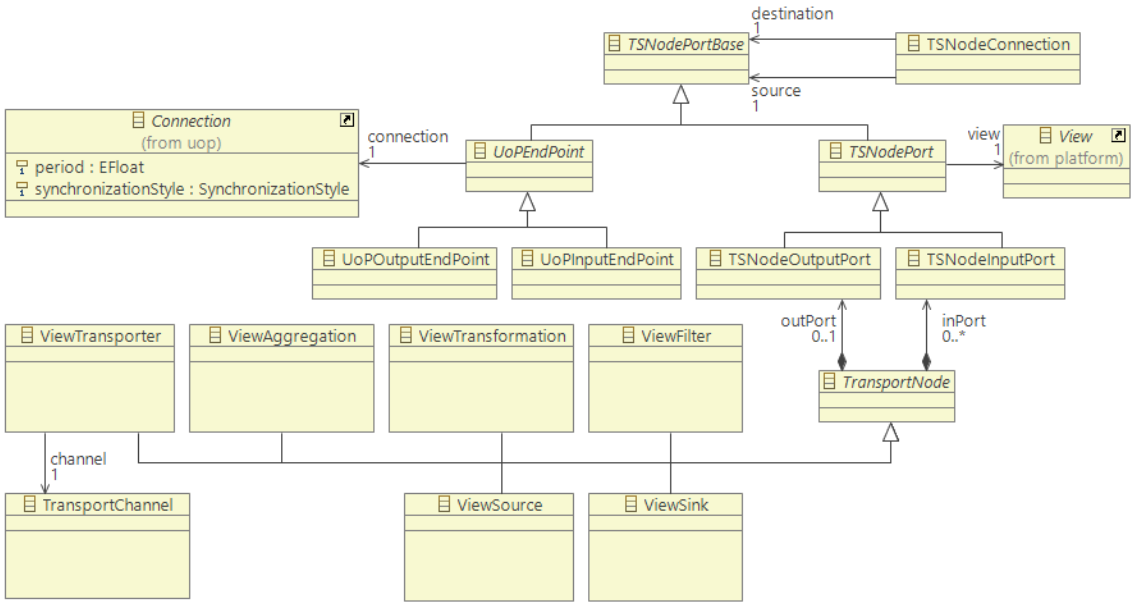

Figure 43: FACE Metamodel "face.integration" Package: Transport

J.2.6.1 Meta-Class: face.integration.IntegrationModel

Description

An IntegrationModel is a container for integration Elements. The relationships for the IntegrationModel meta-class are listed in Table 151.

Table 151: face.integration.IntegrationModel Relationships

Type	Name	Target	Multiplicity
Composition	im	IntegrationModel	0..*
Composition	element	Element	0..*
Generalization		face.Element	

J.2.6.2 Meta-Class: face.integration.Element

Description

An integration Element is the root type for defining the integration elements of the ArchitectureModel. The relationships for the Element meta-class are listed in Table 152.

Table 152: face.integration.Element Relationships

Type	Name	Target	Multiplicity
Generalization		face.Element	

J.2.6.3 Meta-Class: face.integration.IntegrationContext

Description

An IntegrationContext is a container used to group a set of TransportNodes and TSNodeConnections related to each other by a common, integrator defined context (e.g., collection and distribution of navigation data). The relationships for the IntegrationContext meta-class are listed in Table 153.

Table 153: face.integration.IntegrationContext Relationships

Type	Name	Target	Multiplicity
Composition	connection	TSNodeConnection	0..*
Composition	node	TransportNode	0..*
Generalization		Element	

J.2.6.4 Meta-Class: face.integration.TSNodeConnection

Description

A TSNodeConnection represents a connection between two TransportNodes. The relationships for the TSNodeConnection meta-class are listed in Table 154.

Table 154: face.integration.TSNodeConnection Relationships

Type	Name	Target	Multiplicity
Association	source	TSNodePortBase	1
Association	destination	TSNodePortBase	1

J.2.6.5 Meta-Class: face.integration.TSNodePortBase

Description

A TSNodePortBase is a port that can be used to connect a TransportNode and a UoPEndPoint together using a TSNodeConnection.

J.2.6.6 Meta-Class: face.integration.UoPInstance

Description

A UoPInstance represents an instance of a specific UoP within the system bounded by an integration model. An integration model can contain multiple instances of the same UoP. The attributes for the UoPInstance meta-class are listed in Table 155, and its relationships are shown in Table 156.

Table 155: face.integration.UoPInstance Attributes

Name	Type	Multiplicity
configurationURI	string	0..1

Table 156: face.integration.UoPInstance Relationships

Type	Name	Target	Multiplicity
Association	realizes	face.uop.UnitOfPortability	1
Composition	output	UoPOutputEndPoint	0..*
Composition	input	UoPInputEndPoint	0..*
Generalization		Element	

J.2.6.7 Meta-Class: face.integration.UoPEndPoint

Description

A UoPEndPoint is a specialization of a TSNodePortBase that allows connections in a UoPInstance to be part of a TSNodeConnection. This supports connecting UOP input and output endpoints to each other and to transport node input and output ports. The relationships for the UoPEndPoint meta-class are listed in Table 157.

Table 157: face.integration.UoPEndPoint Relationships

Type	Name	Target	Multiplicity
Association	connection	face.uop.Connection	1
Generalization		TSNodePortBase	

J.2.6.8 Meta-Class: face.integration.UoPInputEndPoint

Description

A UoPInputEndPoint is a specialization of a UoPEndPoint providing an endpoint which is used to input data to a UoP. The relationships for the UoPInputEndPoint meta-class are listed in Table 158.

Table 158: face.integration.UoPInputEndPoint Relationships

Type	Name	Target	Multiplicity
Generalization		UoPEndPoint	

J.2.6.9 Meta-Class: face.integration.UoPOutputEndPoint

Description

A UoPOutputEndPoint is a specialization of a UoPEndPoint providing an endpoint which is used to output data from a UoP. The relationships for the UoPOutputEndPoint meta-class are listed in Table 159.

Table 159: face.integration.UoPOutputEndPoint Relationships

Type	Name	Target	Multiplicity
Generalization		UoPEndPoint	

J.2.6.10 Meta-Class: face.integration.TransportNode

Description

A TransportNode is an abstraction of a node that performs a function along a path of communication from source UoPs to destination UoPs. The relationships for the TransportNode meta-class are listed in Table 160.

Table 160: face.integration.TransportNode Relationships

Type	Name	Target	Multiplicity
Composition	outPort	TSNodeOutputPort	0..1
Composition	inPort	TSNodeInputPort	0..*
Generalization		face.Element	

J.2.6.11 Meta-Class: face.integration.TSNodePort

Description

A TSNodePort is a port that provides a connection point to a TransportNode. A TSNodePort is typed by the "view" it references. The relationships for the TSNodePort meta-class are listed in Table 161.

Table 161: face.integration.TSNodePort Relationships

Type	Name	Target	Multiplicity
Association	view	face.uop.MessageType	1
Generalization		TSNodePortBase	

J.2.6.12 Meta-Class: face.integration.TSNodeInputPort

Description

A TSNodeInputPort is a specialization of a TSNodePort providing an endpoint which is used to input data to a TransportNode. The relationships for the TSNodeInputPort meta-class are listed in Table 162.

Table 162: face.integration.TSNodeInputPort Relationships

Type	Name	Target	Multiplicity
Generalization		TSNodePort	

J.2.6.13 Meta-Class: face.integration.TSNodeOutputPort

Description

A TSNodeOutputPort is a specialization of a TSNodePort providing an endpoint which is used to output data from a TransportNode. The relationships for the TSNodeOutputPort meta-class are listed in Table 163.

Table 163: face.integration.TSNodeOutputPort Relationships

Type	Name	Target	Multiplicity
Generalization		TSNodePort	

J.2.6.14 Meta-Class: face.integration.ViewAggregation

Description

A ViewAggregation represents of an instance of aggregation of data from multiple incoming views into a single outgoing view type, including transformation of input data to that required by the output view type. The relationships for the ViewAggregation meta-class are listed in Table 164.

Table 164: face.integration.ViewAggregation Relationships

Type	Name	Target	Multiplicity
Generalization		TransportNode	

J.2.6.15 Meta-Class: face.integration.ViewFilter

Description

A ViewFilter represents of an instance of a filter of data allowing a view to either pass through a filter, or to be filtered out (i.e., not passed through). A ViewFilter performs no transformation of data. The relationships for the ViewFilter meta-class are listed in Table 165.

Table 165: face.integration.ViewFilter Relationships

Type	Name	Target	Multiplicity
Generalization		TransportNode	

J.2.6.16 Meta-Class: face.integration.ViewSource

Description

A ViewSource is a TransportNode that only provides a View. The relationships for the ViewSource meta-class are listed in Table 166.

Table 166: face.integration.ViewSource Relationships

Type	Name	Target	Multiplicity
Generalization		TransportNode	

J.2.6.17 Meta-Class: face.integration.ViewSink

Description

A ViewSink is a TransportNode that only receives a View. The relationships for the ViewSink meta-class are listed in Table 167.

Table 167: face.integration.ViewSink Relationships

Type	Name	Target	Multiplicity
Generalization		TransportNode	

J.2.6.18 Meta-Class: face.integration.ViewTransformation

Description

A ViewTransformation represents an instance of transformation of data from one view type to another. The relationships for the ViewTransformation meta-class are listed in Table 168.

Table 168: face.integration.ViewTransformation Relationships

Type	Name	Target	Multiplicity
Generalization		TransportNode	

J.2.6.19 Meta-Class: face.integration.ViewTransporter

Description

A ViewTransporter represents the use of a TransportChannel with the intent of moving a view over it. The relationships for the ViewTransporter meta-class are listed in Table 169.

Table 169: face.integration.ViewTransporter Relationships

Type	Name	Target	Multiplicity
Association	channel	TransportChannel	1
Generalization		TransportNode	

J.2.6.20 Meta-Class: face.integration.TransportChannel

Description

A TransportChannel is a placeholder for an integrator supplied configuration between transport endpoints. The relationships for the TransportChannel meta-class are listed in Table 170.

Table 170: face.integration.TransportChannel Relationships

Type	Name	Target	Multiplicity
Generalization		Element	

J.2.7 Meta-Package: face.traceability

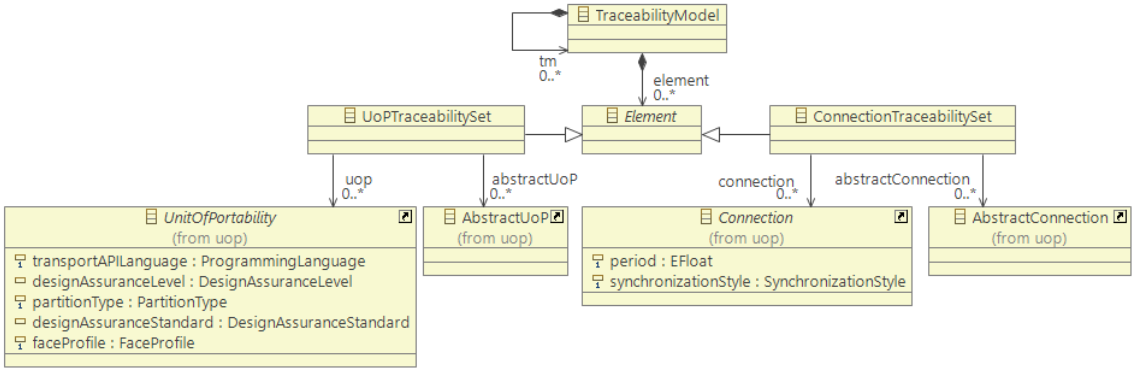

Figure 44: FACE Metamodel "face.traceability" Package

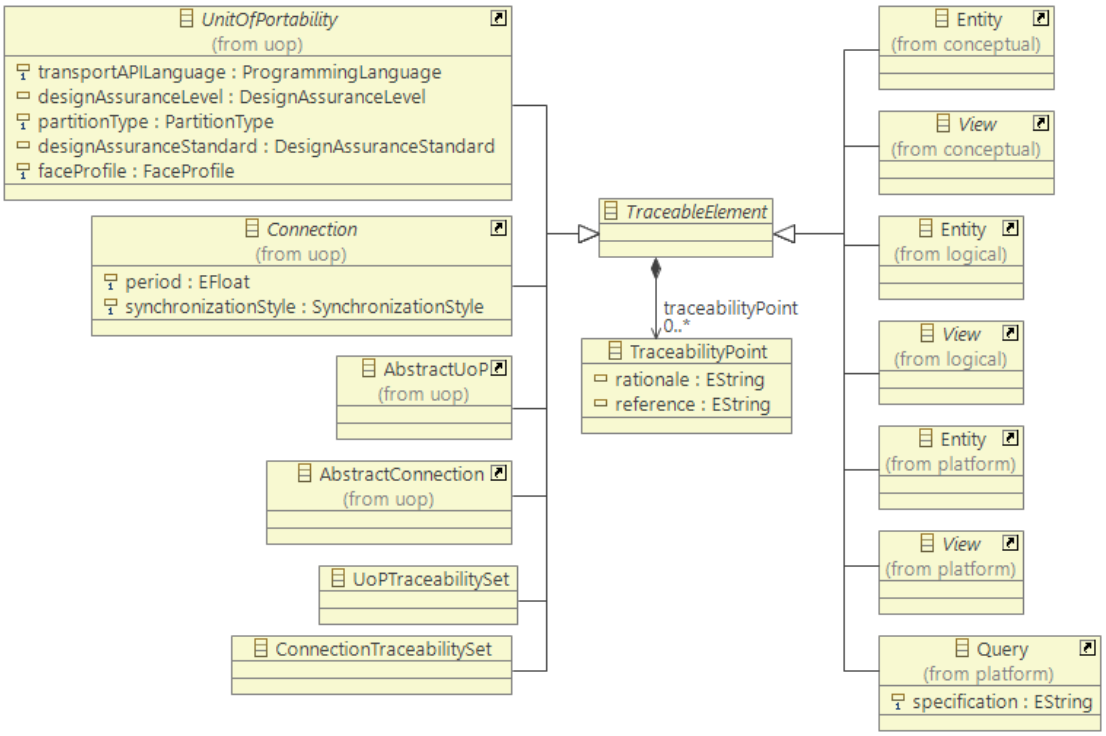

Figure 45: FACE Metamodel "face.traceability" Package: Traceable Elements

J.2.7.1 Meta-Class: face.traceability.TraceabilityModel

Description

A TraceabilityModel is a container for traceability Elements. The relationships for the TraceabilityModel meta-class are listed in Table 171.

Table 171: face.traceability.TraceabilityModel Relationships

Type	Name	Target	Multiplicity
Composition	element	Element	0..*
Composition	tm	TraceabilityModel	0..*
Generalization		face.Element	

J.2.7.2 Meta-Class: face.traceability.Element

Description

A traceability Element is the root type for defining the traceability elements of the FACE Architecure Model. The relationships for the Element meta-class are listed in Table 172.

Table 172: face.traceability.Element Relationships

Type	Name	Target	Multiplicity
Generalization		face.Element	

J.2.7.3 Meta-Class: face.traceability.TraceableElement

Description

A TraceableElement is used to capture traceabilty to other models. The relationships for the TraceableElement meta-class are listed in Table 173.

Table 173: face.traceability.TraceableElement Relationships

Type	Name	Target	Multiplicity
Composition	traceabilityPoint	TraceabilityPoint	0..*

J.2.7.4 Meta-Class: face.traceability.TraceabilityPoint

Description

A TraceabilityPoint is used to document the relationship between a TraceableElement and an external model. The “reference” attribute is a reference to the external model. The “rationale” attribute is used to document the reasoning behind the Trace. The attributes for the TraceabilityPoint meta-class are listed in Table 174.

Table 174: face.traceability.TraceabilityPoint Attributes

Name	Type	Multiplicity
rationale	string	0..1
reference	string	0..1

J.2.7.5 Meta-Class: face.traceability.UoPTraceabilitySet

Description

A UoPTraceabilitySet is used to relate a set of UoPs and/or AbstractUoPs to a set of TraceabilityPoints. The relationships for the UoPTraceabilitySet meta-class are listed in Table 175.

Table 175: face.traceability.UoPTraceabilitySet Relationships

Type	Name	Target	Multiplicity
Association	uop	face.uop.UnitOfPortability	0..*
Association	abstractUoP	face.uop.AbstractUoP	0..*
Generalization		Element	

Type	Name	Target	Multiplicity
Generalization		TraceableElement	

J.2.7.6 Meta-Class: face.traceability.ConnectionTraceabilitySet

Description

A ConnectionTraceabilitySet is used to relate a set of Connections and/or AbstractConnections to a set of TraceabilityPoints. The relationships for the ConnectionTraceabilitySet meta-class are listed in Table 176.

Table 176: face.traceability.ConnectionTraceabilitySet Relationships

Type	Name	Target	Multiplicity
Association	connection	face.uop.Connection	0..*
Association	abstractConnection	face.uop.AbstractConnection	0..*
Generalization		Element	
Generalization		TraceableElement	

J.3 Data Architecture Template Specification Grammar

In the FACE Data Model Language, a UoPModel Template is used to specify the presentation of data in an Open UDDL Platform Query. The Template refers to elements in the Query (i.e., its query specification) through its selected or projected elements. Only elements in the Query's "select" clause may be referenced in the Template. This eliminates ambiguity in how the data is presented. Please note that Templates only control how the data is presented across the TS API; they cannot affect the data selected by the Query.

This section may be modified in subsequent releases. Prior to implementing, please ensure you are using the latest revision of FACE Technical Standard, Edition 3.x, and you have checked to see if any minor releases, corrigenda, or approved corrections have been published.

J.3.1 Data Architecture Template Grammar Definition

The following Extended Backus-Naur Form grammar describes defines the language for a Template specification.

```
template_specification = { using_external_template_statement } ,
structured_template_element_type_decl , { structured_template_element_type_decl
} ;

using_external_template_statement = kw_using , external_template_type_reference
, semicolon ;

structured_template_element_type_decl = main_template_method_decl |
supporting_template_method_decl | union_type_decl ;

main_template_method_decl = main_entity_type_template_method_decl |
main_equivalent_entity_type_template_method_decl ;
```

```
main_entity_type_template_method_decl = kw_main , left_paren , [
primary_entity_type_template_method_parameter ] , [  comma , kw_varargs , [
optional_entity_type_template_method_parameter_list ]  ] , right_paren ,
entity_type_template_method_body ;

primary_entity_type_template_method_parameter =
entity_type_template_method_parameter ;

optional_entity_type_template_method_parameter_list =
entity_type_template_method_parameter , {  comma ,
entity_type_template_method_parameter  } ;

entity_type_template_method_parameter =
entity_type_structured_template_element_declared_parameter_expression ;

main_equivalent_entity_type_template_method_decl = kw_main ,
equivalent_entity_type_template_method_decl ;

supporting_template_method_decl = supporting_entity_type_template_method_decl |
supporting_equivalent_entity_type_template_method_decl ;

supporting_entity_type_template_method_decl = template_element_type_name ,
left_paren , primary_entity_type_template_method_parameter , right_paren ,
entity_type_template_method_body ;

supporting_equivalent_entity_type_template_method_decl =
template_element_type_name , equivalent_entity_type_template_method_decl ;

entity_type_template_method_body = left_brace ,
entity_type_template_method_member , { entity_type_template_method_member } ,
right_brace ;

entity_type_template_method_member =
entity_type_structured_template_element_member ;

equivalent_entity_type_template_method_decl = left_angle_bracket ,
equivalent_entity_type_template_method_parameter_list , right_angle_bracket ,
equivalent_entity_type_template_method_body ;

equivalent_entity_type_template_method_parameter_list =
equivalent_entity_type_template_method_parameter , {  comma ,
equivalent_entity_type_template_method_parameter  } ;

equivalent_entity_type_template_method_body = left_brace ,
equivalent_entity_type_template_method_member , {
equivalent_entity_type_template_method_member } , right_brace ;

equivalent_entity_type_template_method_member =
equivalent_entity_type_template_element_member_statement , semicolon ;

equivalent_entity_type_template_element_member_statement = [
optional_annotation ] ,
designated_equivalent_entity_non_entity_type_characteristic_reference , [
deref , idlstruct_member_reference  ] , [  kw_as ,
structured_template_element_member_name  ] ;

union_type_decl = kw_union , template_element_type_name , left_paren ,
union_parameter , right_paren , union_body ;

union_parameter =
entity_type_structured_template_element_declared_parameter_expression ;

union_body = left_brace , union_switch_statement , right_brace ;
```

```
union_switch_statement = kw_switch , left_paren , discriminator_type ,
right_paren , union_switch_body ;

union_switch_body = left_brace , case_expression , { case_expression } ,
right_brace ;

discriminator_type = idldiscriminator_type |
designated_entity_enumeration_type_characteristic_reference ;

case_expression = case_label , { case_label } , union_member ;

case_label = kw_case , case_label_literal , colon | kw_default , colon;

case_label_literal = enum_literal_reference_expression |
idldiscriminator_type_literal ;

union_member = entity_type_structured_template_element_member ;

entity_type_structured_template_element_member =
entity_type_structured_template_element_member_statement , semicolon ;

entity_type_structured_template_element_member_statement =
designated_entity_characteristic_reference_statement |
structured_template_element_type_reference_statement ;

optional_annotation = at_symbol , kw_optional ;

inline_annotation = at_symbol , kw_inline ;

designated_entity_characteristic_reference_statement =
explicit_designated_entity_non_entity_type_characteristic_reference_expression
| designated_entity_non_entity_type_characteristic_wildcard_reference ;

explicit_designated_entity_non_entity_type_characteristic_reference_expression
= [ optional_annotation ] ,
designated_entity_non_entity_type_characteristic_reference , [  deref ,
idlstruct_member_reference  ] , [  kw_as ,
structured_template_element_member_name  ] ;

structured_template_element_type_reference_statement = inline_annotation ,
structured_template_element_type_reference_expression | [ optional_annotation ]
, structured_template_element_type_reference_expression ,
structured_template_element_member_name ;

structured_template_element_type_reference_expression =
entity_type_structured_template_element_type_reference , left_paren ,
structured_template_element_type_reference_parameter_list , right_paren |
equivalent_entity_type_template_method_reference , left_angle_bracket ,
structured_template_element_type_reference_parameter_list , right_angle_bracket
;

structured_template_element_type_reference_parameter_list =
primary_structured_template_element_type_reference_parameter , {  comma ,
optional_structured_template_element_type_reference_parameter  } ;

primary_structured_template_element_type_reference_parameter =
structured_template_element_type_reference_parameter ;

optional_structured_template_element_type_reference_parameter =
structured_template_element_type_reference_parameter ;
```

```
structured_template_element_type_reference_parameter = [
entity_type_structured_template_element_declared_parameter_reference , equals
] , designated_entity_type_reference_path ;

external_template_type_reference = identifier ;

entity_type_structured_template_element_type_reference = identifier ;

entity_type_structured_template_element_declared_parameter_reference =
identifier ;

entity_type_structured_template_element_declared_parameter_expression =
entity_type_reference , [
entity_type_structured_template_element_declared_parameter_alias ] ;

entity_type_structured_template_element_declared_parameter_alias = identifier ;

structured_template_element_member_name = identifier ;

template_element_type_name = identifier ;

equivalent_entity_type_template_method_reference = identifier ;

equivalent_entity_type_template_method_parameter = identifier ;

designated_equivalent_entity_non_entity_type_characteristic_reference =
equivalent_entity_type_template_method_parameter_reference , period ,
equivalent_entity_type_template_method_characteristic_reference ;

equivalent_entity_type_template_method_parameter_reference = identifier ;

equivalent_entity_type_template_method_characteristic_reference = identifier ;

designated_entity_non_entity_type_characteristic_reference =
designated_entity_type_reference_path , period ,
query_projected_non_entity_type_characteristic_reference |
query_projected_non_entity_type_characteristic_reference_or_alias ;

designated_entity_non_entity_type_characteristic_wildcard_reference = [
designated_entity_type_reference_path , period  ] , asterisk ;

designated_entity_enumeration_type_characteristic_reference =
designated_entity_type_reference_path , period ,
query_projected_enumeration_type_characteristic_reference |
query_projected_enumeration_type_characteristic_reference_or_alias;

designated_entity_type_reference_path = [
explicit_entity_type_reference_join_path ] , designated_entity_type_reference ;

explicit_entity_type_reference_join_path = (  join_path_entity_type_reference ,
period  ) , {  join_path_entity_type_reference , period  };

join_path_entity_type_reference = qualified_entity_type_reference ;

designated_entity_type_reference = qualified_entity_type_reference ;

qualified_entity_type_reference = entity_type_reference , [
entity_characteristic_value_qualifier ] ;

entity_type_reference = query_selected_entity_type_reference_or_alias ;

entity_characteristic_value_qualifier = query_where_clause_criteria ;
```

```
idlstruct_member_reference = identifier ;

enum_literal_reference_expression = left_brace , enumeration_type_reference ,
colon , enumeration_literal_reference , right_brace ;

enumeration_type_reference = identifier ;

enumeration_literal_reference = identifier ;

(* criteria is a production rule defined in the query grammar specified in
Appendix J.3.1. *)
query_where_clause_criteria = left_bracket , criteria , right_bracket ;

query_projected_non_entity_type_characteristic_reference_or_alias = identifier
;

query_projected_non_entity_type_characteristic_reference = identifier ;

query_projected_enumeration_type_characteristic_reference_or_alias = identifier
;

query_projected_enumeration_type_characteristic_reference = identifier ;

query_selected_entity_type_reference_or_alias = identifier ;

idldiscriminator_type = idlunsigned_int | idlboolean ;

idlunsigned_int = idlunsigned_short | idlunsigned_long | idlunsigned_long_long
;

idlunsigned_short = kw_unsigned , kw_short ;

idlunsigned_long = kw_unsigned , kw_long ;

idlunsigned_long_long = kw_unsigned , kw_long , kw_long ;

idlboolean = kw_boolean ;

idldiscriminator_type_literal = idlinteger_literal | idloctal_literal |
idlhex_literal | idlboolean_literal ;

idlinteger_literal = zero_digit_literal | positive_digit_literal , {
digit_literal } ;

idloctal_literal = zero_digit_literal , octal_digit_literal , {
octal_digit_literal } ;

idlhex_literal = zero_digit_literal , ( "x" | "X" ) hex_digit_literal , {
hex_digit_literal } ;

idlboolean_literal = kw_true | kw_false ;

kw_main = "MAIN" | "main" ;

kw_using = "USING" | "using" ;

kw_as = "AS" | "as" ;

kw_varargs = "VARARGS" | "varargs" ;

kw_optional = "OPTIONAL" | "optional" ;

kw_inline = "INLINE" | "inline" ;
```

```
kw_union = "UNION" | "union" ;

kw_switch = "SWITCH" | "switch" ;

kw_case = "CASE" | "case" ;

kw_default = "DEFAULT" | "default" ;

kw_boolean = "BOOLEAN" | "boolean" ;

kw_short = "SHORT" | "short" ;

kw_long = "LONG" | "long" ;

kw_unsigned = "UNSIGNED" | "unsigned" ;

kw_true = "TRUE" ;

kw_false = "FALSE" ;

deref = "->" ;

asterisk = "*" ;

left_brace = "{" ;

right_brace = "}" ;

left_paren = "(" ;

right_paren = ")" ;

left_bracket = "[" ;

right_bracket = "]" ;

left_angle_bracket = "<" ;

right_angle_bracket = ">" ;

comma = "," ;

colon = ":" ;

semicolon = ";" ;

period = "." ;

at_symbol = "@" ;

equals = "=" ;

octal_digit_literal = zero_digit_literal | "1" | "2" | "3" | "4" | "5" | "6" |
"7" ;

hex_digit_literal = digit_literal | "a" | "b" | "c" | "d" | "e" | "f" | "A" |
"B" | "C" | "D" | "E" | "F" ;

identifier = char_literal , {  char_literal | digit_literal  } ;

char_literal = "a" | "b"| "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" |
"l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" |
```

```
"y" | "z" | "A" | "B"| "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" |
"L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" |
"Y" | "Z" | "_" ;

digit_literal = zero_digit_literal | positive_digit_literal ;

zero_digit_literal = "0" ;

positive_digit_literal = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

J.4 Data Architecture Template Rules

Legend

term	The name of a term in the Template grammar. A set of rules may follow a term. These rules apply to all instances of this term in a modeled FACE.UOP.TEMPLATE. This term is referred to as the rule's "context term" in this legend.
template_term_name	The name of a term in the Template grammar. In a rule, it represents an instance of the named term in a modeled FACE.UOP.TEMPLATE. (Rules reference the context term as well as other terms in its expression.)
METATYPE_NAME	The name of a metatype. In a rule, it represents an instance of the named metatype in a model.
QUERY_TERM_NAME	The name of a term in the Query grammar. In a rule, it represents an instance of the named term in the **specification** of the DATAMODEL.PLATFORM.QUERY (which itself is a ***QUERY_SPECIFICATION***) that is the **boundQuery** of the FACE.UOP.TEMPLATE whose **specification** is the ***template_specification*** to which the rules below are being applied.
property_name	The name of metatype's property. In a rule, it represents a value associated with the named property of an instance of a metatype in a model.
"literal"	Represents a literal value.

template_specification

There must be one and only one ***main_template_method_decl***, either a ***main_template_method_decl*** or a ***main_equivalent_entity_type_template_method_decl***.

If ***main_template_method_decl*** is a ***main_equivalent_entity_type_template_method_decl***, then:

- A ***supporting_template_method_decl*** must not be specified
- A ***union_type_decl*** must not be specified
- A ***using_external_template_statement*** must not be specified
- The **boundQuery** of the FACE.UOP.TEMPLATE whose **specification** is this ***template_specification*** must not be set
- The **effectiveQuery** of the FACE.UOP.TEMPLATE whose **specification** is this ***template_specification*** must not be set

If ***main_template_method_decl*** is a ***main_entity_type_template_method_decl***, then:

- The **boundQuery** of the FACE.UOP.TEMPLATE whose **specification** is this ***template_specification*** must be set
- An ***external_template_type_reference*** must match the **name** of a FACE.UOP.TEMPLATE
- An ***external_template_type_reference*** must not match the **name** of the FACE.UOP.TEMPLATE whose **specification** is this ***template_specification***
- If an external_template_type_reference is specified, then:

  — Let THIS_TEMPLATE_NAME = the **name** of the FACE.UOP.TEMPLATE whose **specification** is this ***template_specification***

  — Let SS_0 = the **specification**s (n.b. which each are ***template_specification***s) of each and every FACE.UOP.TEMPLATE whose **name** is the same as an ***external_template_type_reference*** in this ***template_specification***

  — A **specification** (i.e., ***template_specification***) in SS_0 must not have an ***external_template_type_reference*** that matches by name THIS_TEMPLATE_NAME

  — Let n = 0
- Recursively, if an ***external_template_type_reference*** is specified in a **specification** in SS_n, then:

  — Let SS_{n+1} = the **specification**s of each and every FACE.UOP.TEMPLATE whose **name** is the same as an ***external_template_type_reference*** in SS_n

  — A **specification** (i.e., ***template_specification***) in SS_{n+1} must not have an ***external_template_type_reference*** that matches by name THIS_TEMPLATE_NAME
- The ***external_template_type_reference*** of two or more ***using_external_template_statement***s must not be the same
- The ***template_element_type_name*** of a ***supporting_template_method_decl*** or ***union_type_decl***:

  — Must not be the same as any ***using_external_template_statement***'s ***external_template_type_reference***

  — Must not be the same as the **name** of the FACE.UOP.TEMPLATE whose **specification** is this ***template_specification***

  — Must not be the same as the ***template_element_type_name*** of any other ***supporting_template_method_decl*** or ***union_type_decl***

  — Must not be the same as the **name** of a **type** of any DATAMODEL.PLATFORM.CHARACTERISTIC referenced by a ***designated_entity_characteristic_reference_statement***
- For a given ***supporting_template_method_decl*** or ***union_type_decl***:

  — Given a sequence of ***entity_type_structured_template_element_type_references*** from a ***main_entity_type_template_method_decl*** to that ***supporting_template_method_decl*** or ***union_type_decl***:

— Let CP = a normalized, canonical ***designated_entity_type_reference_path*** from the ***query_selected_entity_type_reference_or_alias*** of the ***main_entity_type_template_method_decl***'s ***primary_entity_type_template_method_parameter*** to the ***query_selected_entity_type_reference_or_alias*** of that ***supporting_template_method_decl***'s or ***union_type_decl***'s ***entity_type_structured_template_element_declared_parameter_expression*** constructed by combining head-to-tail the ***designated_entity_type_reference_path*** associated with each ***entity_type_structured_template_element_type_reference*** in sequence and normalized by removing any adjacent ***query_selected_entity_type_reference_or_alias***es that match either the ***SELECTED_ENTITY_ALIAS*** or ***ENTITY_TYPE_REFERENCE*** of the same ***SELECTED_ENTITY*** in the ***FROM_CLAUSE*** of the outermost ***QUERY_STATEMENT***

- Two or more ***query_selected_entity_type_reference_or_alias***es in CP must not match either the ***SELECTED_ENTITY_ALIAS*** or ***ENTITY_TYPE_REFERENCE*** of the same ***SELECTED_ENTITY*** in the ***FROM_CLAUSE*** of the outermost ***QUERY_STATEMENT***
- The CP for all sequences of ***entity_type_structured_template_element_type_reference***s from a ***main_entity_type_template_method_decl*** to that ***supporting_template_method_decl*** or ***union_type_decl*** must be the same

If an ***entity_type_structured_template_element_member*** is ***structured_template_element_type_reference_statement***, then:

- If ***entity_type_structured_template_element_type_reference*** is specified, then ***entity_type_structured_template_element_type_reference*** must match by name:

  — The ***template_element_type_name*** of a ***supporting_entity_type_template_method_decl*** or a ***union_type_decl*** of this ***template_specification***, or

  — An ***external_template_type_reference*** that is the **name** of the FACE.UOP.TEMPLATE whose ***main_template_method_decl*** is ***main_entity_type_template_method_decl***

- If ***equivalent_entity_type_template_method_reference*** is specified, then ***equivalent_entity_type_template_method_reference*** must match by name:

  — The ***template_element_type_name*** of a ***supporting_equivalent_entity_type_template_method_decl*** of this ***template_specification***, or

  — An ***external_template_type_reference*** that is the **name** of the FACE.UOP.TEMPLATE whose ***main_template_method_decl*** is ***main_equivalent_entity_type_template_method_decl***

using_external_template_statement

// No contextually-relevant rules for instances of this term.

structured_template_element_type_decl

// No contextually-relevant rules for instances of this term.

main_template_method_decl

// No contextually-relevant rules for instances of this term.

main_entity_type_template_method_decl

The ***query_selected_entity_type_reference_or_alias*** of two or more ***entity_type_structured_template_element_declared_parameter_expression***s must not match by name either the ***SELECTED_ENTITY_ALIAS*** or ***ENTITY_TYPE_REFERENCE*** of the same ***SELECTED_ENTITY*** in the ***FROM_CLAUSE*** of the outermost ***QUERY_STATEMENT***.

All ***entity_type_structured_template_element_declared_parameter_alias***es must be unique.

A ***primary_entity_type_template_method_parameter*** must specified.

kw_varargs must not be specified.

If ***entity_type_structured_template_element_member*** is ***explicit_designated_entity_non_entity_type_characteristic_reference_expression***:

- If ***designated_entity_non_entity_type_characteristic_reference*** is ***query_projected_non_entity_type_characteristic_reference_or_alias***, then:

  — One of the conditions below must be satisfied, resolving it to the DATAMODEL.PLATFORM.CHARACTERISTIC identified by the first rule satisfied in priority order:

    - ***query_projected_non_entity_type_characteristic_reference_or_alias*** matches the **rolename** of a DATAMODEL.PLATFORM.CHARACTERISTIC that is in the ***PROJECTED_CHARACTERISTIC_LIST*** of the outermost ***QUERY_STATEMENT*** and is a **composition** of the ***SELECTED_ENTITY*** referenced by the ***primary_entity_type_template_method_parameter***'s ***query_selected_entity_type_reference_or_alias***

    - ***query_projected_non_entity_type_characteristic_reference_or_alias*** matches by name a ***PROJECTED_CHARACTERISTIC_ALIAS*** of an ***EXPLICIT_SELECTED_ENTITY_CHARACTERISTIC_REFERENCE*** in the ***PROJECTED_CHARACTERISTIC_LIST*** of the outermost ***QUERY_STATEMENT***

    - ***query_projected_non_entity_type_characteristic_reference_or_alias*** matches the **rolename** of one and only one DATAMODEL.PLATFORM.CHARACTERISTIC in the ***PROJECTED_CHARACTERISTIC_LIST*** of the outermost ***QUERY_STATEMENT***

  If the DATAMODEL.PLATFORM.CHARACTERISTIC (determined above) is not a **composition** of the ***SELECTED_ENTITY*** referenced by ***primary_entity_type_template_method_parameter***'s ***query_selected_entity_type_reference_or_alias***, then there must be an unambiguous

designated_entity_type_reference_path (inferred from the ***ENTITY_EXPRESSION*** of the outermost ***QUERY_STATEMENT***) from the ***SELECTED_ENTITY*** referenced by the ***primary_entity_type_template_method_parameter***'s ***query_selected_entity_type_reference_or_alias*** to the ***SELECTED_ENTITY*** composing that DATAMODEL.PLATFORM.CHARACTERISTIC.

- Otherwise, ***designated_entity_non_entity_type_characteristic_reference*** is not ***query_projected_non_entity_type_characteristic_reference_or_alias***, then:

  — If the first ***query_selected_entity_type_reference_or_alias*** in ***designated_entity_type_reference_path*** does not match by name the ***primary_entity_type_template_method_parameter***'s ***query_selected_entity_type_reference_or_alias*** or, if specified, its ***entity_type_structured_template_element_declared_parameter_alias***, then there must be an unambiguous ***designated_entity_type_reference_path*** (inferred from the ***ENTITY_EXPRESSION*** of the outermost ***QUERY_STATEMENT***) from the ***SELECTED_ENTITY*** referenced by the ***primary_entity_type_template_method_parameter***'s ***query_selected_entity_type_reference_or_alias*** to the ***SELECTED_ENTITY*** referenced by the first ***query_selected_entity_type_reference_or_alias*** in the ***explicit_designated_entity_non_entity_type_characteristic_reference_expression***'s ***designated_entity_type_reference_path***

- If ***idlstruct_member_reference*** is specified, then the **type** of the DATAMODEL.PLATFORM.CHARACTERISTIC referenced by ***explicit_designated_entity_non_entity_type_characteristic_reference_expression*** must be a DATAMODEL.PLATFORM.STRUCT, and ***idlstruct_member_reference*** must match the **rolename** of a DATAMODEL.PLATFORM.STRUCTMEMBER that is a **member** of that DATAMODEL.PLATFORM.STRUCT

If ***entity_type_structured_template_element_member*** is ***designated_entity_non_entity_type_characteristic_wildcard_reference***:

- If ***designated_entity_type_reference_path*** is specified:

  — If the first ***query_selected_entity_type_reference_or_alias*** in ***explicit_entity_type_reference_join_path*** matches by name the ***primary_entity_type_template_method_parameter***'s ***query_selected_entity_type_reference_or_alias*** or, if specified, its ***entity_type_structured_template_element_declared_parameter_alias***, then:

    - There must at least one DATAMODEL.PLATFORM.CHARACTERISTIC in the ***PROJECTED_CHARACTER_LIST*** of the outermost ***QUERY_STATEMENT*** that is a **composition** of the ***SELECTED_ENTITY*** referenced by the ***designated_entity_type_reference_path***'s ***designated_entity_type_reference***'s ***query_selected_entity_type_reference_or_alias***

  — Otherwise, the first ***query_selected_entity_type_reference_or_alias*** in ***explicit_entity_type_reference_join_path*** does not match by name the ***primary_entity_type_template_method_parameter***'s ***query_selected_entity_type_reference_or_alias*** or, if specified, its ***entity_type_structured_template_element_declared_parameter_alias***, then:

- There must be an unambiguous ***designated_entity_type_reference_path*** (inferred from the ***ENTITY_EXPRESSION*** of the outermost ***QUERY_STATEMENT***) from the ***SELECTED_ENTITY*** referenced by the ***primary_entity_type_template_method_parameter***'s ***query_selected_entity_type_reference_or_alias*** to the ***SELECTED_ENTITY*** referenced by the first ***query_selected_entity_type_reference_or_alias*** in the ***designated_entity_non_entity_type_characteristic_wildcard_reference***'s ***designated_entity_type_reference_path***

- There must at least one DATAMODEL.PLATFORM.CHARACTERISTIC in the ***PROJECTED_CHARACTER_LIST*** of the outermost ***QUERY_STATEMENT*** that is a **composition** of the ***SELECTED_ENTITY*** referenced by the fully inferred ***designated_entity_type_reference_path***'s ***designated_entity_type_reference***'s ***query_selected_entity_type_reference_or_alias***.

- Otherwise, ***designated_entity_type_reference_path*** is not specified, then:

  — There must at least one DATAMODEL.PLATFORM.CHARACTERISTIC in the ***PROJECTED_CHARACTER_LIST*** of the outermost ***QUERY_STATEMENT*** that is a **composition** of the ***SELECTED_ENTITY*** referenced by the ***primary_entity_type_template_method_parameter***'s ***query_selected_entity_type_reference_or_alias***

If ***entity_type_structured_template_element_member*** is ***structured_template_element_type_reference_statement***, then:

- If ***entity_type_structured_template_element_type_reference*** is specified, then:

  — An ***entity_type_structured_template_element_declared_parameter_reference*** must not be specified

  — An ***optional_structured_template_element_type_reference_parameter*** must not be specified

- If ***entity_type_structured_template_element_type_reference*** matches by name the ***template_element_type_name*** of a ***supporting_entity_type_template_method_decl***, then:

  — The ***SELECTED_ENTITY*** referenced by the ***structured_template_element_type_reference_expression***'s ***primary_structured_template_element_type_reference_parameter***'s ***designated_entity_type_reference_path***'s ***designated_entity_type_reference***'s ***query_selected_entity_type_reference_or_alias*** must be the same as the ***SELECTED_ENTITY*** referenced by the ***supporting_template_method_decl***'s ***primary_entity_type_template_method_parameter***'s ***query_selected_entity_type_reference_or_alias***

- If ***inline_annotation*** is specified, then:

  — For each adjacent pair of ***query_selected_entity_type_reference_or_alias***es in the ***primary_structured_template_element_type_reference_parameter***'s explicitly specified or unambiguously inferred ***designated_entity_type_reference_path***:

- Let L = the ***SELECTED_ENTITY*** referenced by the left ***query_selected_entity_type_reference_or_alias*** of the pair
- Let R = the ***SELECTED_ENTITY*** referenced by the right (i.e., immediately following) ***query_selected_entity_type_reference_or_alias*** of the pair
- Let J = the *"Join_Characteristic"* for L and R

— If J is a **composition** or **participant** of L, then:

- The upper bound is the **upperBound** of J
- If R is a ***SELECTED_ENTITY*** that is the ***SELECTED_ENTITY_REFERENCE*** of a ***SELECTED_ENTITY_CHARACTERISTIC_REFERENCE CHARARACTERISTIC_BASIS***, then the lower bound is 0
- Otherwise, R is not a ***SELECTED_ENTITY*** that is the ***SELECTED_ENTITY_REFERENCE*** of a ***SELECTED_ENTITY_CHARACTERISTIC_REFERENCE CHARARACTERISTIC_BASIS***; in this case, the lower bound is the **lowerBound** of J

— If J is a **composition** R, then:

- The upper bound is 1
- If L is a ***SELECTED_ENTITY*** that is the ***SELECTED_ENTITY_REFERENCE*** of a ***SELECTED_ENTITY_CHARACTERISTIC_REFERENCE CHARARACTERISTIC_BASIS***, then the lower bound is 0
- Otherwise, L is not a ***SELECTED_ENTITY*** that is the ***SELECTED_ENTITY_REFERENCE*** of a ***SELECTED_ENTITY_CHARACTERISTIC_REFERENCE CHARARACTERISTIC_BASIS***; in this case, the lower bound is 1

— If J is a **participant** of R, then:

- The upper bound is the **sourceUpperBound** of J
- If L is a ***SELECTED_ENTITY*** that is the ***SELECTED_ENTITY_REFERENCE*** of a ***SELECTED_ENTITY_CHARACTERISTIC_REFERENCE CHARARACTERISTIC_BASIS***, then the lower bound for J is 0
- Otherwise, L is not a ***SELECTED_ENTITY*** that is the ***SELECTED_ENTITY_REFERENCE*** of a ***SELECTED_ENTITY_CHARACTERISTIC_REFERENCE CHARARACTERISTIC_BASIS***; in this case, the lower bound is the **sourceLowerBound** of J

Using the definitions above:

— The lower bound and the upper bound for all Js in the ***primary_structured_template_element_type_reference_parameter***'s explicitly specified or unambiguously inferred ***designated_entity_type_reference_path*** must be 1

- If ***entity_type_structured_template_element_type_reference*** matches by name the ***template_element_type_name*** of a ***union_type_decl***, then:

  — The ***SELECTED_ENTITY*** referenced by the ***structured_template_element_type_reference_expression***'s ***primary_structured_template_element_type_reference_parameter***'s ***designated_entity_type_reference_path***'s ***designated_entity_type_reference***'s ***query_selected_entity_type_reference_or_alias*** must be the same as the ***SELECTED_ENTITY*** referenced by the ***union_type_decl***'s ***union_parameter***'s ***query_selected_entity_type_reference_or_alias***

  — ***inline_annotation*** must not be specified

- If ***entity_type_structured_template_element_type_reference*** matches by name an ***external_template_type_reference***, then:

  — ***inline_annotation*** must not be specified

  — Let T = the FACE.UOP.TEMPLATE whose **name** is ***external_template_type_reference***

  — Let M = ***main_entity_type_template_method_decl*** in T

  — Let Q = the **specification** of the DATAMODEL.PLATFORM.QUERY that is the **boundQuery** of T

  — Let RE = the ***SELECTED_ENTITY*** referenced by the ***primary_structured_template_element_type_reference_parameter***'s ***designated_entity_type_reference***'s ***query_selected_entity_type_reference_or_alias***

  — Let DE = the ***SELECTED_ENTITY*** in the ***FROM_CLAUSE*** of the outermost ***QUERY_STATEMENT*** of Q referenced by M's ***primary_entity_type_template_method_parameter***'s ***query_selected_entity_type_reference_or_alias***

  — RE's ***ENTITY_TYPE_REFERENCE*** must match by name DE's ***ENTITY_TYPE_REFERENCE***

- If ***equivalent_entity_type_template_method_reference*** is specified, then:

  — ***inline_annotation*** must be specified

  — Let M = an ***equivalent_entity_type_template_method_decl*** where:

    - If ***equivalent_entity_type_template_method_reference*** matches by name an ***external_template_type_reference***, then M is the ***equivalent_entity_type_template_method_decl*** of the ***main_equivalent_entity_type_template_method_decl*** of the FACE.UOP.TEMPLATE whose **name** is ***external_template_type_reference***

        - Otherwise, M is the ***equivalent_entity_type_template_method_decl*** of the ***supporting_equivalent_entity_type_template_method_decl*** whose ***template_element_type_name*** is ***equivalent_entity_type_template_method_reference***

- If ***entity_type_structured_template_element_declared_parameter_reference*** is specified for the first ***structured_template_element_type_reference_parameter*** in ***structured_template_element_type_reference_parameter_list***, then:

  — An ***entity_type_structured_template_element_declared_parameter_reference*** must be specified for all ***structured_template_element_type_reference_parameter***s in a ***structured_template_element_type_reference_parameter_list***

  — All ***entity_type_structured_template_element_declared_parameter_reference***s must be unique

  — There must be one ***structured_template_element_type_reference_parameter*** in ***structured_template_element_type_reference_parameter_list*** for each ***equivalent_entity_type_template_method_parameter*** in ***equivalent_entity_type_template_method_parameter_list*** of M

  — For each ***structured_template_element_type_reference_parameter*** in ***structured_template_element_type_reference_parameter_list***:

    - The ***structured_template_element_type_reference_parameter*** 's ***entity_type_structured_template_element_declared_parameter_reference*** must match by name an ***equivalent_entity_type_template_method_parameter*** of M

  — For each ***equivalent_entity_type_template_method_member*** of M whose ***equivalent_entity_type_template_method_parameter_reference*** matches by name the ***structured_template_element_type_reference_parameter*** 's ***entity_type_structured_template_element_declared_parameter_reference***:

    - There must a **composition** in the ***SELECTED_ENTITY*** referenced by the ***structured_template_element_type_reference_parameter***'s ***designated_entity_type_reference***'s ***query_selected_entity_type_reference_or_alias*** whose **rolename** matches by name the ***equivalent_entity_type_template_method_member***'s ***equivalent_entity_type_template_method_characteristic_reference***

    - If ***idlstruct_member_reference*** is specified, then the **type** of the **composition** in the ***SELECTED_ENTITY*** referenced by the ***structured_template_element_type_reference_parameter***'s ***designated_entity_type_reference***'s ***query_selected_entity_type_reference_or_alias*** must be a DATAMODEL.PLATFORM.STRUCT, and ***idlstruct_member_reference*** must match the **rolename** of a DATAMODEL.PLATFORM.STRUCTMEMBER that is a **member** of that DATAMODEL.PLATFORM.STRUCT

Otherwise, an ***entity_type_structured_template_element_declared_parameter_reference*** must not be specified for any ***structured_template_element_type_reference_parameter***s in a ***structured_template_element_type_reference_parameter_list***, then:

— There must be one ***structured_template_element_type_reference_parameter*** in ***structured_template_element_type_reference_parameter_list*** for each ***equivalent_entity_type_template_method_parameter*** in ***equivalent_entity_type_template_method_parameter_list*** of M

— For each ***structured_template_element_type_reference_parameter*** in ***structured_template_element_type_reference_parameter_list***:

- Let RP = the current ***structured_template_element_type_reference_parameter***
- Let EP = the ***equivalent_entity_type_template_method_parameter_reference*** whose position in the ***equivalent_entity_type_template_method_parameter_list*** of M is the same as RP in ***structured_template_element_type_reference_parameter_list***

— For each ***equivalent_entity_type_template_method_member*** of M whose ***equivalent_entity_type_template_method_parameter_reference*** matches by name EP:

- There must a **composition** in the ***SELECTED_ENTITY*** referenced by RP's ***designated_entity_type_reference***'s ***query_selected_entity_type_reference_or_alias*** whose **rolename** matches by name the ***equivalent_entity_type_template_method_member***'s ***equivalent_entity_type_template_method_characteristic_reference***
- If ***idlstruct_member_reference*** is specified, then the **type** of the **composition** in the ***SELECTED_ENTITY*** referenced by RP's ***designated_entity_type_reference***'s ***query_selected_entity_type_reference_or_alias*** must be a DATAMODEL.PLATFORM.STRUCT, and ***idlstruct_member_reference*** must match the **rolename** of a DATAMODEL.PLATFORM.STRUCTMEMBER that is a **member** of that DATAMODEL.PLATFORM.STRUCT

primary_entity_type_template_method_parameter

// No contextually-relevant rules for instances of this term.

optional_entity_type_template_method_parameter_list

// No contextually-relevant rules for instances of this term.

entity_type_template_method_parameter

// No contextually-relevant rules for instances of this term.

main_equivalent_entity_type_template_method_decl

// No contextually-relevant rules for instances of this term.

supporting_template_method_decl

// No contextually-relevant rules for instances of this term.

supporting_entity_type_template_method_decl

If ***entity_type_structured_template_element_member*** is ***explicit_designated_entity_non_entity_type_characteristic_reference_expression***:

- If ***designated_entity_non_entity_type_characteristic_reference*** is ***query_projected_non_entity_type_characteristic_reference_or_alias***, then:

  — One of the conditions below must be satisfied, resolving it to the DATAMODEL.PLATFORM.CHARACTERISTIC identified by the first rule satisfied in priority order:

    - ***query_projected_non_entity_type_characteristic_reference_or_alias*** matches the **rolename** of a DATAMODEL.PLATFORM.CHARACTERISTIC that is in the ***PROJECTED_CHARACTERISTIC_LIST*** of the outermost ***QUERY_STATEMENT*** and is a **composition** of the ***SELECTED_ENTITY*** referenced by the ***primary_entity_type_template_method_parameter***'s ***query_selected_entity_type_reference_or_alias***
    - ***query_projected_non_entity_type_characteristic_reference_or_alias*** matches by name a ***PROJECTED_CHARACTERISTIC_ALIAS*** of an ***EXPLICIT_SELECTED_ENTITY_CHARACTERISTIC_REFERENCE*** in the ***PROJECTED_CHARACTERISTIC_LIST*** of the outermost ***QUERY_STATEMENT***
    - ***query_projected_non_entity_type_characteristic_reference_or_alias*** matches the **rolename** of one and only one DATAMODEL.PLATFORM.CHARACTERISTIC in the ***PROJECTED_CHARACTERISTIC_LIST*** of the outermost ***QUERY_STATEMENT***

  If the DATAMODEL.PLATFORM.CHARACTERISTIC (determined above) is not a **composition** of the ***SELECTED_ENTITY*** referenced by ***primary_entity_type_template_method_parameter***'s ***query_selected_entity_type_reference_or_alias***, then there must be an unambiguous ***designated_entity_type_reference_path*** (inferred from the ***ENTITY_EXPRESSION*** of the outermost ***QUERY_STATEMENT***) from the ***SELECTED_ENTITY*** referenced by the ***primary_entity_type_template_method_parameter***'s ***query_selected_entity_type_reference_or_alias*** to the ***SELECTED_ENTITY*** composing that DATAMODEL.PLATFORM.CHARACTERISTIC.

- Otherwise, ***designated_entity_non_entity_type_characteristic_reference*** is not ***query_projected_non_entity_type_characteristic_reference_or_alias***, then:

  — If the first ***query_selected_entity_type_reference_or_alias*** in ***designated_entity_type_reference_path*** does not match by name the

primary_entity_type_template_method_parameter's ***query_selected_entity_type_reference_or_alias*** or, if specified, its ***entity_type_structured_template_element_declared_parameter_alias***, then there must be an unambiguous ***designated_entity_type_reference_path*** (inferred from the ***ENTITY_EXPRESSION*** of the outermost ***QUERY_STATEMENT***) from the ***SELECTED_ENTITY*** referenced by the ***primary_entity_type_template_method_parameter***'s ***query_selected_entity_type_reference_or_alias*** to the ***SELECTED_ENTITY*** referenced by the first ***query_selected_entity_type_reference_or_alias*** in the ***explicit_designated_entity_non_entity_type_characteristic_reference_expression***'s ***designated_entity_type_reference_path***

- If ***idlstruct_member_reference*** is specified, then the **type** of the DATAMODEL.PLATFORM.CHARACTERISTIC referenced by ***explicit_designated_entity_non_entity_type_characteristic_reference_expression*** must be a DATAMODEL.PLATFORM.STRUCT, and ***idlstruct_member_reference*** must match the **rolename** of a DATAMODEL.PLATFORM.STRUCTMEMBER that is a **member** of that DATAMODEL.PLATFORM.STRUCT

If ***entity_type_structured_template_element_member*** is ***designated_entity_non_entity_type_characteristic_wildcard_reference***:

- If ***designated_entity_type_reference_path*** is specified:

  — If the first ***query_selected_entity_type_reference_or_alias*** in ***explicit_entity_type_reference_join_path*** matches by name the ***primary_entity_type_template_method_parameter***'s ***query_selected_entity_type_reference_or_alias*** or, if specified, its ***entity_type_structured_template_element_declared_parameter_alias***, then:

    - There must at least one DATAMODEL.PLATFORM.CHARACTERISTIC in the ***PROJECTED_CHARACTER_LIST*** of the outermost ***QUERY_STATEMENT*** that is a **composition** of the ***SELECTED_ENTITY*** referenced by the ***designated_entity_type_reference_path***'s ***designated_entity_type_reference***'s ***query_selected_entity_type_reference_or_alias***

  — Otherwise, the first ***query_selected_entity_type_reference_or_alias*** in ***explicit_entity_type_reference_join_path*** does not match by name the ***primary_entity_type_template_method_parameter***'s ***query_selected_entity_type_reference_or_alias*** or, if specified, its ***entity_type_structured_template_element_declared_parameter_alias***, then:

    - There must be an unambiguous ***designated_entity_type_reference_path*** (inferred from the ***ENTITY_EXPRESSION*** of the outermost ***QUERY_STATEMENT***) from the ***SELECTED_ENTITY*** referenced by the ***primary_entity_type_template_method_parameter***'s ***query_selected_entity_type_reference_or_alias*** to the ***SELECTED_ENTITY*** referenced by the first ***query_selected_entity_type_reference_or_alias*** in the ***designated_entity_non_entity_type_characteristic_wildcard_reference***'s ***designated_entity_type_reference_path***

- There must at least one DATAMODEL.PLATFORM.CHARACTERISTIC in the ***PROJECTED_CHARACTER_LIST*** of the outermost ***QUERY_STATEMENT*** that is a **composition** of the ***SELECTED_ENTITY*** referenced by the fully inferred ***designated_entity_type_reference_path***'s ***designated_entity_type_reference***'s ***query_selected_entity_type_reference_or_alias***

- Otherwise, ***designated_entity_type_reference_path*** is not specified, then:

  — There must at least one DATAMODEL.PLATFORM.CHARACTERISTIC in the ***PROJECTED_CHARACTER_LIST*** of the outermost ***QUERY_STATEMENT*** that is a **composition** of the ***SELECTED_ENTITY*** referenced by the ***primary_entity_type_template_method_parameter***'s ***query_selected_entity_type_reference_or_alias***

If ***entity_type_structured_template_element_member*** is ***structured_template_element_type_reference_statement***, then:

- If ***entity_type_structured_template_element_type_reference*** is specified, then:

  — An ***entity_type_structured_template_element_declared_parameter_referenc***e must not be specified

  — An ***optional_structured_template_element_type_reference_parameter*** must not be specified

- If ***entity_type_structured_template_element_type_reference*** matches by name the ***template_element_type_name*** of a ***supporting_entity_type_template_method_decl***, then:

  — The ***SELECTED_ENTITY*** referenced by the ***structured_template_element_type_reference_expression***'s ***primary_structured_template_element_type_reference_parameter***'s ***designated_entity_type_reference_path***'s ***designated_entity_type_reference***'s ***query_selected_entity_type_reference_or_alias*** must be the same as the ***SELECTED_ENTITY*** referenced by the ***supporting_template_method_decl***'s ***primary_entity_type_template_method_parameter***'s ***query_selected_entity_type_reference_or_alias***

  — If ***inline_annotation*** is specified, then:

    - Let JS = the cumulative set of all "Join_Characteristic"s (as defined in the rules for term **designated_entity_type_reference_path**) in the canonical **designated_entity_type_reference_path** from the **query_selected_entity_type_reference_or_alias** of the **main_entity_type_template_method_decl**'s **primary_entity_type_template_method_parameter** to the **query_selected_entity_type_reference_or_alias** of this **supporting_template_method_decl**'s **entity_type_structured_template_element_declared_parameter_expression** (constructed by combining head-to-tail the **designated_entity_type_reference_path** associated with each **entity_type_structured_template_element_type_reference** in sequence)

— For each adjacent pair of ***query_selected_entity_type_reference_or_alias***es in the ***primary_structured_template_element_type_reference_parameter***'s explicitly specified or unambiguously inferred ***designated_entity_type_reference_path***:

- Let L = the ***SELECTED_ENTITY*** referenced by the left ***query_selected_entity_type_reference_or_alias*** of the pair
- Let R = the ***SELECTED_ENTITY*** referenced by the right (i.e., immediately following) ***query_selected_entity_type_reference_or_alias*** of the pair
- Let J = the *"Join_Characteristic"* for L and R

— If J is a **composition** or **participant** of L, then:

- If J is in JS, then both the lower bound and upper bound for J is 1
- If J is not in JS, then the lower bound is the **lowerBound** of J and upper bound is the **upperBound** of J
- If R is a ***SELECTED_ENTITY*** that is the ***SELECTED_ENTITY_REFERENCE*** of a ***SELECTED_ENTITY_CHARACTERISTIC_REFERENCE CHARARACTERISTIC_BASIS***, then the lower bound for J is 0 (if true, this supersedes any previously determined lower bound for J)

— If J is a **composition** or **participant** of R, then:

- If J is in JS, then the lower bound and upper bound for J is 1
- If J is not in JS, then:
- If J is a **composition** of R, then the lower bound and upper bound for J is 1
- If J is a **participant** of R, then the lower bound is the **sourceLowerBound** of J and upper bound is the **sourceUpperBound** of J
- If L is a ***SELECTED_ENTITY*** that is the ***SELECTED_ENTITY_REFERENCE*** of a ***SELECTED_ENTITY_CHARACTERISTIC_REFERENCE CHARARACTERISTIC_BASIS***, then the lower bound for J is 0 (if true, this supersedes any previously determined lower bound for J)

Using the definitions above:

— The lower bound and the upper bound for all Js in the ***primary_structured_template_element_type_reference_parameter***'s explicitly specified or unambiguously inferred ***designated_entity_type_reference_path*** must be 1

- If ***entity_type_structured_template_element_type_reference*** matches by name the ***template_element_type_name*** of a ***union_type_decl***, then:

— The ***SELECTED_ENTITY*** referenced by the ***structured_template_element_type_reference_expression***'s ***primary_structured_template_element_type_reference_parameter***'s

designated_entity_type_reference_path's ***designated_entity_type_reference***'s ***query_selected_entity_type_reference_or_alias*** must be the same as the ***SELECTED_ENTITY*** referenced by the ***union_type_decl***'s ***union_parameter***'s ***query_selected_entity_type_reference_or_alias***

— ***inline_annotation*** must not be specified

- If ***entity_type_structured_template_element_type_reference*** matches by name an ***external_template_type_reference***, then:

  — ***inline_annotation*** must not be specified

  — Let T = the FACE.UOP.TEMPLATE whose **name** is ***external_template_type_reference***

  — Let M = ***main_entity_type_template_method_decl*** in T

  — Let Q = the **specification** of the DATAMODEL.PLATFORM.QUERY that is the **boundQuery** of T

  — Let RE = the ***SELECTED_ENTITY*** referenced by the ***primary_structured_template_element_type_reference_parameter***'s ***designated_entity_type_reference***'s ***query_selected_entity_type_reference_or_alias***

  — Let DE = the ***SELECTED_ENTITY*** in the ***FROM_CLAUSE*** of the outermost ***QUERY_STATEMENT*** of Q referenced by M's ***primary_entity_type_template_method_parameter***'s ***query_selected_entity_type_reference_or_alias***

  — RE's ***ENTITY_TYPE_REFERENCE*** must match by name DE's ***ENTITY_TYPE_REFERENCE***

- If ***equivalent_entity_type_template_method_reference*** is specified, then:

  — ***inline_annotation*** must be specified

  — Let M = an ***equivalent_entity_type_template_method_decl*** where:

    - If ***equivalent_entity_type_template_method_reference*** matches by name an ***external_template_type_reference***, then M is the ***equivalent_entity_type_template_method_decl*** of the ***main_equivalent_entity_type_template_method_decl*** of the FACE.UOP.TEMPLATE whose **name** is ***external_template_type_reference***

  — Otherwise, M is the ***equivalent_entity_type_template_method_decl*** of the ***supporting_equivalent_entity_type_template_method_decl*** whose ***template_element_type_name*** is ***equivalent_entity_type_template_method_reference***

If ***entity_type_structured_template_element_declared_parameter_reference*** is specified for the first ***structured_template_element_type_reference_parameter*** in ***structured_template_element_type_reference_parameter_list***, then:

- An ***entity_type_structured_template_element_declared_parameter_reference*** must be specified for all ***structured_template_element_type_reference_parameter***s in a ***structured_template_element_type_reference_parameter_list***

- All ***entity_type_structured_template_element_declared_parameter_reference***s must be unique
- There must be one ***structured_template_element_type_reference_parameter*** in ***structured_template_element_type_reference_parameter_list*** for each ***equivalent_entity_type_template_method_parameter*** in ***equivalent_entity_type_template_method_parameter_list*** of M
- For each ***structured_template_element_type_reference_parameter*** in ***structured_template_element_type_reference_parameter_list***:

  — The ***structured_template_element_type_reference_parameter*** 's ***entity_type_structured_template_element_declared_parameter_reference*** must match by name an ***equivalent_entity_type_template_method_parameter*** of M

  — For each ***equivalent_entity_type_template_method_member*** of M whose ***equivalent_entity_type_template_method_parameter_reference*** matches by name the ***structured_template_element_type_reference_parameter*** 's ***entity_type_structured_template_element_declared_parameter_reference***:

    - There must a **composition** in the ***SELECTED_ENTITY*** referenced by the ***structured_template_element_type_reference_parameter***'s ***designated_entity_type_reference***'s ***query_selected_entity_type_reference_or_alias*** whose **rolename** matches by name the ***equivalent_entity_type_template_method_member***'s ***equivalent_entity_type_template_method_characteristic_reference***
    - If ***idlstruct_member_reference*** is specified, then the **type** of the **composition** in the ***SELECTED_ENTITY*** referenced by the ***structured_template_element_type_reference_parameter***'s ***designated_entity_type_reference***'s ***query_selected_entity_type_reference_or_alias*** must be a DATAMODEL.PLATFORM.STRUCT, and ***idlstruct_member_reference*** must match the **rolename** of a DATAMODEL.PLATFORM.STRUCTMEMBER that is a **member** of that DATAMODEL.PLATFORM.STRUCT

Otherwise, an ***entity_type_structured_template_element_declared_parameter_reference*** must not be specified for any ***structured_template_element_type_reference_parameter***s in a ***structured_template_element_type_reference_parameter_list***, then:

- There must be one ***structured_template_element_type_reference_parameter*** in ***structured_template_element_type_reference_parameter_list*** for each ***equivalent_entity_type_template_method_parameter*** in ***equivalent_entity_type_template_method_parameter_list*** of M
- For each ***structured_template_element_type_reference_parameter*** in ***structured_template_element_type_reference_parameter_list***:

  — Let RP = the current ***structured_template_element_type_reference_parameter***

— Let EP = the ***equivalent_entity_type_template_method_parameter_reference*** whose position in the ***equivalent_entity_type_template_method_parameter_list*** of M is the same as RP in ***structured_template_element_type_reference_parameter_list***

— For each ***equivalent_entity_type_template_method_member*** of M whose ***equivalent_entity_type_template_method_parameter_reference*** matches by name EP:

- There must a **composition** in the ***SELECTED_ENTITY*** referenced by RP's ***designated_entity_type_reference***'s ***query_selected_entity_type_reference_or_alias*** whose **rolename** matches by name the ***equivalent_entity_type_template_method_member***'s ***equivalent_entity_type_template_method_characteristic_reference***
- If ***idlstruct_member_reference*** is specified, then the **type** of the **composition** in the ***SELECTED_ENTITY*** referenced by RP's ***designated_entity_type_reference***'s ***query_selected_entity_type_reference_or_alias*** must be a DATAMODEL.PLATFORM.STRUCT, and ***idlstruct_member_reference*** must match the **rolename** of a DATAMODEL.PLATFORM.STRUCTMEMBER that is a **member** of that DATAMODEL.PLATFORM.STRUCT

supporting_equivalent_entity_type_template_method_decl

// No contextually-relevant rules for instances of this term.

entity_type_template_method_body

All ***entity_type_template_method_member***s must result in a unique set of member names, where the member name(s) of a given ***entity_type_template_method_member*** is (are):

- If ***entity_type_template_method_member*** is ***explicit_designated_entity_non_entity_type_characteristic_reference_expression***, then the member name is:

  — ***structured_template_element_member_name***, if specified

  — Otherwise, ***idlstruct_member_reference***, if specified

  — Otherwise, ***designated_entity_type_reference_path*** is specified, then ***query_projected_non_entity_type_characteristic_reference***

  — Otherwise, ***query_projected_non_entity_type_characteristic_reference_or_alias***

- If ***entity_type_template_method_member*** is ***designated_entity_non_entity_type_characteristic_wildcard_reference***, then each DATAMODEL.PLATFORM.CHARACTERISTIC in the ***PROJECTED_CHARACTER_LIST*** of the outermost ***QUERY_STATEMENT*** that is a composition of the ***SELECTED_ENTITY*** referenced by the explicitly specified or unambiguously inferred ***designated_entity_type_reference_path***'s ***designated_entity_type_reference***'s ***query_selected_entity_type_reference_or_alias*** is effectively a member, and the member's name is the name of the DATAMODEL.PLATFORM.CHARACTERISTIC

- If ***entity_type_template_method_member*** is ***structured_template_element_type_reference_statement***, then:

  — If ***entity_type_structured_template_element_type_reference*** is specified, then:

    - If ***inline_annotation*** is specified, then the members name(s) is (are) determined by applying this rule to the ***entity_type_template_method_body*** of the ***supporting_entity_type_template_method_decl*** or ***main_entity_type_template_method_decl*** referenced by ***entity_type_structured_template_element_type_reference***

  — Otherwise, ***inline_annotation*** is not specified, then the member's name is ***structured_template_element_member_name***

- If ***equivalent_entity_type_template_method_reference*** is specified, then each ***equivalent_entity_type_template_element_member*** of the ***supporting_equivalent_entity_type_template_method_decl*** or ***main_equivalent_entity_type_template_method_decl*** referenced by ***equivalent_entity_type_template_method_reference*** is effectively a member, and the member's name is the ***equivalent_entity_type_template_element_member***'s:

  — ***structured_template_element_member_name***, if specified

  — Otherwise, ***idlstruct_member_reference***, if specified

  — Otherwise, ***equivalent_entity_type_template_method_characteristic_reference***

entity_type_template_method_member

// No contextually-relevant rules for instances of this term.

equivalent_entity_type_template_method_decl

An ***equivalent_entity_type_template_method_parameter_reference*** must match by name an ***equivalent_entity_type_template_method_parameter***.

equivalent_entity_type_template_method_parameter_list

All ***equivalent_entity_type_template_method_parameter***s must be unique.

equivalent_entity_type_template_method_body

All ***equivalent_entity_type_template_element_member***s must have a unique member name, where the member name of a given ***equivalent_entity_type_template_element_member*** is:

- ***structured_template_element_member_name***, if specified
- Otherwise, ***idlstruct_member_reference***, if specified
- Otherwise, ***equivalent_entity_type_template_method_characteristic_reference***

Two or more ***equivalent_entity_type_template_element_member***s must not have, as a set, the same ***equivalent_entity_type_template_method_parameter_reference***,

idlstruct_member_reference, *equivalent_entity_type_template_method_characteristic_reference*, and *optional_annotation*.

equivalent_entity_type_template_method_member

// No contextually-relevant rules for instances of this term.

equivalent_entity_type_template_element_member_statement

// No contextually-relevant rules for instances of this term.

union_type_decl

If *discriminator_type* is a *designated_entity_enumeration_type_characteristic_reference*, then:

- If *designated_entity_enumeration_type_characteristic_reference* is *query_projected_enumeration_type_characteristic_reference_or_alias*, then:

  — One of the conditions below must be satisfied, resolving it to the DATAMODEL.PLATFORM.CHARACTERISTIC identified by the first rule satisfied in priority order:

    - *query_projected_enumeration_type_characteristic_reference_or_alias* matches the **rolename** of a DATAMODEL.PLATFORM.CHARACTERISTIC that is in the *PROJECTED_CHARACTERISTIC_LIST* of the outermost *QUERY_STATEMENT* and is a **composition** of the *SELECTED_ENTITY* referenced by the *union_parameter*'s *query_selected_entity_type_reference_or_alias*
    - *query_projected_enumeration_type_characteristic_reference_or_alias* matches by name a *PROJECTED_CHARACTERISTIC_ALIAS* of an *EXPLICIT_SELECTED_ENTITY_CHARACTERISTIC_REFERENCE* in the *PROJECTED_CHARACTERISTIC_LIST* of the outermost *QUERY_STATEMENT*
    - *query_projected_enumeration_type_characteristic_reference_or_alias* matches the **rolename** of one and only one DATAMODEL.PLATFORM.CHARACTERISTIC in the *PROJECTED_CHARACTERISTIC_LIST* of the outermost *QUERY_STATEMENT*

  If the DATAMODEL.PLATFORM.CHARACTERISTIC (determined above) is not a **composition** of the *SELECTED_ENTITY* referenced by *union_parameter*'s *query_selected_entity_type_reference_or_alias*, then there must be an unambiguous *designated_entity_type_reference_path* (inferred from the *ENTITY_EXPRESSION* of the outermost *QUERY_STATEMENT*) from the *SELECTED_ENTITY* referenced by the *union_parameter*'s *query_selected_entity_type_reference_or_alias* to the *SELECTED_ENTITY* composing that DATAMODEL.PLATFORM.CHARACTERISTIC.

- Otherwise, *designated_entity_enumeration_type_characteristic_reference* is not *query_projected_enumeration_type_characteristic_reference_or_alias*, then:

— If the first ***query_selected_entity_type_reference_or_alias*** in ***designated_entity_type_reference_path*** does not match by name the ***union_parameter***'s ***query_selected_entity_type_reference_or_alias*** or, if specified, its ***entity_type_structured_template_element_declared_parameter_alias***, then there must be an unambiguous ***designated_entity_type_reference_path*** (inferred from the ***ENTITY_EXPRESSION*** of the outermost ***QUERY_STATEMENT***) from the ***SELECTED_ENTITY*** referenced by the ***union_parameter***'s ***query_selected_entity_type_reference_or_alias*** to the ***SELECTED_ENTITY*** referenced by the first ***query_selected_entity_type_reference_or_alias*** in the ***discriminator_type***'s ***designated_entity_type_reference_path***

Given the following definitions:

- Let JS = the cumulative set of all *"Join_Characteristic"*s (as defined in the rules for term designated_entity_type_reference_path) in the canonical ***designated_entity_type_reference_path*** from the ***query_selected_entity_type_reference_or_alias*** of the ***main_entity_type_template_method_decl***'s ***primary_entity_type_template_method_parameter*** to the ***query_selected_entity_type_reference_or_alias*** of this ***union_type_decl***'s ***entity_type_structured_template_element_declared_parameter_expression*** (constructed by combining head-to-tail the ***designated_entity_type_reference_path*** associated with each ***entity_type_structured_template_element_type_reference*** in sequence)

For each adjacent pair of ***query_selected_entity_type_reference_or_alias***es in the ***discriminator_type***'s explicitly specified or unambiguously inferred ***designated_entity_type_reference_path***:

- Let L = the ***SELECTED_ENTITY*** referenced by the left ***query_selected_entity_type_reference_or_alias*** of the pair
- Let R = the ***SELECTED_ENTITY*** referenced by the right (i.e., immediately following) ***query_selected_entity_type_reference_or_alias*** of the pair
- Let J = the *"Join_Characteristic"* for L and R

  If J is a **composition** or **participant** of L, then:

  — If J is in JS, then both the lower bound and upper bound for J is 1

  — If J is not in JS, then the lower bound is the **lowerBound** of J and upper bound is the **upperBound** of J

  — If R is a ***SELECTED_ENTITY*** that is the ***SELECTED_ENTITY_REFERENCE*** of a ***SELECTED_ENTITY_CHARACTERISTIC_REFERENCE*** ***CHARARACTERISTIC_BASIS***, then the lower bound for J is 0 (if true, this supersedes any previously determined lower bound for J)

  If J is a **composition** or **participant** of R, then:

  — If J is in JS, then the lower bound and upper bound for J is 1

  — If J is not in JS, then:

        - If J is a **composition** of R, then the lower bound and upper bound for J is 1
        - If J is a **participant** of R, then the lower bound is the **sourceLowerBound** of J and upper bound is the **sourceUpperBound** of J

    — If L is a ***SELECTED_ENTITY*** that is the ***SELECTED_ENTITY_REFERENCE*** of a ***SELECTED_ENTITY_CHARACTERISTIC_REFERENCE*** ***CHARARACTERISTIC_BASIS***, then the lower bound for J is 0 (if true, this supersedes any previously determined lower bound for J)

    Using the definitions above:

    — The lower bound and the upper bound for all Js in the ***discriminator_type***'s explicitly specified or unambiguously inferred ***designated_entity_type_reference_path*** must be 1, and both the **lowerBound** and **upperBound** of the DATAMODEL.PLATFORM.CHARACTERISTIC referenced by ***designated_entity_enumeration_type_characteristic_reference*** must also be 1

- Each ***case_label_literal*** must be an ***enum_literal_reference_expression*** whose:

    — ***enumeration_type_reference*** must match the **name** of the DATAMODEL.LOGICAL.ENUMERATED in the DATAMODEL.LOGICAL.MEASUREMENT realized by the **type** of the DATAMODEL.PLATFORM.COMPOSITION referenced by ***designated_entity_enumeration_type_characteristic_reference***, and

    — ***enumeration_literal_reference*** must match the **name** of a DATAMODEL.LOGICAL.ENUMERATIONLABEL that is a **label** of the above DATAMODEL.LOGICAL.ENUMERATED; if a DATAMODEL.LOGICAL.ENUMERATIONCONSTRAINT is specified for the above DATAMODEL.LOGICAL.MEASUREMENT, then that DATAMODEL.LOGICAL.ENUMERATIONLABEL must also be an **allowedValue** of that DATAMODEL.LOGICAL.ENUMERATIONCONSTRAINT

If ***entity_type_structured_template_element_member*** is ***explicit_designated_entity_non_entity_type_characteristic_reference_expression***:

- If ***designated_entity_non_entity_type_characteristic_reference*** is ***query_projected_non_entity_type_characteristic_reference_or_alias***, then:

    — One of the conditions below must be satisfied, resolving it to the DATAMODEL.PLATFORM.CHARACTERISTIC identified by the first rule satisfied in priority order:

        - ***query_projected_non_entity_type_characteristic_reference_or_alias*** matches the **rolename** of a DATAMODEL.PLATFORM.CHARACTERISTIC that is in the ***PROJECTED_CHARACTERISTIC_LIST*** of the outermost ***QUERY_STATEMENT*** and is a **composition** of the ***SELECTED_ENTITY*** referenced by the ***union_parameter***'s ***query_selected_entity_type_reference_or_alias***
        - ***query_projected_non_entity_type_characteristic_reference_or_alias*** matches by name a ***PROJECTED_CHARACTERISTIC_ALIAS*** of an ***EXPLICIT_SELECTED_ENTITY_CHARACTERISTIC_REFERENCE*** in the

PROJECTED_CHARACTERISTIC_LIST of the outermost ***QUERY_STATEMENT***

- ***query_projected_non_entity_type_characteristic_reference_or_alias*** matches the **rolename** of one and only one DATAMODEL.PLATFORM.CHARACTERISTIC in the ***PROJECTED_CHARACTERISTIC_LIST*** of the outermost ***QUERY_STATEMENT***

If the DATAMODEL.PLATFORM.CHARACTERISTIC (determined above) is not a **composition** of the ***SELECTED_ENTITY*** referenced by ***union_parameter***'s ***query_selected_entity_type_reference_or_alias***, then there must be an unambiguous ***designated_entity_type_reference_path*** (inferred from the ***ENTITY_EXPRESSION*** of the outermost ***QUERY_STATEMENT***) from the ***SELECTED_ENTITY*** referenced by the ***union_parameter***'s ***query_selected_entity_type_reference_or_alias*** to the ***SELECTED_ENTITY*** composing that DATAMODEL.PLATFORM.CHARACTERISTIC.

- Otherwise, ***designated_entity_non_entity_type_characteristic_reference*** is not ***query_projected_non_entity_type_characteristic_reference_or_alias***, then:

  — If the first ***query_selected_entity_type_reference_or_alias*** in ***designated_entity_type_reference_path*** does not match by name the ***union_parameter***'s ***query_selected_entity_type_reference_or_alias*** or, if specified, its ***entity_type_structured_template_element_declared_parameter_alias***, then there must be an unambiguous ***designated_entity_type_reference_path*** (inferred from the ***ENTITY_EXPRESSION*** of the outermost ***QUERY_STATEMENT***) from the ***SELECTED_ENTITY*** referenced by the ***union_parameter***'s ***query_selected_entity_type_reference_or_alias*** to the ***SELECTED_ENTITY*** referenced by the first ***query_selected_entity_type_reference_or_alias*** in the ***explicit_designated_entity_non_entity_type_characteristic_reference_expression***'s ***designated_entity_type_reference_path***

- If ***idlstruct_member_reference*** is specified, then the **type** of the DATAMODEL.PLATFORM.CHARACTERISTIC referenced by ***explicit_designated_entity_non_entity_type_characteristic_reference_expression*** must be a DATAMODEL.PLATFORM.STRUCT, and ***idlstruct_member_reference*** must match the **rolename** of a DATAMODEL.PLATFORM.STRUCTMEMBER that is a **member** of that DATAMODEL.PLATFORM.STRUCT

If ***entity_type_structured_template_element_member*** is ***structured_template_element_type_reference_statement***, then:

- If ***entity_type_structured_template_element_type_reference*** is specified, then:

  — An entity_type_structured_template_element_declared_parameter_reference must not be specified

  — An optional_structured_template_element_type_reference_parameter must not be specified

- If ***entity_type_structured_template_element_type_reference*** matches by name the ***template_element_type_name*** of a ***supporting_entity_type_template_method_decl***, then:

— The ***SELECTED_ENTITY*** referenced by the ***structured_template_element_type_reference_expression***'s ***primary_structured_template_element_type_reference_parameter***'s ***designated_entity_type_reference_path***'s ***designated_entity_type_reference***'s ***query_selected_entity_type_reference_or_alias*** must be the same as the ***SELECTED_ENTITY*** referenced by the ***supporting_template_method_decl***'s ***union_parameter***'s ***query_selected_entity_type_reference_or_alias***

- If ***entity_type_structured_template_element_type_reference*** matches by name the ***template_element_type_name*** of a ***union_type_decl***, then:

  — The ***SELECTED_ENTITY*** referenced by the ***structured_template_element_type_reference_expression***'s ***primary_structured_template_element_type_reference_parameter***'s ***designated_entity_type_reference_path***'s ***designated_entity_type_reference***'s ***query_selected_entity_type_reference_or_alias*** must be the same as the ***SELECTED_ENTITY*** referenced by the ***union_type_decl***'s ***union_parameter***'s ***query_selected_entity_type_reference_or_alias***

- If ***entity_type_structured_template_element_type_reference*** matches by name an ***external_template_type_reference***, then:

  — Let T = the FACE.UOP.TEMPLATE whose **name** is ***external_template_type_reference***

  — Let M = ***main_entity_type_template_method_decl*** in T

  — Let Q = the **specification** of the DATAMODEL.PLATFORM.QUERY that is the **boundQuery** of T

  — Let RE = the ***SELECTED_ENTITY*** referenced by the ***primary_structured_template_element_type_reference_parameter***'s ***designated_entity_type_reference***'s ***query_selected_entity_type_reference_or_alias***

  — Let DE = the ***SELECTED_ENTITY*** in the ***FROM_CLAUSE*** of the outermost ***QUERY_STATEMENT*** of Q referenced by M's ***union_parameter***'s ***query_selected_entity_type_reference_or_alias***

  — RE's ***ENTITY_TYPE_REFERENCE*** must match by name DE's ***ENTITY_TYPE_REFERENCE***

- An equivalent_entity_type_template_method_reference must not be specified

union_parameter

// No contextually-relevant rules for instances of this term.

union_body

// No contextually-relevant rules for instances of this term.

union_switch_statement

If ***discriminator_type*** is ***idlunsigned_short***, then each ***case_label_literal*** must be an ***idldiscriminator_type_literal*** whose value is an integer in the range 0 … 2^{16}-1.

If *discriminator_type* is *idlunsigned_long*, then each *case_label_literal* must be an *idldiscriminator_type_literal* value is an integer in the range 0 … 2^{32}-1.

If *discriminator_type* is *idlunsigned_long_long*, then each *case_label_literal* must be an *idldiscriminator_type_literal* value is an integer in the range 0 … 2^{64}-1.

discriminator_type

// No contextually-relevant rules for instances of this term.

union_switch_body

The *kw_default case_label* may be specified at most once.

There must be at least *case_label* with a *case_label_literal*.

All *case_label_literals* must be unique.

case_expression

// No contextually-relevant rules for instances of this term.

case_label

// No contextually-relevant rules for instances of this term.

case_label_literal

// No contextually-relevant rules for instances of this term.

union_member

A *designated_entity_non_entity_type_characteristic_wildcard_reference* must not be specified.

A *structured_template_element_type_reference_statement* must not specify an *inline_annotation*.

entity_type_structured_template_element_member

// No contextually-relevant rules for instances of this term.

entity_type_structured_template_element_member_statement

// No contextually-relevant rules for instances of this term.

optional_annotation

// No contextually-relevant rules for instances of this term.

inline_annotation

// No contextually-relevant rules for instances of this term.

designated_entity_characteristic_reference_statement

// No contextually-relevant rules for instances of this term.

explicit_designated_entity_non_entity_type_characteristic_reference_expression

// No contextually-relevant rules for instances of this term.

structured_template_element_type_reference_statement

// No contextually-relevant rules for instances of this term.

structured_template_element_type_reference_expression

// No contextually-relevant rules for instances of this term.

structured_template_element_type_reference_parameter_list

// No contextually-relevant rules for instances of this term.

primary_structured_template_element_type_reference_parameter

// No contextually-relevant rules for instances of this term.

optional_structured_template_element_type_reference_parameter

// No contextually-relevant rules for instances of this term.

structured_template_element_type_reference_parameter

// No contextually-relevant rules for instances of this term.

external_template_type_reference

An ***external_template_type_reference*** must match the **name** of a FACE.UOP.TEMPLATE.

entity_type_structured_template_element_type_reference

// No contextually-relevant rules for instances of this term.

entity_type_structured_template_element_declared_parameter_reference

// No contextually-relevant rules for instances of this term.

entity_type_structured_template_element_declared_parameter_expression

A ***query_selected_entity_type_reference_or_alias*** must match by name either the ***SELECTED_ENTITY_ALIAS*** or the ***ENTITY_TYPE_REFERENCE*** of one and only one ***SELECTED_ENTITY*** in the ***FROM_CLAUSE*** of the outermost ***QUERY_STATEMENT***.

entity_type_structured_template_element_declared_parameter_alias

An ***entity_type_structured_template_element_declared_parameter_alias*** must not be the same as the ***SELECTED_ENTITY_ALIAS*** or ***ENTITY_TYPE_REFERENCE*** of any ***SELECTED_ENTITY*** in the ***FROM_CLAUSE*** of the outermost ***QUERY_STATEMENT***.

An ***entity_type_structured_template_element_declared_parameter_alias*** must not be the same as any ***PROJECTED_CHARACTERISTIC_ALIAS*** in the ***PROJECTED_CHARACTERISTIC_LIST*** of the outermost ***QUERY_STATEMENT***.

An ***entity_type_structured_template_element_declared_parameter_alias*** must not be a reserved word as defined by OCL constraint face::Element::isReservedWord specified in Section J.6.1.

structured_template_element_member_name

A ***structured_template_element_member_name*** must not be a reserved word as defined by OCL constraint face::Element::isReservedWord specified in Section J.6.1.

template_element_type_name

A ***template_element_type_name*** must not be a reserved word as defined by OCL constraint face::Element::isReservedWord specified in Section J.6.1.

A ***template_element_type_name*** must not be *"main"*.

equivalent_entity_type_template_method_reference

// No contextually-relevant rules for instances of this term.

equivalent_entity_type_template_method_parameter

// No contextually-relevant rules for instances of this term.

designated_equivalent_entity_non_entity_type_characteristic_reference

// No contextually-relevant rules for instances of this term.

equivalent_entity_type_template_method_parameter_reference

// No contextually-relevant rules for instances of this term.

equivalent_entity_type_template_method_characteristic_reference

// No contextually-relevant rules for instances of this term.

designated_entity_non_entity_type_characteristic_reference

If *designated_entity_type_reference_path* is specified, then *query_projected_non_entity_type_characteristic_reference* must match the **rolename** of a DATAMODEL.PLATFORM.CHARACTERISTIC that is a **composition** of the *SELECTED_ENTITY* referenced by the *designated_entity_type_reference*'s *query_selected_entity_type_reference_or_alias*.

designated_entity_non_entity_type_characteristic_wildcard_reference

// No contextually-relevant rules for instances of this term.

designated_entity_enumeration_type_characteristic_reference

If *designated_entity_type_reference_path* is specified, then *query_projected_enumeration_type_characteristic_reference* must match the **rolename** of a DATAMODEL.PLATFORM.CHARACTERISTIC that is a **composition** of the *SELECTED_ENTITY* referenced by the *designated_entity_type_reference*'s *query_selected_entity_type_reference_or_alias*.

designated_entity_type_reference_path

Two or more *query_selected_entity_type_reference_or_alias*es must not match either the *SELECTED_ENTITY_ALIAS* or *ENTITY_TYPE_REFERENCE* of the same *SELECTED_ENTITY* in the *FROM_CLAUSE* of the outermost *QUERY_STATEMENT*.

If an *explicit_entity_type_reference_join_path* is specified, then:

- Let N = the number of *query_selected_entity_type_reference_or_alias*es
- If N > 1, then for each *query_selected_entity_type_reference_or_alias* [n] where n < N, there must an *ENTITY_TYPE_CHARACTERISTIC_EQUIVALENCE_EXPRESSION* in the *FROM_CLAUSE* of the outermost *QUERY_STATEMENT* whose *SELECTED_ENTITY_CHARACTERISTIC_REFERENCE* is a reference to a DATAMODEL.PLATFORM.CHARACTERISTIC that specifically joins *query_selected_entity_type_reference_or_alias* [n] and *query_selected_entity_type_reference_or_alias* [n + 1], where the DATAMODEL.PLATFORM.CHARACTERISTIC is:

  — A **composition** or **participant** of the *SELECTED_ENTITY* referenced by *query_selected_entity_type_reference_or_alias* [n], and whose **type** is the *SELECTED_ENTITY* referenced by the immediately following *query_selected_entity_type_reference_or_alias* [n + 1] in *explicit_entity_type_reference_join_path*, or

  — A **composition** or **participant** of the *SELECTED_ENTITY* referenced by the immediately following *query_selected_entity_type_reference_or_alias* [n + 1] in *explicit_entity_type_reference_join_path*, and whose **type** is the *SELECTED_ENTITY* referenced by that *query_selected_entity_type_reference_or_alias* [n]

- For the last or only *query_selected_entity_type_reference_or_alias* (i.e., n = N) , there must be an *ENTITY_TYPE_CHARACTERISTIC_EQUIVALENCE_EXPRESSION* in the *FROM_CLAUSE* of the outermost *QUERY_STATEMENT* whose

SELECTED_ENTITY_CHARACTERISTIC_REFERENCE is a reference to a DATAMODEL.PLATFORM.CHARACTERISTIC that specifically joins ***query_selected_entity_type_reference_or_alias*** [n] and the ***designated_entity_type_reference***'s ***query_selected_entity_type_reference_or_alias***, where the DATAMODEL.PLATFORM.CHARACTERISTIC is a **composition** or **participant** of:

— The ***SELECTED_ENTITY*** referenced by ***query_selected_entity_type_reference_or_alias*** [n] and whose **type** is the ***SELECTED_ENTITY*** referenced by the ***designated_entity_type_reference***'s ***query_selected_entity_type_reference_or_alias***, or

— The ***SELECTED_ENTITY*** referenced by the ***designated_entity_type_reference***'s ***query_selected_entity_type_reference_or_alias*** and whose **type** is the ***SELECTED_ENTITY*** referenced by that ***query_selected_entity_type_reference_or_alias*** [n]

Note: In both cases above, the DATAMODEL.PLATFORM.CHARACTERISTIC that joins a given adjacent pair of ***query_selected_entity_type_reference_or_alias***es – that is, a ***query_selected_entity_type_reference_or_alias*** and its immediately following ***query_selected_entity_type_reference_or_alias*** (e.g., [n] and [n + 1]) in ***designated_entity_type_reference_path*** – is referred to as a *"Join_Characteristic"*.

explicit_entity_type_reference_join_path

// No contextually-relevant rules for instances of this term.

join_path_entity_type_reference

// No contextually-relevant rules for instances of this term.

designated_entity_type_reference

// No contextually-relevant rules for instances of this term.

qualified_entity_type_reference

// No contextually-relevant rules for instances of this term.

entity_type_reference

// No contextually-relevant rules for instances of this term.

entity_characteristic_value_qualifier

// No contextually-relevant rules for instances of this term.

idlstruct_member_reference

// No contextually-relevant rules for instances of this term.

enum_literal_reference_expression

// No contextually-relevant rules for instances of this term.

enumeration_type_reference

// No contextually-relevant rules for instances of this term.

enumeration_literal_reference

// No contextually-relevant rules for instances of this term.

query_where_clause_criteria

// No contextually-relevant rules for instances of this term.

query_projected_non_entity_type_characteristic_reference_or_alias

A ***query_projected_non_entity_type_characteristic_reference_or_alias*** must be a DATAMODEL.PLATFORM.CHARACTERISTIC in the ***PROJECTED_CHARACTERISTIC_LIST*** of the outermost ***QUERY_STATEMENT***.

A ***query_projected_non_entity_type_characteristic_reference_or_alias*** must be a DATAMODEL.PLATFORM.COMPOSITION whose **type** is not an ENTITY.

query_projected_non_entity_type_characteristic_reference

A ***query_projected_non_entity_type_characteristic_reference*** must be a DATAMODEL.PLATFORM.CHARACTERISTIC in the ***PROJECTED_CHARACTERISTIC_LIST*** of the outermost ***QUERY_STATEMENT***.

A ***query_projected_non_entity_type_characteristic_reference*** must be a DATAMODEL.PLATFORM.COMPOSITION whose **type** is not an ENTITY.

query_projected_enumeration_type_characteristic_reference_or_alias

A ***query_projected_enumeration_type_characteristic_reference_or_alias*** must be a DATAMODEL.PLATFORM.CHARACTERISTIC in the ***PROJECTED_CHARACTERISTIC_LIST*** of the outermost ***QUERY_STATEMENT***.

A ***query_projected_enumeration_type_characteristic_reference_or_alias*** must be a DATAMODEL.PLATFORM.COMPOSITION whose **type** is a DATAMODEL.LOGICAL.ENUMERATION that realizes a DATAMODEL.LOGICAL.MEASUREMENT whose **measurementSystem** is the DATAMODEL.LOGICAL.MEASUREMENTSYSTEM whose **name** is "*AbstractDiscreteSetMeasurementSystem*".

query_projected_enumeration_type_characteristic_reference

A ***query_projected_enumeration_type_characteristic_reference*** must be a DATAMODEL.PLATFORM.CHARACTERISTIC in the ***PROJECTED_CHARACTERISTIC_LIST*** of the outermost ***QUERY_STATEMENT***.

A ***query_projected_enumeration_type_characteristic_reference*** must be a DATAMODEL.PLATFORM.COMPOSITION whose **type** is a DATAMODEL.LOGICAL.ENUMERATION that realizes a DATAMODEL.LOGICAL.MEASUREMENT whose **measurementSystem** is the DATAMODEL.LOGICAL.MEASUREMENTSYSTEM whose **name** is "*AbstractDiscreteSetMeasurementSystem*".

query_selected_entity_type_reference_or_alias

A ***query_selected_entity_type_reference_or_alias*** must match by name either the ***SELECTED_ENTITY_ALIAS*** or the ***ENTITY_TYPE_REFERENCE*** of one and only one ***SELECTED_ENTITY*** in the ***FROM_CLAUSE*** of the outermost ***QUERY_STATEMENT***.

A ***query_selected_entity_type_reference_or_alias*** must not match the ***ENTITY_TYPE_REFERENCE*** of any ***SELECTED_ENTITY*** in the ***FROM_CLAUSE*** of the outermost ***QUERY_STATEMENT*** whose ***ENTITY_TYPE_REFERENCE*** is not unique to one ***SELECTED_ENTITY***.

idldiscriminator_type

// No contextually-relevant rules for instances of this term.

idlunsigned_int

// No contextually-relevant rules for instances of this term.

idlunsigned_short

// No contextually-relevant rules for instances of this term.

idlunsigned_long

// No contextually-relevant rules for instances of this term.

idlunsigned_long_long

// No contextually-relevant rules for instances of this term.

idlboolean

// No contextually-relevant rules for instances of this term.

idldiscriminator_type_literal

// No contextually-relevant rules for instances of this term.

idlinteger_literal

// No contextually-relevant rules for instances of this term.

idloctal_literal

// No contextually-relevant rules for instances of this term.

idlhex_literal

// No contextually-relevant rules for instances of this term.

idlboolean_literal

// No contextually-relevant rules for instances of this term.

IDENTIFIER

An ***IDENTIFIER*** is a valid String as defined by OCL constraint face::Element::isValidIdentifier specified in Section J.6.1.

J.5 EMOF Metamodel

```
<emof:Package xmlns:xmi="http://www.omg.org/XMI"
xmlns:emof="http://schema.omg.org/spec/MOF/2.0/emof.xml" xmi:version="2.0"
xmi:id="face" name="face" uri="http://www.opengroup.us/face/3.1">
  <ownedType xmi:type="emof:Class" xmi:id="face.ArchitectureModel"
name="ArchitectureModel" superClass="face.Element">
    <ownedAttribute xmi:id="face.ArchitectureModel.dm" name="dm" lower="0"
upper="*" isComposite="true">
      <type xmi:type="emof:Class" href="datamodel.emof#//DataModel"/>
    </ownedAttribute>
    <ownedAttribute xmi:id="face.ArchitectureModel.um" name="um" lower="0"
upper="*" type="face.uop.UoPModel" isComposite="true"/>
    <ownedAttribute xmi:id="face.ArchitectureModel.im" name="im" lower="0"
upper="*" type="face.integration.IntegrationModel" isComposite="true"/>
    <ownedAttribute xmi:id="face.ArchitectureModel.tm" name="tm" lower="0"
upper="*" type="face.traceability.TraceabilityModel" isComposite="true"/>
  </ownedType>
  <ownedType xmi:type="emof:Class" xmi:id="face.Element" name="Element"
isAbstract="true">
    <ownedAttribute xmi:id="face.Element.name" name="name" isOrdered="true"
default="">
      <type xmi:type="emof:PrimitiveType"
href="http://schema.omg.org/spec/MOF/2.0/emof.xml#String"/>
    </ownedAttribute>
    <ownedAttribute xmi:id="face.Element.description" name="description"
isOrdered="true" default="">
      <type xmi:type="emof:PrimitiveType"
href="http://schema.omg.org/spec/MOF/2.0/emof.xml#String"/>
    </ownedAttribute>
  </ownedType>
  <nestedPackage xmi:id="face.uop" name="uop"
uri="http://www.opengroup.us/face/uop/3.1">
    <ownedType xmi:type="emof:Class" xmi:id="face.uop.UoPModel" name="UoPModel"
superClass="face.Element">
      <ownedAttribute xmi:id="face.uop.UoPModel.element" name="element"
lower="0" upper="*" type="face.uop.Element" isComposite="true"/>
      <ownedAttribute xmi:id="face.uop.UoPModel.um" name="um" lower="0"
upper="*" type="face.uop.UoPModel" isComposite="true"/>
    </ownedType>
```

```
    <ownedType xmi:type="emof:Class" xmi:id="face.uop.Element" name="Element"
isAbstract="true" superClass="face.Element"/>
    <ownedType xmi:type="emof:Class" xmi:id="face.uop.SupportingComponent"
name="SupportingComponent" isAbstract="true" superClass="face.uop.Element">
      <ownedAttribute xmi:id="face.uop.SupportingComponent.version"
name="version" isOrdered="true">
        <type xmi:type="emof:PrimitiveType"
href="http://schema.omg.org/spec/MOF/2.0/emof.xml#String"/>
      </ownedAttribute>
    </ownedType>
    <ownedType xmi:type="emof:Class" xmi:id="face.uop.LanguageRunTime"
name="LanguageRunTime" superClass="face.uop.SupportingComponent"/>
    <ownedType xmi:type="emof:Class" xmi:id="face.uop.ComponentFramework"
name="ComponentFramework" superClass="face.uop.SupportingComponent"/>
    <ownedType xmi:type="emof:Enumeration" xmi:id="face.uop.ClientServerRole"
name="ClientServerRole">
      <ownedLiteral xmi:id="face.uop.ClientServerRole.Client" name="Client"/>
      <ownedLiteral xmi:id="face.uop.ClientServerRole.Server" name="Server"/>
    </ownedType>
    <ownedType xmi:type="emof:Enumeration" xmi:id="face.uop.FaceProfile"
name="FaceProfile">
      <ownedLiteral xmi:id="face.uop.FaceProfile.GeneralPurpose"
name="GeneralPurpose"/>
      <ownedLiteral xmi:id="face.uop.FaceProfile.Security" name="Security"/>
      <ownedLiteral xmi:id="face.uop.FaceProfile.SafetyBase"
name="SafetyBase"/>
      <ownedLiteral xmi:id="face.uop.FaceProfile.SafetyExtended"
name="SafetyExtended"/>
    </ownedType>
    <ownedType xmi:type="emof:Enumeration"
xmi:id="face.uop.DesignAssuranceLevel" name="DesignAssuranceLevel">
      <ownedLiteral xmi:id="face.uop.DesignAssuranceLevel.A" name="A"/>
      <ownedLiteral xmi:id="face.uop.DesignAssuranceLevel.B" name="B"/>
      <ownedLiteral xmi:id="face.uop.DesignAssuranceLevel.C" name="C"/>
      <ownedLiteral xmi:id="face.uop.DesignAssuranceLevel.D" name="D"/>
      <ownedLiteral xmi:id="face.uop.DesignAssuranceLevel.E" name="E"/>
    </ownedType>
    <ownedType xmi:type="emof:Enumeration"
xmi:id="face.uop.DesignAssuranceStandard" name="DesignAssuranceStandard">
      <ownedLiteral xmi:id="face.uop.DesignAssuranceStandard.DO_178B_ED_12B"
name="DO_178B_ED_12B" literal="DO_178B_ED_12B"/>
      <ownedLiteral xmi:id="face.uop.DesignAssuranceStandard.DO_178C_ED_12C"
name="DO_178C_ED_12C" literal="DO_178C_ED_12C"/>
    </ownedType>
    <ownedType xmi:type="emof:Enumeration"
xmi:id="face.uop.MessageExchangeType" name="MessageExchangeType">
      <ownedLiteral xmi:id="face.uop.MessageExchangeType.InboundMessage"
name="InboundMessage"/>
      <ownedLiteral xmi:id="face.uop.MessageExchangeType.OutboundMessage"
name="OutboundMessage"/>
    </ownedType>
    <ownedType xmi:type="emof:Enumeration" xmi:id="face.uop.PartitionType"
name="PartitionType">
      <ownedLiteral xmi:id="face.uop.PartitionType.POSIX" name="POSIX"/>
      <ownedLiteral xmi:id="face.uop.PartitionType.ARINC653" name="ARINC653"/>
    </ownedType>
    <ownedType xmi:type="emof:Enumeration"
xmi:id="face.uop.ProgrammingLanguage" name="ProgrammingLanguage">
      <ownedLiteral xmi:id="face.uop.ProgrammingLanguage.C" name="C"/>
      <ownedLiteral xmi:id="face.uop.ProgrammingLanguage.CPP" name="CPP"/>
      <ownedLiteral xmi:id="face.uop.ProgrammingLanguage.Java" name="Java"/>
      <ownedLiteral xmi:id="face.uop.ProgrammingLanguage.Ada" name="Ada"/>
    </ownedType>
```

```
    <ownedType xmi:type="emof:Enumeration"
xmi:id="face.uop.SynchronizationStyle" name="SynchronizationStyle">
      <ownedLiteral xmi:id="face.uop.SynchronizationStyle.Blocking"
name="Blocking"/>
      <ownedLiteral xmi:id="face.uop.SynchronizationStyle.NonBlocking"
name="NonBlocking"/>
    </ownedType>
    <ownedType xmi:type="emof:Enumeration" xmi:id="face.uop.ThreadType"
name="ThreadType">
      <ownedLiteral xmi:id="face.uop.ThreadType.Foreground" name="Foreground"/>
      <ownedLiteral xmi:id="face.uop.ThreadType.Background" name="Background"/>
    </ownedType>
    <ownedType xmi:type="emof:Class" xmi:id="face.uop.AbstractUoP"
name="AbstractUoP" superClass="face.uop.Element
face.traceability.TraceableElement">
      <ownedAttribute xmi:id="face.uop.AbstractUoP.connection"
name="connection" lower="0" upper="*" type="face.uop.AbstractConnection"
isComposite="true"/>
    </ownedType>
    <ownedType xmi:type="emof:Class" xmi:id="face.uop.AbstractConnection"
name="AbstractConnection" superClass="face.Element
face.traceability.TraceableElement">
      <ownedAttribute xmi:id="face.uop.AbstractConnection.conceptualView"
name="conceptualView" isOrdered="true" lower="0">
        <type xmi:type="emof:Class" href="datamodel.emof#//conceptual/View"/>
      </ownedAttribute>
      <ownedAttribute xmi:id="face.uop.AbstractConnection.logicalView"
name="logicalView" isOrdered="true" lower="0">
        <type xmi:type="emof:Class" href="datamodel.emof#//logical/View"/>
      </ownedAttribute>
    </ownedType>
    <ownedType xmi:type="emof:Class" xmi:id="face.uop.UnitOfPortability"
name="UnitOfPortability" isAbstract="true" superClass="face.uop.Element
face.traceability.TraceableElement">
      <ownedAttribute xmi:id="face.uop.UnitOfPortability.supportingComponent"
name="supportingComponent" lower="0" upper="*"
type="face.uop.SupportingComponent"/>
      <ownedAttribute xmi:id="face.uop.UnitOfPortability.thread" name="thread"
upper="*" type="face.uop.Thread" isComposite="true"/>
      <ownedAttribute xmi:id="face.uop.UnitOfPortability.memoryRequirements"
name="memoryRequirements" isOrdered="true"
type="face.uop.RAMMemoryRequirements" isComposite="true"/>
      <ownedAttribute xmi:id="face.uop.UnitOfPortability.realizes"
name="realizes" isOrdered="true" lower="0" type="face.uop.AbstractUoP"/>
      <ownedAttribute xmi:id="face.uop.UnitOfPortability.connection"
name="connection" upper="*" type="face.uop.Connection" isComposite="true"/>
      <ownedAttribute xmi:id="face.uop.UnitOfPortability.transportAPILanguage"
name="transportAPILanguage" isOrdered="true"
type="face.uop.ProgrammingLanguage"/>
      <ownedAttribute xmi:id="face.uop.UnitOfPortability.designAssuranceLevel"
name="designAssuranceLevel" isOrdered="true" lower="0"
type="face.uop.DesignAssuranceLevel"/>
      <ownedAttribute xmi:id="face.uop.UnitOfPortability.partitionType"
name="partitionType" isOrdered="true" type="face.uop.PartitionType"/>
      <ownedAttribute
xmi:id="face.uop.UnitOfPortability.designAssuranceStandard"
name="designAssuranceStandard" isOrdered="true" lower="0"
type="face.uop.DesignAssuranceStandard"/>
      <ownedAttribute xmi:id="face.uop.UnitOfPortability.faceProfile"
name="faceProfile" isOrdered="true" type="face.uop.FaceProfile"/>
      <ownedAttribute xmi:id="face.uop.UnitOfPortability.lcmPort"
name="lcmPort" lower="0" upper="2" type="face.uop.LifeCycleManagementPort"
isComposite="true"/>
```

```
    </ownedType>
    <ownedType xmi:type="emof:Class" xmi:id="face.uop.PortableComponent"
name="PortableComponent" superClass="face.uop.UnitOfPortability"/>
    <ownedType xmi:type="emof:Class"
xmi:id="face.uop.PlatformSpecificComponent" name="PlatformSpecificComponent"
superClass="face.uop.UnitOfPortability"/>
    <ownedType xmi:type="emof:Class" xmi:id="face.uop.Thread" name="Thread">
      <ownedAttribute xmi:id="face.uop.Thread.period" name="period"
isOrdered="true">
        <type xmi:type="emof:PrimitiveType"
href="http://www.eclipse.org/emf/2002/Ecore.emof#ecore.EFloat"/>
      </ownedAttribute>
      <ownedAttribute xmi:id="face.uop.Thread.timeCapacity" name="timeCapacity"
isOrdered="true">
        <type xmi:type="emof:PrimitiveType"
href="http://www.eclipse.org/emf/2002/Ecore.emof#ecore.EFloat"/>
      </ownedAttribute>
      <ownedAttribute xmi:id="face.uop.Thread.relativePriority"
name="relativePriority" isOrdered="true">
        <type xmi:type="emof:PrimitiveType"
href="http://schema.omg.org/spec/MOF/2.0/emof.xml#Integer"/>
      </ownedAttribute>
      <ownedAttribute xmi:id="face.uop.Thread.relativeCoreAffinity"
name="relativeCoreAffinity" isOrdered="true">
        <type xmi:type="emof:PrimitiveType"
href="http://schema.omg.org/spec/MOF/2.0/emof.xml#Integer"/>
      </ownedAttribute>
      <ownedAttribute xmi:id="face.uop.Thread.threadType" name="threadType"
isOrdered="true" type="face.uop.ThreadType"/>
    </ownedType>
    <ownedType xmi:type="emof:Class" xmi:id="face.uop.RAMMemoryRequirements"
name="RAMMemoryRequirements">
      <ownedAttribute xmi:id="face.uop.RAMMemoryRequirements.heapStackMin"
name="heapStackMin" isOrdered="true" lower="0">
        <type xmi:type="emof:PrimitiveType"
href="http://schema.omg.org/spec/MOF/2.0/emof.xml#Integer"/>
      </ownedAttribute>
      <ownedAttribute xmi:id="face.uop.RAMMemoryRequirements.heapStackMax"
name="heapStackMax" isOrdered="true" lower="0">
        <type xmi:type="emof:PrimitiveType"
href="http://schema.omg.org/spec/MOF/2.0/emof.xml#Integer"/>
      </ownedAttribute>
      <ownedAttribute xmi:id="face.uop.RAMMemoryRequirements.heapStackTypical"
name="heapStackTypical" isOrdered="true" lower="0">
        <type xmi:type="emof:PrimitiveType"
href="http://schema.omg.org/spec/MOF/2.0/emof.xml#Integer"/>
      </ownedAttribute>
      <ownedAttribute xmi:id="face.uop.RAMMemoryRequirements.textMax"
name="textMax" isOrdered="true" lower="0">
        <type xmi:type="emof:PrimitiveType"
href="http://schema.omg.org/spec/MOF/2.0/emof.xml#Integer"/>
      </ownedAttribute>
      <ownedAttribute xmi:id="face.uop.RAMMemoryRequirements.roDataMax"
name="roDataMax" isOrdered="true" lower="0">
        <type xmi:type="emof:PrimitiveType"
href="http://schema.omg.org/spec/MOF/2.0/emof.xml#Integer"/>
      </ownedAttribute>
      <ownedAttribute xmi:id="face.uop.RAMMemoryRequirements.dataMax"
name="dataMax" isOrdered="true" lower="0">
        <type xmi:type="emof:PrimitiveType"
href="http://schema.omg.org/spec/MOF/2.0/emof.xml#Integer"/>
      </ownedAttribute>
```

```
      <ownedAttribute xmi:id="face.uop.RAMMemoryRequirements.bssMax"
name="bssMax" isOrdered="true" lower="0">
        <type xmi:type="emof:PrimitiveType"
href="http://schema.omg.org/spec/MOF/2.0/emof.xml#Integer"/>
      </ownedAttribute>
    </ownedType>
    <ownedType xmi:type="emof:Class" xmi:id="face.uop.Connection"
name="Connection" isAbstract="true"
superClass="face.traceability.TraceableElement face.Element">
      <ownedAttribute xmi:id="face.uop.Connection.realizes" name="realizes"
isOrdered="true" lower="0" type="face.uop.AbstractConnection"/>
      <ownedAttribute xmi:id="face.uop.Connection.period" name="period"
isOrdered="true">
        <type xmi:type="emof:PrimitiveType"
href="http://www.eclipse.org/emf/2002/Ecore.emof#ecore.EFloat"/>
      </ownedAttribute>
      <ownedAttribute xmi:id="face.uop.Connection.synchronizationStyle"
name="synchronizationStyle" isOrdered="true"
type="face.uop.SynchronizationStyle"/>
    </ownedType>
    <ownedType xmi:type="emof:Class" xmi:id="face.uop.ClientServerConnection"
name="ClientServerConnection" superClass="face.uop.Connection">
      <ownedAttribute xmi:id="face.uop.ClientServerConnection.requestType"
name="requestType" isOrdered="true" type="face.uop.MessageType"/>
      <ownedAttribute xmi:id="face.uop.ClientServerConnection.responseType"
name="responseType" isOrdered="true" type="face.uop.MessageType"/>
      <ownedAttribute xmi:id="face.uop.ClientServerConnection.role" name="role"
isOrdered="true" type="face.uop.ClientServerRole"/>
    </ownedType>
    <ownedType xmi:type="emof:Class" xmi:id="face.uop.PubSubConnection"
name="PubSubConnection" isAbstract="true" superClass="face.uop.Connection">
      <ownedAttribute xmi:id="face.uop.PubSubConnection.messageType"
name="messageType" isOrdered="true" type="face.uop.MessageType"/>
      <ownedAttribute xmi:id="face.uop.PubSubConnection.messageExchangeType"
name="messageExchangeType" isOrdered="true"
type="face.uop.MessageExchangeType"/>
    </ownedType>
    <ownedType xmi:type="emof:Class" xmi:id="face.uop.QueuingConnection"
name="QueuingConnection" superClass="face.uop.PubSubConnection">
      <ownedAttribute xmi:id="face.uop.QueuingConnection.depth" name="depth"
isOrdered="true">
        <type xmi:type="emof:PrimitiveType"
href="http://schema.omg.org/spec/MOF/2.0/emof.xml#Integer"/>
      </ownedAttribute>
    </ownedType>
    <ownedType xmi:type="emof:Class"
xmi:id="face.uop.SingleInstanceMessageConnection"
name="SingleInstanceMessageConnection" superClass="face.uop.PubSubConnection"/>
    <ownedType xmi:type="emof:Class" xmi:id="face.uop.LifeCycleManagementPort"
name="LifeCycleManagementPort">
      <ownedAttribute
xmi:id="face.uop.LifeCycleManagementPort.messageExchangeType"
name="messageExchangeType" isOrdered="true"
type="face.uop.MessageExchangeType"/>
      <ownedAttribute xmi:id="face.uop.LifeCycleManagementPort.lcmMessageType"
name="lcmMessageType" isOrdered="true" type="face.uop.MessageType"/>
    </ownedType>
    <ownedType xmi:type="emof:Class" xmi:id="face.uop.MessageType"
name="MessageType" isAbstract="true" superClass="face.uop.Element
face.traceability.TraceableElement"/>
    <ownedType xmi:type="emof:Class" xmi:id="face.uop.CompositeTemplate"
name="CompositeTemplate" superClass="face.uop.Element face.uop.MessageType">
```

```
      <ownedAttribute xmi:id="face.uop.CompositeTemplate.composition"
name="composition" isOrdered="true" lower="2" upper="*"
type="face.uop.TemplateComposition" isComposite="true"/>
      <ownedAttribute xmi:id="face.uop.CompositeTemplate.realizes"
name="realizes" isOrdered="true" lower="0">
        <type xmi:type="emof:Class"
href="datamodel.emof#//logical/CompositeQuery"/>
      </ownedAttribute>
      <ownedAttribute xmi:id="face.uop.CompositeTemplate.isUnion"
name="isUnion" isOrdered="true">
        <type xmi:type="emof:PrimitiveType"
href="http://schema.omg.org/spec/MOF/2.0/emof.xml#Boolean"/>
      </ownedAttribute>
    </ownedType>
    <ownedType xmi:type="emof:Class" xmi:id="face.uop.TemplateComposition"
name="TemplateComposition">
      <ownedAttribute xmi:id="face.uop.TemplateComposition.rolename"
name="rolename" isOrdered="true" default="">
        <type xmi:type="emof:PrimitiveType"
href="http://schema.omg.org/spec/MOF/2.0/emof.xml#String"/>
      </ownedAttribute>
      <ownedAttribute xmi:id="face.uop.TemplateComposition.realizes"
name="realizes" isOrdered="true" lower="0">
        <type xmi:type="emof:Class"
href="datamodel.emof#//logical/QueryComposition"/>
      </ownedAttribute>
      <ownedAttribute xmi:id="face.uop.TemplateComposition.type" name="type"
isOrdered="true" type="face.uop.MessageType"/>
    </ownedType>
    <ownedType xmi:type="emof:Class" xmi:id="face.uop.Template" name="Template"
superClass="face.uop.MessageType">
      <ownedAttribute xmi:id="face.uop.Template.specification"
name="specification" isOrdered="true">
        <type xmi:type="emof:PrimitiveType"
href="http://schema.omg.org/spec/MOF/2.0/emof.xml#String"/>
      </ownedAttribute>
      <ownedAttribute xmi:id="face.uop.Template.boundQuery" name="boundQuery"
isOrdered="true" lower="0">
        <type xmi:type="emof:Class" href="datamodel.emof#//platform/Query"/>
      </ownedAttribute>
      <ownedAttribute xmi:id="face.uop.Template.effectiveQuery"
name="effectiveQuery" isOrdered="true" lower="0">
        <type xmi:type="emof:Class" href="datamodel.emof#//platform/Query"/>
      </ownedAttribute>
    </ownedType>
  </nestedPackage>
  <nestedPackage xmi:id="face.integration" name="integration"
uri="http://www.opengroup.us/face/integration/3.1">
    <ownedType xmi:type="emof:Class" xmi:id="face.integration.IntegrationModel"
name="IntegrationModel" superClass="face.Element">
      <ownedAttribute xmi:id="face.integration.IntegrationModel.im" name="im"
lower="0" upper="*" type="face.integration.IntegrationModel"
isComposite="true"/>
      <ownedAttribute xmi:id="face.integration.IntegrationModel.element"
name="element" lower="0" upper="*" type="face.integration.Element"
isComposite="true"/>
    </ownedType>
    <ownedType xmi:type="emof:Class" xmi:id="face.integration.Element"
name="Element" isAbstract="true" superClass="face.Element"/>
    <ownedType xmi:type="emof:Class"
xmi:id="face.integration.IntegrationContext" name="IntegrationContext"
superClass="face.integration.Element">
```

```
      <ownedAttribute xmi:id="face.integration.IntegrationContext.connection"
name="connection" lower="0" upper="*" type="face.integration.TSNodeConnection"
isComposite="true"/>
      <ownedAttribute xmi:id="face.integration.IntegrationContext.node"
name="node" lower="0" upper="*" type="face.integration.TransportNode"
isComposite="true"/>
    </ownedType>
    <ownedType xmi:type="emof:Class" xmi:id="face.integration.TSNodeConnection"
name="TSNodeConnection">
      <ownedAttribute xmi:id="face.integration.TSNodeConnection.source"
name="source" isOrdered="true" type="face.integration.TSNodePortBase"/>
      <ownedAttribute xmi:id="face.integration.TSNodeConnection.destination"
name="destination" isOrdered="true" type="face.integration.TSNodePortBase"/>
    </ownedType>
    <ownedType xmi:type="emof:Class" xmi:id="face.integration.TSNodePortBase"
name="TSNodePortBase" isAbstract="true"/>
    <ownedType xmi:type="emof:Class" xmi:id="face.integration.UoPInstance"
name="UoPInstance" superClass="face.integration.Element">
      <ownedAttribute xmi:id="face.integration.UoPInstance.realizes"
name="realizes" isOrdered="true" type="face.uop.UnitOfPortability"/>
      <ownedAttribute xmi:id="face.integration.UoPInstance.output"
name="output" lower="0" upper="*" type="face.integration.UoPOutputEndPoint"
isComposite="true"/>
      <ownedAttribute xmi:id="face.integration.UoPInstance.input" name="input"
lower="0" upper="*" type="face.integration.UoPInputEndPoint"
isComposite="true"/>
      <ownedAttribute xmi:id="face.integration.UoPInstance.configurationURI"
name="configurationURI" isOrdered="true" lower="0">
        <type xmi:type="emof:PrimitiveType"
href="http://schema.omg.org/spec/MOF/2.0/emof.xml#String"/>
      </ownedAttribute>
    </ownedType>
    <ownedType xmi:type="emof:Class" xmi:id="face.integration.UoPEndPoint"
name="UoPEndPoint" isAbstract="true"
superClass="face.integration.TSNodePortBase">
      <ownedAttribute xmi:id="face.integration.UoPEndPoint.connection"
name="connection" isOrdered="true" type="face.uop.Connection"/>
    </ownedType>
    <ownedType xmi:type="emof:Class" xmi:id="face.integration.UoPInputEndPoint"
name="UoPInputEndPoint" superClass="face.integration.UoPEndPoint"/>
    <ownedType xmi:type="emof:Class"
xmi:id="face.integration.UoPOutputEndPoint" name="UoPOutputEndPoint"
superClass="face.integration.UoPEndPoint"/>
    <ownedType xmi:type="emof:Class" xmi:id="face.integration.TransportNode"
name="TransportNode" isAbstract="true" superClass="face.Element">
      <ownedAttribute xmi:id="face.integration.TransportNode.outPort"
name="outPort" isOrdered="true" lower="0"
type="face.integration.TSNodeOutputPort" isComposite="true"/>
      <ownedAttribute xmi:id="face.integration.TransportNode.inPort"
name="inPort" lower="0" upper="*" type="face.integration.TSNodeInputPort"
isComposite="true"/>
    </ownedType>
    <ownedType xmi:type="emof:Class" xmi:id="face.integration.TSNodePort"
name="TSNodePort" isAbstract="true"
superClass="face.integration.TSNodePortBase">
      <ownedAttribute xmi:id="face.integration.TSNodePort.view" name="view"
isOrdered="true" type="face.uop.MessageType"/>
    </ownedType>
    <ownedType xmi:type="emof:Class" xmi:id="face.integration.TSNodeInputPort"
name="TSNodeInputPort" superClass="face.integration.TSNodePort"/>
    <ownedType xmi:type="emof:Class" xmi:id="face.integration.TSNodeOutputPort"
name="TSNodeOutputPort" superClass="face.integration.TSNodePort"/>
```

```
    <ownedType xmi:type="emof:Class" xmi:id="face.integration.ViewAggregation"
name="ViewAggregation" superClass="face.integration.TransportNode"/>
    <ownedType xmi:type="emof:Class" xmi:id="face.integration.ViewFilter"
name="ViewFilter" superClass="face.integration.TransportNode"/>
    <ownedType xmi:type="emof:Class" xmi:id="face.integration.ViewSource"
name="ViewSource" superClass="face.integration.TransportNode"/>
    <ownedType xmi:type="emof:Class" xmi:id="face.integration.ViewSink"
name="ViewSink" superClass="face.integration.TransportNode"/>
    <ownedType xmi:type="emof:Class"
xmi:id="face.integration.ViewTransformation" name="ViewTransformation"
superClass="face.integration.TransportNode"/>
    <ownedType xmi:type="emof:Class" xmi:id="face.integration.ViewTransporter"
name="ViewTransporter" superClass="face.integration.TransportNode">
      <ownedAttribute xmi:id="face.integration.ViewTransporter.channel"
name="channel" isOrdered="true" type="face.integration.TransportChannel"/>
    </ownedType>
    <ownedType xmi:type="emof:Class" xmi:id="face.integration.TransportChannel"
name="TransportChannel" superClass="face.integration.Element"/>
  </nestedPackage>
  <nestedPackage xmi:id="face.traceability" name="traceability"
uri="http://www.opengroup.us/face/traceability/3.1">
    <ownedType xmi:type="emof:Class"
xmi:id="face.traceability.TraceabilityModel" name="TraceabilityModel"
superClass="face.Element">
      <ownedAttribute xmi:id="face.traceability.TraceabilityModel.element"
name="element" lower="0" upper="*" type="face.traceability.Element"
isComposite="true"/>
      <ownedAttribute xmi:id="face.traceability.TraceabilityModel.tm" name="tm"
lower="0" upper="*" type="face.traceability.TraceabilityModel"
isComposite="true"/>
    </ownedType>
    <ownedType xmi:type="emof:Class" xmi:id="face.traceability.Element"
name="Element" isAbstract="true" superClass="face.Element"/>
    <ownedType xmi:type="emof:Class"
xmi:id="face.traceability.TraceableElement" name="TraceableElement"
isAbstract="true">
      <ownedAttribute
xmi:id="face.traceability.TraceableElement.traceabilityPoint"
name="traceabilityPoint" lower="0" upper="*"
type="face.traceability.TraceabilityPoint" isComposite="true"/>
    </ownedType>
    <ownedType xmi:type="emof:Class"
xmi:id="face.traceability.TraceabilityPoint" name="TraceabilityPoint">
      <ownedAttribute xmi:id="face.traceability.TraceabilityPoint.rationale"
name="rationale" isOrdered="true" lower="0">
        <type xmi:type="emof:PrimitiveType"
href="http://schema.omg.org/spec/MOF/2.0/emof.xml#String"/>
      </ownedAttribute>
      <ownedAttribute xmi:id="face.traceability.TraceabilityPoint.reference"
name="reference" isOrdered="true" lower="0">
        <type xmi:type="emof:PrimitiveType"
href="http://schema.omg.org/spec/MOF/2.0/emof.xml#String"/>
      </ownedAttribute>
    </ownedType>
    <ownedType xmi:type="emof:Class"
xmi:id="face.traceability.UoPTraceabilitySet" name="UoPTraceabilitySet"
superClass="face.traceability.Element face.traceability.TraceableElement">
      <ownedAttribute xmi:id="face.traceability.UoPTraceabilitySet.uop"
name="uop" lower="0" upper="*" type="face.uop.UnitOfPortability"/>
      <ownedAttribute xmi:id="face.traceability.UoPTraceabilitySet.abstractUoP"
name="abstractUoP" lower="0" upper="*" type="face.uop.AbstractUoP"/>
    </ownedType>
```

```
    <ownedType xmi:type="emof:Class"
xmi:id="face.traceability.ConnectionTraceabilitySet"
name="ConnectionTraceabilitySet" superClass="face.traceability.Element
face.traceability.TraceableElement">
      <ownedAttribute
xmi:id="face.traceability.ConnectionTraceabilitySet.connection"
name="connection" lower="0" upper="*" type="face.uop.Connection"/>
      <ownedAttribute
xmi:id="face.traceability.ConnectionTraceabilitySet.abstractConnection"
name="abstractConnection" lower="0" upper="*"
type="face.uop.AbstractConnection"/>
    </ownedType>
    <ownedType xmi:type="emof:Class"
xmi:id="face.traceability.ConceptualEntityTrace" name="ConceptualEntityTrace"
superClass="face.traceability.Element face.traceability.TraceableElement">
      <ownedAttribute xmi:id="face.traceability.ConceptualEntityTrace.entity"
name="entity" isOrdered="true">
        <type xmi:type="emof:Class" href="datamodel.emof#//conceptual/Entity"/>
      </ownedAttribute>
    </ownedType>
    <ownedType xmi:type="emof:Class"
xmi:id="face.traceability.ConceptualViewTrace" name="ConceptualViewTrace"
superClass="face.traceability.Element face.traceability.TraceableElement">
      <ownedAttribute xmi:id="face.traceability.ConceptualViewTrace.view"
name="view" isOrdered="true">
        <type xmi:type="emof:Class" href="datamodel.emof#//conceptual/View"/>
      </ownedAttribute>
    </ownedType>
    <ownedType xmi:type="emof:Class"
xmi:id="face.traceability.LogicalEntityTrace" name="LogicalEntityTrace"
superClass="face.traceability.Element face.traceability.TraceableElement">
      <ownedAttribute xmi:id="face.traceability.LogicalEntityTrace.entity"
name="entity" isOrdered="true">
        <type xmi:type="emof:Class" href="datamodel.emof#//logical/Entity"/>
      </ownedAttribute>
    </ownedType>
    <ownedType xmi:type="emof:Class"
xmi:id="face.traceability.LogicalViewTrace" name="LogicalViewTrace"
superClass="face.traceability.Element face.traceability.TraceableElement">
      <ownedAttribute xmi:id="face.traceability.LogicalViewTrace.view"
name="view" isOrdered="true">
        <type xmi:type="emof:Class" href="datamodel.emof#//logical/View"/>
      </ownedAttribute>
    </ownedType>
    <ownedType xmi:type="emof:Class"
xmi:id="face.traceability.PlatformEntityTrace" name="PlatformEntityTrace"
superClass="face.traceability.Element face.traceability.TraceableElement">
      <ownedAttribute xmi:id="face.traceability.PlatformEntityTrace.entity"
name="entity" isOrdered="true">
        <type xmi:type="emof:Class" href="datamodel.emof#//platform/Entity"/>
      </ownedAttribute>
    </ownedType>
    <ownedType xmi:type="emof:Class"
xmi:id="face.traceability.PlatformViewTrace" name="PlatformViewTrace"
superClass="face.traceability.Element face.traceability.TraceableElement">
      <ownedAttribute xmi:id="face.traceability.PlatformViewTrace.view"
name="view" isOrdered="true">
        <type xmi:type="emof:Class" href="datamodel.emof#//platform/View"/>
      </ownedAttribute>
    </ownedType>
  </nestedPackage>
</emof:Package>
```

J.6 Object Constraint Language Constraints

The OCL constraints governing USM and DSDM content are detailed in this section. These constraints are using the OMG Object Constraint Language (OCL), Version 2.4.

J.6.1 FACE Data Model Language OCL Constraints

```
package face

  context Element
    /*
     * Helper method that determines if a string is a valid identifier.
     * An identifier is valid if it consists of alphanumeric characters.
     */
    static def: isValidIdentifier(str : String) : Boolean =
      str.size() > 0 and
      str.replaceAll('[a-zA-Z][_a-zA-Z0-9]*', '').size() = 0

    /*
     * Helper method that determines if a string is an IDL 4.1 keyword.
     */
    static def: isReservedWord(str : String) : Boolean =
      let strLower : String = str.toLowerCase() in
        (strLower = 'abstract') or
        (strLower = 'alias') or
        (strLower = 'any') or
        (strLower = 'attribute') or
        (strLower = 'bitfield') or
        (strLower = 'bitmask') or
        (strLower = 'bitset') or
        (strLower = 'boolean') or
        (strLower = 'case') or
        (strLower = 'char') or
        (strLower = 'component') or
        (strLower = 'connector') or
        (strLower = 'const') or
        (strLower = 'consumes') or
        (strLower = 'context') or
        (strLower = 'custom') or
        (strLower = 'default') or
        (strLower = 'double') or
        (strLower = 'emits') or
        (strLower = 'enum') or
        (strLower = 'eventtype') or
        (strLower = 'exception') or
        (strLower = 'factory') or
        (strLower = 'false') or
        (strLower = 'finder') or
        (strLower = 'fixed') or
        (strLower = 'float') or
        (strLower = 'getraises') or
        (strLower = 'home') or
        (strLower = 'import') or
        (strLower = 'in') or
        (strLower = 'inout') or
        (strLower = 'interface') or
        (strLower = 'local') or
        (strLower = 'long') or
        (strLower = 'manages') or
        (strLower = 'map') or
        (strLower = 'mirrorport') or
        (strLower = 'module') or
```

```
            (strLower = 'multiple') or
            (strLower = 'native') or
            (strLower = 'object') or
            (strLower = 'octet') or
            (strLower = 'oneway') or
            (strLower = 'out') or
            (strLower = 'port') or
            (strLower = 'porttype') or
            (strLower = 'primarykey') or
            (strLower = 'private') or
            (strLower = 'provides') or
            (strLower = 'public') or
            (strLower = 'publishes') or
            (strLower = 'raises') or
            (strLower = 'readonly') or
            (strLower = 'sequence') or
            (strLower = 'setraises') or
            (strLower = 'short') or
            (strLower = 'string') or
            (strLower = 'struct') or
            (strLower = 'supports') or
            (strLower = 'switch') or
            (strLower = 'true') or
            (strLower = 'truncatable') or
            (strLower = 'typedef') or
            (strLower = 'typeid') or
            (strLower = 'typename') or
            (strLower = 'typeprefix') or
            (strLower = 'union') or
            (strLower = 'unsigned') or
            (strLower = 'uses') or
            (strLower = 'valuebase') or
            (strLower = 'valuetype') or
            (strLower = 'void') or
            (strLower = 'wchar') or
            (strLower = 'wstring')

        /*
         * The name of an Element is a valid identifier.
         */
        inv nameIsValidIdentifier:
          Element::isValidIdentifier(self.name)

      context ArchitectureModel
        /*
         * Every Element in an ArchitectureModel has a unique name.
         */
        inv hasUniqueName:
          let children : Bag(String)=
            self.dm->collect(name.toLowerCase())->union(
            self.um->collect(name.toLowerCase())->union(
            self.im->collect(name.toLowerCase())->union(
            self.tm->collect(name.toLowerCase()))))) in

          children->size() = children->asSet()->size()

endpackage

package face::uop

  context Element
    /*
     * All UoP Elements have a unique name.
```

```
   */
  inv hasUniqueName:
    not Element.allInstances()->excluding(self)
                              ->collect(name)
                              ->includes(self.name)

context Connection
  /*
   * Helper method that gets the Views associated with a Connection.
   */
  def: getViews() : Set(MessageType) =
    if self.oclIsKindOf(PubSubConnection) then
       self.oclAsType(PubSubConnection).messageType.oclAsSet()
    else -- self.oclIsTypeOf(ClientServerConnection)
      self.oclAsType(ClientServerConnection).requestType
        ->including(self.oclAsType(ClientServerConnection).responseType)
    endif

  /*
   * If a Connection realizes an AbstractConnection,
   * its requestType or responseType or both (for ClientServerConnections) or
   * its messageType (for PubSubConnections) realizes either the
   * AbstractConnection's logicalView or a logical View that realizes the
   * AbstractConnection's conceptualView.
   */
  inv realizationTypeConsistent:
    self.realizes <> null implies

    self.getViews()->exists(view |

    if view.oclIsTypeOf(CompositeTemplate) then
      let cTemplate
        = view.oclAsType(CompositeTemplate) in

      if self.realizes.logicalView <> null then
        cTemplate.realizes <> null and
        cTemplate.realizes = self.realizes.logicalView
      else -- self.realizes.conceptualView <> null
        cTemplate.realizes <> null and
        cTemplate.realizes.realizes <> null and
        cTemplate.realizes.realizes = self.realizes.conceptualView
      endif

    else -- view.oclIsTypeOf(Template)
      let lbTemplate = view.oclAsType(Template) in

      lbTemplate.effectiveQuery <> null implies

      if self.realizes.logicalView <> null then
        lbTemplate.effectiveQuery.realizes <> null and
        lbTemplate.effectiveQuery.realizes = self.realizes.logicalView
      else -- self.realizes.conceptualView <> null
        lbTemplate.effectiveQuery.realizes <> null and
        lbTemplate.effectiveQuery.realizes.realizes <> null and
        lbTemplate.effectiveQuery.realizes.realizes
          = self.realizes.conceptualView
      endif
    endif
    )

context QueuingConnection
  /*
   * A QueuingConnection's queue depth is greater than zero.
```

```
   */
  inv depthValid:
    self.depth > 0

context AbstractUoP
  /*
   * An AbstractUoP is entirely logical or entirely conceptual.
   * (Its AbstractConnections all have their logicalView set and
   * conceptualView not set or all have their conceptualView set and
   * logicalView not set.)
   */
  inv onlyLogicalOrOnlyConceptual:
    self.connection->collect(logicalView)->forAll(lv | lv <> null) xor
    self.connection->collect(conceptualView)->forAll(cv | cv <> null)

 context UnitOfPortability
  /*
   * If a UoP "A" realizes an AbstractUoP "B", then A and B
   * have the same number of connections, and every Connection in A
   * realizes a unique AbstractConnection in B.
   * If a UoP does not realize an AbstractUoP, none of its Connections
   * realize.
   */
  inv connectionsConsistentWithUoPRealization:
    if self.realizes <> null then
      self.connection.realizes = self.realizes.connection->asBag()
    else
      self.connection.realizes->forAll(ac | ac = null)
    endif

context MessageType
  /*
   * A Platform Element's name is not an IDL reserved word.
   */
  inv nameIsNotReservedWord:
    not Element::isReservedWord(self.name)

context CompositeTemplate
  /*
   * A TemplateComposition's rolename is unique within a CompositeTemplate.
   */
  inv compositionsHaveUniqueRolenames:
    self.composition->collect(rolename)
                    ->isUnique(rn | rn)

  /*
   * A CompositeTemplate does not compose itself.
   */
  inv noCyclesInConstruction:
    let composedTemplates = self.composition
                                ->collect(type)
                                ->selectByKind(CompositeTemplate)
                                ->closure(composition
                                        ->collect(type)
                                        ->selectByKind(CompositeTemplate)) in

    not composedTemplates->includes(self)

  /*
   * A CompositeTemplate does not compose the same Template more than once.
   */
  inv viewComposedOnce:
    self.composition->collect(type)->isUnique(view | view)
```

```
  /*
   * TemplateCompositions in a platform CompositeTemplate realize
   * QueryCompositions in the logical CompositeQuery that the platform
   * CompositeTemplate realizes.
   */
  inv compositionsConsistentWithRealization:
    if self.realizes = null
    then
      self.composition->forAll(c | c.realizes = null)
    else
      self.composition->forAll(c |
          self.realizes.composition->exists(c2 | c.realizes = c2)
      )
    endif

  /*
   * A CompositeTemplate that realizes has the same "isUnion" property
   * as the CompositeQuery it realizes.
   */
  inv realizationUnionConsistent:
    self.realizes->forAll(realized | self.isUnion = realized.isUnion)

  /*
   * A CompositeTemplate does not contain two TemplateCompositions
   * that realize the same QueryComposition.
   */
  inv realizedCompositionsHaveDifferentTypes:
    self.realizes <> null implies
    self.composition->forAll(c1, c2 | c1 <> c2 implies
                                      c1.realizes <> c2.realizes)

context TemplateComposition
  /*
   * The rolename of a TemplateComposition is a valid identifier.
   */
  inv rolenameIsValidIdentifier:
    Element::isValidIdentifier(self.rolename)

  /*
   * The rolename of a TemplateComposition is not an IDL reserved word.
   */
  inv rolenameIsNotReservedWord:
    not Element::isReservedWord(self.rolename)

  /*
   * If TemplateComposition "A" realizes QueryComposition "B", then
   * if A's type is a CompositeTemplate, then A's type realizes B's type, and
   * if A's type is a Template and defines an effectiveQuery,
   * then A's type's effectiveQuery realizes B's type.
   */
  inv typeConsistentWithRealization:
    self.realizes <> null implies
    if self.type.oclIsTypeOf(CompositeTemplate) then
      self.type.oclAsType(CompositeTemplate).realizes
        = self.realizes.type
    else
      self.type.oclAsType(Template).effectiveQuery <> null
        implies
      self.type.oclAsType(Template).effectiveQuery.realizes
        = self.realizes.type
    endif
```

```
endpackage

package face::integration

  context Element
    /*
     * All Integration Elements have a unique name.
     */
    inv hasUniqueName:
      not Element.allInstances()->excluding(self)
                                ->collect(name.toLowerCase())
                                ->includes(self.name.toLowerCase())

  context UoPInstance
    /*
     * If a UoPInstance "A" realizes an UoP "B", then A has one unique
     * UoPEndPoint that realizes each of B's PubSubConnections, one unique
     * UoPInputEndPoint that realizes each of B's ClientServerConnections,
     * and one UoPOutputEndPoint that realizes each of B's
     * ClientServerConnections. A has no additional UoPEndPoints.
     */
    inv endPointsConsistentWithRealization:
      self.output->union(self.input)->collect(connection)
        = self.realizes.connection->asBag()
            ->union(self.realizes
                        .connection
                        ->asBag()
                        ->selectByKind(face::uop::ClientServerConnection))

        and

      self.output->collect(connection)
                  ->selectByKind(face::uop::ClientServerConnection)
        = self.input->collect(connection)
                    ->selectByKind(face::uop::ClientServerConnection)

  context TSNodePortBase
    /*
     * Helper method that gets the TransportNode containing a given
     * TSNodePortBase
     */
    def: getParentTransportNode() : TransportNode =
      TransportNode.allInstances()->select(tn | tn.inPort->includes(self) or
                                                tn.outPort->includes(self))
                                   ->any(true)

    /*
     * Helper method that gets the View used by a TSNodePortBase.
     * If a UoPInputEndPoint's connection is a ClientServerConnection, then
     * its View is the connection's responseType.
     * If a UoPOutputEndPoint's connection is a ClientServerConnection, then
     * its View is the connection's requestType.
     */
    def: getView() : uop::MessageType =
      if self.oclIsKindOf(TSNodePort) then
        self.oclAsType(TSNodePort).view
      else -- self.oclIsKindOf(UoPEndPoint)
        let uopConnection = self.oclAsType(UoPEndPoint).connection in
        if uopConnection.oclIsKindOf(face::uop::PubSubConnection) then
          uopConnection.oclAsType(face::uop::PubSubConnection).messageType
        else -- uopConnection.oclIsTypeOf(face::uop::ClientServerConnection)
          let clientServerConnection =
            uopConnection.oclAsType(face::uop::ClientServerConnection) in
```

```
        if self.oclIsTypeOf(UoPInputEndPoint) then
          clientServerConnection.responseType
        else
          clientServerConnection.requestType
        endif
      endif
    endif

  /*
   * A TSNodePortBase is connected by a TSNodeConnection.
   */
  inv isConnected:
    TSNodeConnection.allInstances()->collect(source)->union(
    TSNodeConnection.allInstances()->collect(destination))->includes(self)

context TSNodeConnection
  /*
   * A TSNodeConnection uses the same View on its source and destination.
   */
  inv sourceViewMatchesDestinationView:
    self.source.getView() = self.destination.getView()

  /*
   * A TSNodeConnection's source is an output.
   */
  inv sourceIsOutput:
    self.source.oclIsTypeOf(UoPOutputEndPoint) or
    self.source.oclIsTypeOf(TSNodeOutputPort)

  /*
   * A TSNodeConnection's destination is an input.
   */
  inv destinationIsInput:
    self.destination.oclIsTypeOf(UoPInputEndPoint) or
    self.destination.oclIsTypeOf(TSNodeInputPort)

  /*
   * A TSNodeConnection connects TransportNodes that
   * are in the same IntegrationContext as the TSNodeConnection.
   */
  inv connectWithinSameContext:
    let parentContext
      = IntegrationContext.allInstances()->any(x | x.connection
                                                  ->includes(self)) in
    let ports = parentContext.node->collect(inPort)->union(
                parentContext.node->collect(outPort)) in

    (self.source.oclIsKindOf(TSNodePort)
       implies
     ports->includes(self.source))

      and

    (self.destination.oclIsKindOf(TSNodePort)
       implies
     ports->includes(self.destination))

  /*
   * There is at least one ViewTransporter on a path
   * between any two UoPInstances.
   */
  inv transporterOnPath:
    self.destination.oclIsTypeOf(UoPInputEndPoint)
```

```
      implies
    self.source.oclIsTypeOf(TSNodeOutputPort) and
    self.source.getParentTransportNode()
              ->closure(getPreviousNodes())
              ->exists(n | n.oclIsTypeOf(ViewTransporter) or
                           n.oclIsTypeOf(ViewSource))

context TSNodeInputPort
  /*
   * A TSNodeInputPort is the destination of at most one TSNodeConnection.
   */
  inv onlyOneConnection:
      TSNodeConnection.allInstances()
          ->select(x | x.destination = self)->size() <= 1

context UoPInputEndPoint
  /*
   * A UoPInputEndPoint's connection is either a ClientServerConnection
   * or a PubSubConnection whose messageExchangeType is InboundMessage.
   */
  inv uoPEndPointConsistentWithRealization:
    self.connection.oclIsTypeOf(face::uop::ClientServerConnection)

      or

    (self.connection.oclIsKindOf(face::uop::PubSubConnection) and
     self.connection.oclAsType(face::uop::PubSubConnection)
                    .messageExchangeType
       = face::uop::MessageExchangeType::InboundMessage)

  /*
   * A UoPInputEndPoint is the destination of at most one TSNodeConnection.
   */
  inv onlyOneConnection:
      TSNodeConnection.allInstances()
          ->select(x | x.destination = self)->size() <= 1

context UoPOutputEndPoint
  /*
   * A UoPInputEndPoint's connection is either a ClientServerConnection
   * or a PubSubConnection whose messageExchangeType is OutboundMessage.
   */
  inv uoPEndPointConsistentWithRealization:
    self.connection.oclIsTypeOf(face::uop::ClientServerConnection)

      or

    (self.connection.oclIsKindOf(face::uop::PubSubConnection) and
     self.connection.oclAsType(face::uop::PubSubConnection)
                    .messageExchangeType
       = face::uop::MessageExchangeType::OutboundMessage)

context TransportNode
  /*
   * Helper method that gets the set of TransportNodes that are
   * "upstream" from a given TransportNode.
   */
  def: getPreviousNodes() : Set(TransportNode) =
    TSNodeConnection.allInstances()
                    ->select(c | self.outPort->includes(c.destination))
                    ->collect(source)
                    ->selectByKind(TSNodeOutputPort)
                    ->collect(getParentTransportNode())
```

```
                    ->asSet()

/*
 * Helper method that gets the set of TransportNodes that are
 * "downstream" from a given TransportNode.
 */
def: getNextNodes() : Set(TransportNode) =
  TSNodeConnection.allInstances()
                    ->select(c | self.outPort->includes(c.source))
                    ->collect(destination)
                    ->selectByKind(TSNodeInputPort)
                    ->collect(getParentTransportNode())
                    ->asSet()

/*
 * An IntegrationContext has no cycles.
 */
inv noCycles:
  not self.getNextNodes()->closure(getNextNodes())->includes(self)

/*
 * A ViewSource has no inputs.
 * A ViewSink, ViewFilter, ViewTransformation, or ViewTransporter
 * has one input.
 * A ViewAggregation has more than one input.
 */
inv hasCorrectInputCount:
 (self.oclIsTypeOf(ViewSource)
    implies
  self.inPort->size() = 0)

    and

 (self.oclIsTypeOf(ViewSink) or
  self.oclIsTypeOf(ViewFilter) or
  self.oclIsTypeOf(ViewTransformation) or
  self.oclIsTypeOf(ViewTransporter)
    implies
  self.inPort->size() = 1)

    and

 (self.oclIsTypeOf(ViewAggregation)
    implies
  self.inPort->size() > 1)

/*
 * A ViewSink has no outputs.
 * A ViewSource, ViewFilter, ViewAggregation, ViewTransformation,
 * or ViewTransporter has one output.
 */
inv hasCorrectOutputCount:
 (self.oclIsTypeOf(ViewSink)
    implies
  self.outPort->size() = 0)

    and

 (self.oclIsTypeOf(ViewSource) or
  self.oclIsTypeOf(ViewFilter) or
  self.oclIsTypeOf(ViewAggregation) or
  self.oclIsTypeOf(ViewTransformation) or
  self.oclIsTypeOf(ViewTransporter)
```

```
        implies
      self.outPort->size() = 1)

  context ViewSource
    /*
     * A ViewSource is connected to a UoPInputEndPoint.
     */
    inv viewSourceConnectedToUoPInputEndPoint:
      TSNodeConnection.allInstances()
                      ->select(x | self.outPort->includes(x.source))
                      ->collect(destination)
                      ->forAll(oclIsTypeOf(UoPInputEndPoint))

  context ViewSink
    /*
     * A ViewSink is connected to a UoPOutputEndPoint.
     */
    inv viewSinkConnectedToUoPOutputEndPoint:
      TSNodeConnection.allInstances()
                      ->select(x | self.inPort->includes(x.destination))
                      ->collect(source)
                      ->forAll(oclIsTypeOf(UoPOutputEndPoint))

  context ViewFilter
    /*
     * A ViewFilter uses the same View on its input and output.
     */
    inv viewIsConsistent:
      self.inPort->size() = 1 and
      self.outPort->size() = 1
        implies
      self.inPort->any(true).view = self.outPort->any(true).view

  context ViewTransporter
    /*
     * A ViewTransporter uses the same View on its input and output.
     */
    inv viewIsConsistent:
      self.inPort->size() = 1 and
      self.outPort->size() = 1
        implies
      self.inPort->any(true).view = self.outPort->any(true).view

endpackage

package face::traceability

  context Element
    /*
     * All Traceability Elements have a unique name.
     */
    inv hasUniqueName:
      not Element.allInstances()->excluding(self)
                                ->collect(name)
                                ->includes(self.name)

Endpackage
```

J.6.2 FACE Data Model Language OCL Constraints on Open UDDL Content

```
package datamodel

  context Element
```

```
/*
 * Helper method that determines if a string is an IDL 4.1 keyword.
 */
static def: isReservedWord(str : String) : Boolean =
  let strLower : String = str.toLowerCase() in
    (strLower = 'abstract') or
    (strLower = 'alias') or
    (strLower = 'any') or
    (strLower = 'attribute') or
    (strLower = 'bitfield') or
    (strLower = 'bitmask') or
    (strLower = 'bitset') or
    (strLower = 'boolean') or
    (strLower = 'case') or
    (strLower = 'char') or
    (strLower = 'component') or
    (strLower = 'connector') or
    (strLower = 'const') or
    (strLower = 'consumes') or
    (strLower = 'context') or
    (strLower = 'custom') or
    (strLower = 'default') or
    (strLower = 'double') or
    (strLower = 'emits') or
    (strLower = 'enum') or
    (strLower = 'eventtype') or
    (strLower = 'exception') or
    (strLower = 'factory') or
    (strLower = 'false') or
    (strLower = 'finder') or
    (strLower = 'fixed') or
    (strLower = 'float') or
    (strLower = 'getraises') or
    (strLower = 'home') or
    (strLower = 'import') or
    (strLower = 'in') or
    (strLower = 'inout') or
    (strLower = 'interface') or
    (strLower = 'local') or
    (strLower = 'long') or
    (strLower = 'manages') or
    (strLower = 'map') or
    (strLower = 'mirrorport') or
    (strLower = 'module') or
    (strLower = 'multiple') or
    (strLower = 'native') or
    (strLower = 'object') or
    (strLower = 'octet') or
    (strLower = 'oneway') or
    (strLower = 'out') or
    (strLower = 'port') or
    (strLower = 'porttype') or
    (strLower = 'primarykey') or
    (strLower = 'private') or
    (strLower = 'provides') or
    (strLower = 'public') or
    (strLower = 'publishes') or
    (strLower = 'raises') or
    (strLower = 'readonly') or
    (strLower = 'sequence') or
    (strLower = 'setraises') or
    (strLower = 'short') or
    (strLower = 'string') or
```

```
        (strLower = 'struct') or
        (strLower = 'supports') or
        (strLower = 'switch') or
        (strLower = 'true') or
        (strLower = 'truncatable') or
        (strLower = 'typedef') or
        (strLower = 'typeid') or
        (strLower = 'typename') or
        (strLower = 'typeprefix') or
        (strLower = 'union') or
        (strLower = 'unsigned') or
        (strLower = 'uses') or
        (strLower = 'valuebase') or
        (strLower = 'valuetype') or
        (strLower = 'void') or
        (strLower = 'wchar') or
        (strLower = 'wstring')

endpackage

package datamodel::conceptual

  context Entity
    /*
     * Helper method that gets the Characteristics contained in an Entity.
     */
    def: getLocalCharacteristics() : Set(Characteristic) =
      if self.oclIsTypeOf(Association) then
        self.composition
          ->union(self.oclAsType(Association).participant)
          ->oclAsType(Set(Characteristic))
      else
        self.composition->oclAsType(Set(Characteristic))
      endif

    /*
     * Helper method that gets the Characteristics of an Entity,
     * including those from specialized Entities.
     */
    def: getAllCharacteristics() : Set(Characteristic) =
      let allCharacteristics : Set(Characteristic) =
        self->closure(specializes)
            ->collect(getLocalCharacteristics())
            ->asSet() in

      -- get all characteristics that have been specialized
      let specializedCharacteristics : Set(Characteristic) =
        allCharacteristics->collect(specializes)
                          ->asSet() in

       -- return all characteristics that have not been specialized
       allCharacteristics - specializedCharacteristics

    /*
     * Helper method that determines whether or not
     * an Entity is part of a specialization cycle.
     */
    def: isPartOfSpecializationCycle() : Boolean =
      self.specializes->closure(specializes)->includes(self)

    /*
     * Helper method to retrieve the BasisEntities of an Entity,
     * including those from specialized Entities.
```

```
   */
  def: getBasisEntities() : Bag(BasisEntity) =
    self->closure(specializes)
        ->collect(basisEntity)

  /*
   * Helper method that gets the identity of a conceptual Entity.
   */
  def: getEntityIdentity() : Bag(OclAny) =
    self.getAllCharacteristics()
      ->collectNested(getIdentityContribution())
      ->union(self.getBasisEntities())

  /*
   * A Conceptual Entity contains a Composition whose type
   * is an Observable named 'UniqueIdentifier'.
   */
  inv hasUniqueID:
    self.getAllCharacteristics()
        ->selectByType(Composition)
        ->collect(type)
        ->exists(a | a.oclIsTypeOf(Observable)
                     and a.name = 'Identifier')

context Characteristic
  /*
   * Helper method that gets the type of a Characteristic.
   */
  def: getType() : ComposableElement =
    if self.oclIsTypeOf(Composition) then
      self.oclAsType(Composition).type
    else
      self.oclAsType(Participant).getResolvedType()
    endif

  /*
   * Helper method that gets the contribution a Characteristic makes
   * to an Entity's uniqueness.
   */
  def: getIdentityContribution() : Sequence(OclAny) =
    if self.oclIsTypeOf(Composition) then
      self.oclAsType(Composition).getIdentityContribution()
    else
      self.oclAsType(Participant).getIdentityContribution()
    endif

context Composition
  /*
   * Helper method that gets the contribution a Composition makes
   * to an Entity's uniqueness (type and multiplicity).
   */
   def: getIdentityContribution() : Sequence(OclAny) =
     Sequence{self.type,
              self.lowerBound,
              self.upperBound}

context Participant
  /*
   * Helper method that gets a Participant's PathNode sequence.
   */
  def: getPathSequence() : OrderedSet(PathNode) =
    self.path
        ->asOrderedSet()
```

```
        ->closure(pn : PathNode |
                  let projectedParticipantPath =
                    if pn.projectsParticipant() then
                       pn.projectedParticipant().path
                    else
                      null
                    endif in

                  OrderedSet{projectedParticipantPath,
                             pn.node}
                    ->reject(oclIsUndefined()))

    /*
     * Helper method that gets the contribution a Participant makes
     * to an Entity's uniqueness (type, path sequence, and multiplicity).
     */
     def: getIdentityContribution() : Sequence(OclAny) =
       Sequence{self.type,
                self.getPathSequence()->collect(getProjectedCharacteristic()),
                self.lowerBound,
                self.upperBound}

    /*
     * Helper method that determines if a Participant's
     * path sequence contains a cycle.
     */
    def: hasCycleInPath() : Boolean =
      self.getPathSequence()
          ->collect(getProjectedCharacteristic())
          ->includes(self)

    /*
     * Helper method that gets the element projected by a Participant.
     * Returns a ComposableElement.
     */
    def: getResolvedType() : ComposableElement =
      if self.hasCycleInPath() then
        null
      else if self.path = null then
        self.type
      else
        self.getPathSequence()->last().getNodeType()
      endif
      endif

  context PathNode
    /*
     * Helper method that determines if a PathNode projects a Participant.
     */
    def: projectsParticipant() : Boolean =
      self.oclIsTypeOf(CharacteristicPathNode) and
      self.oclAsType(CharacteristicPathNode)
          .projectedCharacteristic
          .oclIsTypeOf(Participant)

    /*
     * Helper method that gets the Participant projected by a PathNode.
     * Returns null if no Participant is projected.
     */
    def: projectedParticipant() : Participant =
      let pp = self.oclAsType(CharacteristicPathNode)
                   .projectedCharacteristic
                   .oclAsType(Participant) in
```

```
      if not pp.oclIsInvalid() then
        pp
      else
        null
      endif

    /*
     * Helper method that gets the Characteristic projected by a PathNode.
     */
    def: getProjectedCharacteristic() : Characteristic =
      if self.oclIsTypeOf(CharacteristicPathNode) then
        self.oclAsType(CharacteristicPathNode).projectedCharacteristic
      else -- ParticipantPathNode
        self.oclAsType(ParticipantPathNode).projectedParticipant
      endif

    /*
     * Helper method that gets the "node type" of a PathNode. For a
     * CharacteristicPathNode, the node type is the type of the projected
     * characteristic. For a ParticipantPathNode, the node type is the
     * Association containing the projected Participant.
     * Returns a ComposableElement.
     */
    def: getNodeType() : ComposableElement =
      if self.oclIsTypeOf(CharacteristicPathNode) then
        self.oclAsType(CharacteristicPathNode)
            .projectedCharacteristic
            .getType()
      else
        -- get Association that contains projectedCharacteristic
        Association.allInstances()
          ->select(participant->includes(self.oclAsType(ParticipantPathNode)
                                              .projectedParticipant))
          ->any(true)
      endif

endpackage

package datamodel::logical

  context Enumerated
    /*
     * An Enumerated's name is not an IDL reserved word.
     */
    inv nameIsNotReservedWord:
      not Element::isReservedWord(self.name)

    /*
     * An EnumerationLabel's name is unique within an Enumerated.
     */
    inv enumerationLabelNameUnique:
      self.label->isUnique(name)

  context EnumerationLabel
    /*
     * An EnumerationLabel's name is not an IDL reserved word.
     */
    inv nameIsNotReservedWord:
      not Element::isReservedWord(self.name)

endpackage

package datamodel::platform
```

```
  context Element
    /*
     * A Platform Element's name is not an IDL reserved word.
     */
    inv nameIsNotReservedWord:
      not Element::isReservedWord(self.name)

  context Characteristic
    /*
     * Helper method that gets the rolename of a Characteristic.
     */
    def: getRolename() : String =
      if self.oclIsKindOf(Composition) then
        self.oclAsType(Composition).rolename
      else
        self.oclAsType(Participant).getRolename()
      endif

    /*
     * The rolename of a Characteristic is not an IDL reserved word.
     */
    inv rolenameIsNotReservedWord:
      self.getRolename() <> null implies
      (not Element::isReservedWord(self.getRolename()))

endpackage
```

J.7 Conditional OCL Constraints

The conditional OCL constraints governing USM and DSDM content are detailed in this section. These constraints are using the OMG Object Constraint Language (OCL), Version 2.4. These constraints are intended to be enforced when required.

J.7.1 Single Observable Constraint

```
package datamodel::conceptual

  context Entity
    /*
     * An Entity does not compose the same Observable more than once.
     */
    inv observableComposedOnce:
      self.getAllCharacteristics()
          ->selectByKind(Composition)
          ->collect(getType())
          ->select(oclIsTypeOf(Observable))
          ->isUnique(obs | obs)

Endpackage
```

J.7.2 Entity Uniqueness Constraint

```
package datamodel::conceptual

  context Entity
    /*
     * An Entity is unique in a Conceptual Data Model.
     * (An Entity is unique if the set of its Characteristics
     * is different from other Entities' in terms of
```

```
     * type, lowerBound, upperBound, and path (for Participants).
     *
     * NOTE: If an Entity is part of a specialization cycle, its uniqueness
     * is undefined. So, if an Entity is part of a specialization cycle,
     * it will not fail entityIsUnique, but will fail noCyclesInSpecialization.
     */
    inv entityIsUnique:
      not self.isPartOfSpecializationCycle() implies
      not Entity.allInstances()
                ->excluding(self)
                ->collectNested(getEntityIdentity())
                ->includes(self.getEntityIdentity())

endpackage
```

J.8 FACE Data Model Language IDL Bindings

LEGEND:

Rule: The following identifier is the name of a rule. In most cases, the identifier is the name of a production rule in the Template grammar. A Rule will either Emit IDL or Return data (to another Rule), such as values or references to elements.

Expression: The following symbols are the preceding rule's expression. An expression defines the possible constructions accepted by the rule. In some cases, the expression is a logical or platform meta-type in the UDDL meta-model in the form of datamodel.logical.NAME or datamodel.platform.NAME, or a uop meta-type in the FACE meta-model in the form of face.uop.NAME, where NAME is the name of a meta-type in the qualified namespace of the corresponding meta-model. In other (most) cases, the expression is the right-hand-side of a production rule in the Template grammar. In these cases, the Rule’s name may be the name of a production rule in the Template grammar, and the expression is the right-hand-side of that production rule.

Alternate: Some expressions are comprised of two or more alternate sub-expressions. In this case, the Expression is empty and the alternate sub-expressions are subsequently listed as Alternates.

Variant: Some expressions or sub-expressions are comprised of optional symbols or symbol groups. In this case, the various combinations of the expression's symbols with and without the optional symbols are subsequently detailed as Variants, where each Variant represents one possible construction.

Emit: Specifies the Rule or Rules to be followed and/or the IDL to be generated for a construction based on the preceding Expression, Alternate, or Variant during any pass of applying this binding to a given Template specification or platform View.

Emit1: A special case of Emit that specifies the IDL to be generated during the first pass of applying this binding to a given Template specification.

Emit2: A special case of Emit that specifies the IDL to be generated during the second pass of applying this binding to a given Template specification.

Condition: Identifies relevant procedural or model construct-based conditions that affect the IDL that is generated.

Let: The context dependent binding of a value or element reference to a name for subsequent use. It is often used to reduce verbosity and/or improve the clarity of the binding specification.

Return: Specifies a value or element reference that is being returned by a Rule to an invoking Rule.

[]: An operator that returns a resolved unambiguous reference (by name) to the enclosed named meta-model element or to a property of a meta-model element. For clarity, the element's meta-type is identified before the [], the name of the meta-model element is identified within the [], and a property, if any, is identified after the []. Example: face.uop.Template[%IDENTIFIER%].name

%%: An operator that returns the value(s) associated with the enclosed named element. The element is either 1) a non-terminal symbol in a modeled Template specification, 2) a Let-bound name, 3) a meta-model element, or 4) a property of a meta-model element. The enclosing %% characters are not returned.

{V => <Y>}: An operator that invokes Rule Y with value V. If a "+" character follows the {}, then there is a set of Vs in context (reflected in the associated Expression), and Rule Y is invoked for each V.

<>: An operator that invokes the enclosed named Rule with the modeled value(s) associated with the same named non-terminal symbol. It is syntactic sugar for { %Y% => <Y> }. If a "+" character follows the <>, then there is a set of instances associated of the named non-terminal in context (reflected in the associated Expression), and the Rule is invoked for each instance.

(): A grouping operator that encapsulates an ordered set of concatentations of string-type data, each of which is separated by a comma (,). The result is a string. The enclosing () characters are not part of the returned value.

"": Represents a string literal value. The enclosing "" characters are not part of the string literal's value.

?: In an Expression or Alternate, a "?" character following a symbol or symbol group means the preceding symbol or symbol group may be present just once or not at all. It is syntactically equivalent to "[" S "]" in EBNF, where S is a set of symbols.

: In an Expression, Alternate or Variant, a "" character following a symbol or symbol group means the preceding symbol or symbol group may be present one or more times or not at all. It is syntactically equivalent to "{" S "}" in EBNF, where S is a set of symbols.

+: In an Expression, Alternate or Variant, a "+" character following a symbol or symbol group means the preceding symbol or symbol group is present one or more times. It is syntactically equivalent to S "{" S "}" in EBNF, where S is a set of symbols.

THIS: Represents a reference to the current construct, which is an instance of the Rule’s expression.

// The text following the // and up to the end of the line is (part of) an informative comment or note.

The input to this binding is a face.uop.MessageType (Template or Composite Template). IDL types will be generated for this MessageType and for various elements it directly or indirectly references. A MessageType and the elements it references may or may not be members of the same face.uop.UoPModel or datamodel.DataModel.

NOTES TO THE IMPLEMENTER:

For each DataModel element referenced, the IDL types generated by this binding must be defined in an IDL module (in the FACE::DM namespace) whose name is the

same as the DataModel in which the element is a member. The IDL module declaration for a DataModel is:

```
  "module" "FACE" "{"
    "module" "DM" "{"
      "module" % datamodel.DataModel.name% "{"
        // The IDL types defined in this IDL module are those
        // generated by this binding for the DataModel elements
        // that are members of datamodel.DataModel.
      "}" ";"
    "}" ";"
  "}" ";"
```

For each Template or CompositeTemplate, the IDL types generated by this binding must be defined in an IDL module (in the FACE::DM namespace) whose name is the same as the root UoPModel in which the element is a member. The IDL module declaration for a Template or CompositeTemplate is:

```
  "module" "FACE" "{"
    "module" "DM" "{"
      "module" %face.uop.UoPModel.name% "{"
        // The IDL types defined in this IDL module are those
        // generated by this binding for the Templates and CompositeTemplates
        // that are members of face.uop.UoPModel.
      "}" ";"
    "}" ";"
  "}" ";"
```

Because the generated IDL types for Templates, CompositeTemplates, and various DataModel elements may be in different IDL modules, an inter-Model type reference must manifest in IDL using an IDL module-scoped name.

This binding may visit a given Template, CompositeTemplate, or DataModel element more than once. An IDL type should not be generated more than once for a given element.

IDL struct_forward_dcl or union_forward_dcl statements may be needed for recursive types.

BINDING SPECIFICATION:

```
// CURRENT_TEMPLATE is the current face.uop.Template being
// mapped to IDL.

Let:
  CURRENT_TEMPLATE =   // nil

// CURRENT_STRUCTURED_TEMPLATE_ELEMENT is the current
// StructuredTemplateElementTypeDecl being processed, and
// CURRENT_STRUCTURED_TEMPLATE_ELEMENT_NAME is the name of that
// StructuredTemplateElementTypeDecl.

Let:
  CURRENT_STRUCTURED_TEMPLATE_ELEMENT =      // nil
  CURRENT_STRUCTURED_TEMPLATE_ELEMENT_NAME = // nil

Rule: MessageType

  // This rule was manufactured to facilitate the binding specification.
  // It is not a production rule in the Template grammar specification.
  // Its Expression is not in the Template grammar, but is instead a
  // meta-type in the FACE meta-model.
```

```
  Expression: face.uop.MessageType

    Let:
      MESSAGE_TYPE = THIS

    Condition: If %MESSAGE_TYPE% is a face.uop.CompositeTemplate:

      Emit: { %MESSAGE_TYPE % => <CompositeTemplate> }

    Condition: If %MESSAGE_TYPE% is a face.uop.Template::

      Emit: { %MESSAGE_TYPE% => <Template> }

Rule: CompositeTemplate

  // This rule was manufactured to facilitate the binding specification.
  // It is not a production rule in the Template grammar specification.
  // Its Expression is not in the Template grammar, but is instead a
  // meta-type in the FACE meta-model.

  Expression: face.uop.CompositeTemplate

    Let:
      COMPOSITE_TEMPLATE = THIS

    // Generate IDL types for its composed member's types

    For each face.uop.TemplateComposition in
%%COMPOSITE_TEMPLATE%.composition%:

      Let:
        MEMBER = face.uop.TemplateComposition

      Emit: { %%MEMBER%.type% => <MessageType> }

    // Now generate an IDL Struct for the CompositeTemplate

    Condition: If %%COMPOSITE_TEMPLATE%.isUnion% is false:

      Emit: "struct" %%TEMPLATE%.name% "{"

    Condition: If %%COMPOSITE_TEMPLATE%.isUnion% is true:

      Emit: "union" %%TEMPLATE%.name% "{" "switch" "(" "unsigned" "short" ")"
"{"

    // Generate the CompositeTemplate's members

    Let:
      CASE_NUMBER = 0

    For each face.uop.TemplateComposition in
%%COMPOSITE_TEMPLATE%.composition%:

      Let:
        MEMBER = face.uop.TemplateComposition

      Condition: If %%COMPOSITE_TEMPLATE%.isUnion% is true:

        // Increment the switch case number.

        Let:
          CASE_NUMBER = %CASE_NUMBER% + 1
```

```
        Emit: "case" %CASE_NUMBER% ":"

      Emit: %%MEMBER%.type.name% %%MEMBER%.rolename% ";"

    Condition: If %%COMPOSITE_TEMPLATE%.isUnion% is true:

      Emit: "}"

    Emit: "}" ";"

Rule: Template

  // This rule was manufactured to facilitate the binding specification.
  // It is not a production rule in the Template grammar specification.
  // Its Expression is not in the Template grammar, but is instead a
  // meta-type in the FACE meta-model.

  // Two passes of %%THIS%.specification% over TemplateSpecification are
  // required to generate the IDL for a given Template. The first pass
  // emits (via Emit1:) the supporting IDL types (e.g. the IDL types for
  // any referenced datamodel.platform.PlatformDataTypes, etc.). The second
  // pass emits (via Emit2:) the IDL types for the Template's elements.

  Expression: face.uop.Template

    Let:
      PREVIOUS_TEMPLATE = %CURRENT_TEMPLATE%
      CURRENT_TEMPLATE = THIS

    Emit1: { %%CURRENT_TEMPLATE%.specification% => <TemplateSpecification> }

    Condition: If the MainTemplateMethodDecl in %CURRENT_TEMPLATE% is a
MainEntityTypeTemplateMethodDecl:

      Emit2: "module" ( "T_" , %%CURRENT_TEMPLATE%.name% ) "{"

    Emit2: { %%CURRENT_TEMPLATE%.specification% => <TemplateSpecification> }

    Condition: If the MainTemplateMethodDecl in %CURRENT_TEMPLATE% is a
MainEntityTypeTemplateMethodDecl:

      Emit2: "}" ";"

      Emit2: "typedef" ( "T_" , %%CURRENT_TEMPLATE%.name% , "::" ,
%%CURRENT_TEMPLATE%.name% ) %%CURRENT_TEMPLATE%.name%  ";"

    Let:
      CURRENT_TEMPLATE = %PREVIOUS_TEMPLATE%

Rule: TemplateSpecification

  Expression: UsingExternalTemplateStatement*
StructuredTemplateElementTypeDecl+

    Variant: StructuredTemplateElementTypeDecl+

      Emit: <StructuredTemplateElementTypeDecl>+

    Variant: UsingExternalTemplateStatement+ StructuredTemplateElementTypeDecl+

      Emit1: <UsingExternalTemplateStatement>+
```

```
      Emit: <StructuredTemplateElementTypeDecl>+

Rule: UsingExternalTemplateStatement

  Expression: KW_USING ExternalTemplateTypeReference SEMICOLON

    Emit1: { %<ExternalTemplateTypeReference>% => <Template> }

Rule: StructuredTemplateElementTypeDecl

  Expression: // The following Alternates comprise the
StructuredTemplateElementTypeDecl Expression

    Let:
      CURRENT_STRUCTURED_TEMPLATE_ELEMENT = %StructuredTemplateElementTypeDecl%

    Alternate: MainTemplateMethodDecl

      Emit: <MainTemplateMethodDecl>

    Alternate: SupportingTemplateMethodDecl

      Emit: <SupportingTemplateMethodDecl>

    Alternate: UnionTypeDecl

      Emit: <UnionTypeDecl>

    Let:
      CURRENT_STRUCTURED_TEMPLATE_ELEMENT =    //nil

Rule: MainTemplateMethodDecl

  Expression: // The following Alternates comprise the MainTemplateMethodDecl
Expression

    Let:
      CURRENT_STRUCTURED_TEMPLATE_ELEMENT_NAME = %CURRENT_TEMPLATE%.name

    Alternate: MainEntityTypeTemplateMethodDecl

      Emit: <MainEntityTypeTemplateMethodDecl>

    Alternate: MainEquivalentEntityTypeTemplateMethodDecl

      Emit: <MainEquivalentEntityTypeTemplateMethodDecl>

    Let:
      CURRENT_STRUCTURED_TEMPLATE_ELEMENT_NAME =    //nil

Rule: MainEntityTypeTemplateMethodDecl

  Expression: KW_MAIN LEFT_PAREN PrimaryEntityTypeTemplateMethodParameter? (
COMMA KW_VARARGS OptionalEntityTypeTemplateMethodParameterList? )? RIGHT_PAREN
EntityTypeTemplateMethodBody

    Emit2: "struct" %CURRENT_STRUCTURED_TEMPLATE_ELEMENT_NAME%

    Emit: <EntityTypeTemplateMethodBody>

    Emit2: ";"

Rule: MainEquivalentEntityTypeTemplateMethodDecl
```

```
  // The Expressions for the EquivalentEntityTypeTemplateMethodDecl and
  // EquivalentEntityTypeTemplateMethodBody Rules are in-lined here to
consolidate multiple emits
  // of the CURRENT_TEMPLATE's name to one rule, and to allow for multiple
emits based on the
  // modeled EquivalentEntityTypeTemplateMethodMembers.

  Expression: KW_MAIN LEFT_ANGLE_BRACKET
EquivalentEntityTypeTemplateMethodParameterList RIGHT_ANGLE_BRACKET LEFT_BRACE
EquivalentEntityTypeTemplateMethodMember+ RIGHT_BRACE

    // Declare an IDL Template Module for this
MainEquivalentEntityTypeTemplateMethodDecl.

    Emit1: "module" ( %CURRENT_STRUCTURED_TEMPLATE_ELEMENT_NAME% , "_Module" )

    Emit1: "<"

    // Generate the IDL Template Module's formal parameter list.

    For each EquivalentEntityTypeTemplateMethodMember of
MainEquivalentEntityTypeTemplateMethodDecl:

      Condition: If %EquivalentEntityTypeTemplateMethodMember% is not the first
EquivalentEntityTypeTemplateMethodMember of
MainEquivalentEntityTypeTemplateMethodDecl:

        Emit1: ","

      Emit1: { %EquivalentEntityTypeTemplateMethodMember% =>
<GenerateIDLTemplateModuleDeclFormalParameterFromEquivalentEntityTypeTemplateMe
thodMember> }

    // Close the IDL Template Module's formal parameter list.

    Emit1: ">"

    Emit1: "{" "struct" %CURRENT_STRUCTURED_TEMPLATE_ELEMENT_NAME% "{"
<EquivalentEntityTypeTemplateMethodMember>+ "}" ";" "}" ";"

Rule: SupportingTemplateMethodDecl

  Expression: // The following Alternates comprise the
SupportingTemplateMethodDecl Expression

    Alternate: SupportingEntityTypeTemplateMethodDecl

        Emit: <SupportingEntityTypeTemplateMethodDecl>

    Alternate: SupportingEquivalentEntityTypeTemplateMethodDecl

        Emit: <SupportingEquivalentEntityTypeTemplateMethodDecl>

Rule: SupportingEntityTypeTemplateMethodDecl

  Expression: TemplateElementTypeName LEFT_PAREN
PrimaryEntityTypeTemplateMethodParameter RIGHT_PAREN
EntityTypeTemplateMethodBody

    Let:
      CURRENT_STRUCTURED_TEMPLATE_ELEMENT_NAME = <TemplateElementTypeName>
```

```
    Emit2: "struct" %CURRENT_STRUCTURED_TEMPLATE_ELEMENT_NAME%

    Emit: <EntityTypeTemplateMethodBody>

    Emit2: ";"

    Let:
      CURRENT_STRUCTURED_TEMPLATE_ELEMENT_NAME =    // nil

Rule: SupportingEquivalentEntityTypeTemplateMethodDecl

  // The Expressions for the EquivalentEntityTypeTemplateMethodDecl and
  // EquivalentEntityTypeTemplateMethodBody Rules are in-lined here to
consolidate multiple emits
  // of the TemplateElementTypeName to one rule, and to allow for multiple
emits based on the
  // EquivalentEntityTypeTemplateMethodMembers.

  Expression: TemplateElementTypeName LEFT_ANGLE_BRACKET
EquivalentEntityTypeTemplateMethodParameterList RIGHT_ANGLE_BRACKET LEFT_BRACE
EquivalentEntityTypeTemplateMethodMember+ RIGHT_BRACE

    Let:
      CURRENT_STRUCTURED_TEMPLATE_ELEMENT_NAME = <TemplateElementTypeName>

    // Declare an IDL Template Module for this
SupportingEquivalentEntityTypeTemplateMethodDecl.

    Emit1: "module" ( %CURRENT_STRUCTURED_TEMPLATE_ELEMENT_NAME% , "_Module" )

    Emit1: "<"

    // Generate the IDL Template Module's formal parameter list.

    For each EquivalentEntityTypeTemplateMethodMember of
SupportingEquivalentEntityTypeTemplateMethodDecl:

      Condition: If %EquivalentEntityTypeTemplateMethodMember% is not the first
EquivalentEntityTypeTemplateMethodMember of
SupportingEquivalentEntityTypeTemplateMethodDecl:

        Emit1: ","

      Emit: { %EquivalentEntityTypeTemplateMethodMember% =>
<GenerateIDLTemplateModuleDeclFormalParameterFromEquivalentEntityTypeTemplateMe
thodMember> }

    // Close the IDL Template Module's formal parameter list.

    Emit1: ">"

    Emit1: "{" "struct" %CURRENT_STRUCTURED_TEMPLATE_ELEMENT_NAME% "{"
<EquivalentEntityTypeTemplateMethodMember>+ "}" ";" "}" ";"

    Let:
      CURRENT_STRUCTURED_TEMPLATE_ELEMENT_NAME =    // nil

Rule:
GenerateIDLTemplateModuleDeclFormalParameterFromEquivalentEntityTypeTemplateMet
hodMember

  // This rule was manufactured to facilitate the binding specification.
  // It is not a production rule in the Template grammar specification.
```

```
  // Its Expression is the EquivalentEntityTypeTemplateMethodMember Rule's
  // Expression in the grammar.

  Expression: EquivalentEntityTypeTemplateElementMemberStatement SEMICOLON

    Emit1: { %EquivalentEntityTypeTemplateElementMemberStatement% =>
<GenerateIDLTemplateModuleDeclFormalParameterFromEquivalentEntityTypeTemplateMe
thodMemberStatement> }

Rule:
GenerateIDLTemplateModuleDeclFormalParameterFromEquivalentEntityTypeTemplateMet
hodMemberStatement

  // This rule was manufactured to facilitate the binding specification.
  // It is not a production rule in the Template grammar specification.
  // Its Expression is the EquivalentEntityTypeTemplateElementMemberStatement
  // Rule's Expression in the grammar.

  Expression: OptionalAnnotation?
DesignatedEquivalentEntityNonEntityTypeCharacteristicReference ( DEREF
IDLStructMemberReference )? ( KW_AS StructuredTemplateElementMemberName )?

    Variant: DesignatedEquivalentEntityNonEntityTypeCharacteristicReference
    Variant: OptionalAnnotation
DesignatedEquivalentEntityNonEntityTypeCharacteristicReference

      Emit1: "typename" (
%<DesignatedEquivalentEntityNonEntityTypeCharacteristicReference>% , "_type" )

    Variant: DesignatedEquivalentEntityNonEntityTypeCharacteristicReference
DEREF IDLStructMemberReference
    Variant: OptionalAnnotation
DesignatedEquivalentEntityNonEntityTypeCharacteristicReference DEREF
IDLStructMemberReference

      Emit1: "typename" ( %%<IDLStructMemberReference>%.rolename% , "_type" )

    Variant: DesignatedEquivalentEntityNonEntityTypeCharacteristicReference
KW_AS StructuredTemplateElementMemberName
    Variant: DesignatedEquivalentEntityNonEntityTypeCharacteristicReference
DEREF IDLStructMemberReference KW_AS StructuredTemplateElementMemberName
    Variant: OptionalAnnotation
DesignatedEquivalentEntityNonEntityTypeCharacteristicReference KW_AS
StructuredTemplateElementMemberName
    Variant: OptionalAnnotation
DesignatedEquivalentEntityNonEntityTypeCharacteristicReference DEREF
IDLStructMemberReference KW_AS StructuredTemplateElementMemberName

      Emit1: "typename" ( %<StructuredTemplateElementMemberName>% , "_type" )

Rule: EntityTypeTemplateMethodBody

  Expression: LEFT_BRACE EntityTypeTemplateMethodMember+ RIGHT_BRACE

    Emit2: "{"

    Emit: <EntityTypeTemplateMethodMember>+

    Emit2: "}"

Rule: EntityTypeTemplateMethodMember

  Expression: EntityTypeStructuredTemplateElementMember
```

```
    Emit: <EntityTypeStructuredTemplateElementMember>

Rule: EquivalentEntityTypeTemplateMethodMember

  Expression: EquivalentEntityTypeTemplateElementMemberStatement SEMICOLON

    Emit1: <EquivalentEntityTypeTemplateElementMemberStatement>

Rule: EquivalentEntityTypeTemplateElementMemberStatement

  Expression: OptionalAnnotation?
DesignatedEquivalentEntityNonEntityTypeCharacteristicReference ( DEREF
IDLStructMemberReference )? ( KW_AS StructuredTemplateElementMemberName )?

    Variant: DesignatedEquivalentEntityNonEntityTypeCharacteristicReference
    Variant: OptionalAnnotation
DesignatedEquivalentEntityNonEntityTypeCharacteristicReference

      Let:
        TEMPLATE_MODULE_TYPE_MEMBER_NAME =
<DesignatedEquivalentEntityNonEntityTypeCharacteristicReference>

      Emit1: ( %TEMPLATE_MODULE_TYPE_MEMBER_NAME% , "_type" )
%TEMPLATE_MODULE_TYPE_MEMBER_NAME%

    Variant: DesignatedEquivalentEntityNonEntityTypeCharacteristicReference
DEREF IDLStructMemberReference
    Variant: OptionalAnnotation
DesignatedEquivalentEntityNonEntityTypeCharacteristicReference DEREF
IDLStructMemberReference

      Let:
        TEMPLATE_MODULE_TYPE_MEMBER_NAME =
%<IDLStructMemberReference>%.rolename

      Emit1: ( %TEMPLATE_MODULE_TYPE_MEMBER_NAME% , "_type" )
%TEMPLATE_MODULE_TYPE_MEMBER_NAME%

    Variant: DesignatedEquivalentEntityNonEntityTypeCharacteristicReference
KW_AS StructuredTemplateElementMemberName
    Variant: DesignatedEquivalentEntityNonEntityTypeCharacteristicReference
DEREF IDLStructMemberReference KW_AS StructuredTemplateElementMemberName
    Variant: OptionalAnnotation
DesignatedEquivalentEntityNonEntityTypeCharacteristicReference KW_AS
StructuredTemplateElementMemberName
    Variant: OptionalAnnotation
DesignatedEquivalentEntityNonEntityTypeCharacteristicReference DEREF
IDLStructMemberReference KW_AS StructuredTemplateElementMemberName

      Let:
        TEMPLATE_MODULE_TYPE_MEMBER_NAME =
<StructuredTemplateElementMemberName>

      Emit1: ( %TEMPLATE_MODULE_TYPE_MEMBER_NAME% , "_type" )
%TEMPLATE_MODULE_TYPE_MEMBER_NAME%

Rule: UnionTypeDecl

  Expression: KW_UNION TemplateElementTypeName LEFT_PAREN UnionParameter
RIGHT_PAREN UnionBody

    Let:
```

```
      CURRENT_STRUCTURED_TEMPLATE_ELEMENT_NAME = <TemplateElementTypeName>

    Emit2: "union" %<TemplateElementTypeName>%

    Emit: <UnionBody>

    Emit2: ";"

    Let:
      CURRENT_STRUCTURED_TEMPLATE_ELEMENT_NAME =    // nil

Rule: UnionBody

  Expression: LEFT_BRACE UnionSwitchStatement RIGHT_BRACE

    Emit2: "{"

    Emit: <UnionSwitchStatement>

    Emit2: "}"

Rule: UnionSwitchStatement

  Expression: KW_SWITCH LEFT_PAREN DiscriminatorType RIGHT_PAREN
UnionSwitchBody

    Emit2: "switch" "("

    Emit: <DiscriminatorType>

    Emit2: ")"

    Emit: <UnionSwitchBody>

Rule: UnionSwitchBody

  Expression: LEFT_BRACE CaseExpression+ RIGHT_BRACE

    Emit2: "{"

    Emit: <CaseExpression>+

    Emit2: "}"

Rule: DiscriminatorType

  Expression: // The following Alternates comprise the DiscriminatorType
Expression

    Alternate: IDLDiscriminatorType

      Emit2: %IDLDiscriminatorType%

    Alternate: DesignatedEntityEnumerationTypeCharacteristicReference

      Emit1: {
%%<DesignatedEntityEnumerationTypeCharacteristicReference>%.type% =>
<PlatformIDLType> }

      Emit2:
%%<DesignatedEntityEnumerationTypeCharacteristicReference>%.type.name%

Rule: CaseExpression
```

```
  Expression: CaseLabel+ UnionMember

    Emit2: <CaseLabel>+

    Emit: <UnionMember>

Rule: CaseLabel

  Expression: // The following Alternates comprise the CaseLabel Expression

    Alternate: KW_CASE CaseLabelLiteral COLON

      Emit2: "case" <CaseLabelLiteral> ":"

    Alternate: KW_DEFAULT COLON

      Emit2: "default" ":"

Rule: CaseLabelLiteral

  Expression: // The following Alternates comprise the CaseLabelLiteral
Expression

    Alternate: IDLDiscriminatorTypeLiteral

      Emit2: %IDLDiscriminatorTypeLiteral%

    Alternate: EnumLiteralReferenceExpression

      Emit2: %<EnumLiteralReferenceExpression>%

Rule: UnionMember

  Expression: EntityTypeStructuredTemplateElementMember

    Emit: <EntityTypeStructuredTemplateElementMember>

Rule: EntityTypeStructuredTemplateElementMember

  Expression: EntityTypeStructuredTemplateElementMemberStatement SEMICOLON

    Emit: <EntityTypeStructuredTemplateElementMemberStatement>

Rule: EntityTypeStructuredTemplateElementMemberStatement

  Expression: // The following Alternates comprise the
        // EntityTypeStructuredTemplateElementMemberStatement Expression

    Alternate: DesignatedEntityCharacteristicReferenceStatement

      Emit: <DesignatedEntityCharacteristicReferenceStatement>

    Alternate: StructuredTemplateElementTypeReferenceStatement

      Emit: <StructuredTemplateElementTypeReferenceStatement>

Rule: DesignatedEntityCharacteristicReferenceStatement

  Expression: // The following Alternates comprise the
        // DesignatedEntityCharacteristicReferenceStatement Expression
```

```
    Alternate:
ExplicitDesignatedEntityNonEntityTypeCharacteristicReferenceExpression

      Emit:
<ExplicitDesignatedEntityNonEntityTypeCharacteristicReferenceExpression>

    Alternate: DesignatedEntityNonEntityTypeCharacteristicWildcardReference

      Emit: <DesignatedEntityNonEntityTypeCharacteristicWildcardReference>

Rule: ExplicitDesignatedEntityNonEntityTypeCharacteristicReferenceExpression

  Expression: OptionalAnnotation?
DesignatedEntityNonEntityTypeCharacteristicReference ( DEREF
IDLStructMemberReference )? ( KW_AS StructuredTemplateElementMemberName )?

    // Emit the IDL for the
DesignatedEntityNonEntityTypeCharacteristicReference's type.

    Emit1: { %THIS% =>
<GenerateExplicitDesignatedEntityNonEntityTypeCharacteristicReferenceIDL> }

    Let:
      C_TYPE_NAME = { %THIS% =>
<GetExplicitDesignatedEntityNonEntityTypeCharacteristicReferenceMemberIDLType>
}

    Variant: DesignatedEntityNonEntityTypeCharacteristicReference
    Variant: OptionalAnnotation
DesignatedEntityNonEntityTypeCharacteristicReference
    Variant: DesignatedEntityNonEntityTypeCharacteristicReference DEREF
IDLStructMemberReference
    Variant: OptionalAnnotation
DesignatedEntityNonEntityTypeCharacteristicReference DEREF
IDLStructMemberReference

      Let:
        C_ROLE_NAME =
%<DesignatedEntityNonEntityTypeCharacteristicReference>%.rolename

      Emit2: %C_TYPE_NAME% %C_ROLE_NAME% ";"

    Variant: DesignatedEntityNonEntityTypeCharacteristicReference KW_AS
StructuredTemplateElementMemberName
    Variant: OptionalAnnotation
DesignatedEntityNonEntityTypeCharacteristicReference KW_AS
StructuredTemplateElementMemberName
    Variant: DesignatedEntityNonEntityTypeCharacteristicReference DEREF
IDLStructMemberReference KW_AS StructuredTemplateElementMemberName
    Variant: OptionalAnnotation
DesignatedEntityNonEntityTypeCharacteristicReference DEREF
IDLStructMemberReference KW_AS StructuredTemplateElementMemberName

      Let:
        C_ROLE_NAME = <StructuredTemplateElementMemberName>

      Emit2: %C_TYPE_NAME% %C_ROLE_NAME% ";"

Rule: DesignatedEntityNonEntityTypeCharacteristicWildcardReference

  Expression: ( DesignatedEntityTypeReferencePath PERIOD )? ASTERISK
```

```
    // A DesignatedEntityNonEntityTypeCharacteristicWildcardReference is a
"wildcard reference"
    // to one or more explicit
DesignatedEntityNonEntityTypeCharacteristicReferences. Each
    // DesignatedEntityNonEntityTypeCharacteristicReference is a
    // datamodel.platform.Composition that is 1) composed in the
    // DesignatedEntityTypeReferencePath's DesignatedEntityTypeReference to
which the ASTERISK is
    // applied (i.e. the Composition is a member of
<DesignatedEntityTypeReference>.composition),
    // and 2) is projected by the Template's corresponding Query.

    For each datamodel.platform.Composition in
%%<DesignatedEntityTypeReference>%.composition%:

      // Generate a DesignatedEntityNonEntityTypeCharacteristicReference
Template construct for
      // this datamodel.platform.Composition.

      Let:
        GENERATED_CHARACTERISTIC_REFERENCE = (
%%<DesignatedEntityTypeReference>%.name% , "." ,
%datamodel.platform.Composition.rolename% )

      // Emit the generated
DesignatedEntityNonEntityTypeCharacteristicReference using the
      // ExplicitDesignatedEntityNonEntityTypeCharacteristicReferenceExpression
rule.

      Emit: { %GENERATED_EXPLICIT_CHARACTERISTIC_REFERENCE% =>
<ExplicitDesignatedEntityNonEntityTypeCharacteristicReferenceExpression> }

Rule: StructuredTemplateElementTypeReferenceStatement

  // This rule was manufactured to facilitate the binding specification.
  // While it is a production rule in the Template grammar specification,
  // its Expression has been modified here to separate each Alternate of
  // the original Rule in the grammar into a separate Rule.

  Expression: // The following Alternates comprise the
         // StructuredTemplateElementTypeReferenceStatement Expression

    Alternate: StructuredTemplateElementTypeMemberReferenceStatement

      Emit: <StructuredTemplateElementTypeMemberReferenceStatement>

    Alternate: DirectStructuredTemplateElementTypeReferenceStatement

      Emit: <DirectStructuredTemplateElementTypeReferenceStatement>

Rule: StructuredTemplateElementTypeMemberReferenceStatement

  // This rule was manufactured to facilitate the binding specification.
  // It is not a production rule in the Template grammar specification.
  // Its Expression effectively separates the Alternates of the original
  // StructuredTemplateElementTypeReferenceExpression Rule in the grammar
  // into separate Rules.

  Expression: // The following Alternates comprise the
         // StructuredTemplateElementTypeMemberReferenceStatement Expression

    Alternate: EntityTypeStructuredTemplateElementTypeMemberReferenceStatement
```

```
      Emit: <EntityTypeStructuredTemplateElementTypeMemberReferenceStatement>

    Alternate: EquivalentEntityTypeTemplateMethodMemberReferenceStatement

      Emit: <EquivalentEntityTypeTemplateMethodMemberReferenceStatement>

Rule: EntityTypeStructuredTemplateElementTypeMemberReferenceStatement

  // This rule was manufactured to facilitate the binding specification.
  // It is not a production rule in the Template grammar specification.
  // Its Expression is a combination of the InlineAnnotation Alternate
  // of the StructuredTemplateElementTypeReferenceStatement and the
  // EntityTypeStructuredTemplateElementTypeReference-type Alternate of
  // the StructuredTemplateElementTypeReferenceExpression Rule in the grammar.

  Expression: InlineAnnotation EntityTypeStructuredTemplateElementTypeReference
LEFT_PAREN StructuredTemplateElementTypeReferenceParameterList RIGHT_PAREN

    // Emit each EntityTypeStructuredTemplateElementMember of the
    // StructuredTemplateElementTypeDecl referenced by
    // EntityTypeStructuredTemplateElementTypeReference as members of the
enclosing
    // StructuredTemplateElementType.

    Emit: <EntityTypeStructuredTemplateElementMember>+

Rule: EquivalentEntityTypeTemplateMethodMemberReferenceStatement

  // This rule was manufactured to facilitate the binding specification.
  // It is not a production rule in the Template grammar specification.
  // Its Expression is a combination of the InlineAnnotation Alternate
  // of the StructuredTemplateElementTypeReferenceStatement and the
  // EquivalentEntityTypeTemplateMethodReference-type Alternate of
  // the StructuredTemplateElementTypeReferenceExpression Rule in the grammar.

  Expression: InlineAnnotation EquivalentEntityTypeTemplateMethodReference
LEFT_ANGLE_BRACKET StructuredTemplateElementTypeReferenceParameterList
RIGHT_ANGLE_BRACKET

    // Emit each EquivalentEntityTypeTemplateMethodMember of the
    // StructuredTemplateElementTypeDecl referenced by the
    // EquivalentEntityTypeTemplateMethodReference as members of the enclosing
    // StructuredTemplateElementType.

    Emit: { %EquivalentEntityTypeTemplateMethodMember% =>
<InlineEquivalentEntityTypeTemplateMethodMemberStatement> }+

Rule: InlineEquivalentEntityTypeTemplateMethodMemberStatement

  // This rule was manufactured to facilitate the binding specification.
  // It is not a production rule in the Template grammar specification.
  // Its Expression is the EquivalentEntityTypeTemplateMethodMember's Rule
  // in the grammar.

  Expression: EquivalentEntityTypeTemplateElementMemberStatement SEMICOLON

    Let:
      INSTANTIATED_MEMBER_STATEMENT = {
%EquivalentEntityTypeTemplateElementMemberStatement% =>
<GetInstantiatedEquivalentEntityTypeTemplateElementMemberStatement> }

    Emit: { %INSTANTIATED_MEMBER_STATEMENT% =>
<ExplicitDesignatedEntityNonEntityTypeCharacteristicReferenceExpression> }
```

```
Rule: GetInstantiatedEquivalentEntityTypeTemplateElementMemberStatement

  // This rule was manufactured to facilitate the binding specification.
  // It is not a production rule in the Template grammar specification.
  // Its Expression is the EquivalentEntityTypeTemplateElementMemberStatement
  // Rule's Expression in the grammar.

  // The Expression for the
DesignatedEquivalentEntityNonEntityTypeCharacteristicReference Rule is
  // in-lined here as its non-terminals are referenced below.

  Expression: OptionalAnnotation?
EquivalentEntityTypeTemplateMethodParameterReference PERIOD
EquivalentEntityTypeTemplateMethodCharacteristicReference ( DEREF
IDLStructMemberReference )? ( KW_AS StructuredTemplateElementMemberName )?

    // Instantiate the EquivalentEntityTypeTemplateMethodMemberStatement by
transforming it into
    // an
ExplicitDesignatedEntityNonEntityTypeCharacteristicReferenceExpression
    // type of EntityTypeStructuredTemplateElementMemberStatement by replacing
the
    // EquivalentEntityTypeTemplateMethodMember's
    // EquivalentEntityTypeTemplateMethodParameterReference with the
    // DesignatedEntityTypeReference specified for the
    // EquivalentEntityTypeTemplateMethodParameterReference's corresponding
    // EquivalentEntityTypeTemplateMethodParameter in the
    // EquivalentEntityTypeTemplateMethodReference's
    // StructuredTemplateElementTypeReferenceParameterList (which is specified
positionally or
    // assigned via a
EntityTypeStructuredTemplateElementDeclaredParameterReference).

    Let:
      EquivalentEntityTypeTemplateMethodParameterReference =
%%<DesignatedEntityTypeReference>%.name%

    // This "instantiated EquivalentEntityTypeTemplateMethodMemberStatement" is
now effectively
    // an
ExplicitDesignatedEntityNonEntityTypeCharacteristicReferenceExpression.

    Return: THIS

Rule: DirectStructuredTemplateElementTypeReferenceStatement

  // This rule was manufactured to facilitate the binding specification.
  // It is not a production rule in the Template grammar specification.
  // Its Expression effectively separates the Alternates of the original
  // StructuredTemplateElementTypeReferenceExpression Rule in the grammar
  // into separate Rules.

  Expression: // The following Alternates comprise the
         // DirectStructuredTemplateElementTypeReferenceStatement Expression

    Alternate: DirectEntityTypeStructuredTemplateElementTypeReferenceStatement

      Emit: <DirectEntityTypeStructuredTemplateElementTypeReferenceStatement>

    Alternate: DirectEquivalentEntityTypeTemplateMethodReferenceStatement

      Emit: <DirectEquivalentEntityTypeTemplateMethodReferenceStatement>
```

```
Rule: DirectEntityTypeStructuredTemplateElementTypeReferenceStatement

  // This rule was manufactured to facilitate the binding specification.
  // It is not a production rule in the Template grammar specification.
  // Its Expression is a combination of the OptionalAnnotation Alternate
  // of the StructuredTemplateElementTypeReferenceStatement and the
  // EntityTypeStructuredTemplateElementTypeReference-type Alternate of
  // the StructuredTemplateElementTypeReferenceExpression Rule in the grammar.

  // The Expression for the StructuredTemplateElementTypeReferenceParameterList
Rule
  // is in-lined here as its non-terminals are referenced in the emit
conditional
  // rules below.

  Expression: OptionalAnnotation?
EntityTypeStructuredTemplateElementTypeReference LEFT_PAREN
PrimaryStructuredTemplateElementTypeReferenceParameter ( COMMA
OptionalStructuredTemplateElementTypeReferenceParameter )* RIGHT_PAREN
StructuredTemplateElementMemberName

    // The C_MULTI and DE_REF conditional rules below refer to Characteristics
that
    // join the JoinPathEntityTypeReferences along the
EntityTypeReferenceJoinPath
    // from the PrimaryEntityTypeTemplateMethodParameter's EntityTypeReference
of
    // %CURRENT_STRUCTURED_TEMPLATE_ELEMENT% to the
    // PrimaryStructuredTemplateElementTypeReferenceParameter's
    // DesignatedEntityTypeReference. An example EntityTypeReferenceJoinPath is
    // A.B.C, where A, B, and C are JoinPathEntityTypeReferences. Each L.R pair
    // in an EntityTypeReferenceJoinPath represents and is unambiguously
traceable to
    // a Entity join in %%CURRENT_TEMPLATE%.boundQuery%. L and R represent the
    // Entitys specified in that join, and the "." represents the
Characteristic used
    // to join them. In the example, there are 2 joins represented: the first
is A.B
    // and the second is B.C. There are 2 joining Characteristics, one for each
join:
    // one that joins A and B, and the other that joins B and C.

    // The C_MULTI condition determines the multiplicity of a joining
Characteristic
    // for the subsequent DE_REF conditional rules. In the C_MULTI condition, L
    // represents the Characteristic's left JoinPathEntityTypeReference (the
Entity
    // to the left of a "."), and R represents the Characteristic's right
    // JoinPathEntityTypeReference (the Entity to the right of that ".").

    C_MULTI Conditional Rule:
      Let:
        JOIN_CHARACTERISTIC = datamodel.platform.Characteristic  // a "."

      If %JOIN_CHARACTERISTIC% is a Composition or Participant of L, then:

        If %JOIN_CHARACTERISTIC% is not in the EntityTypeReferenceJoinPath
between the MainEntityTypeTemplateMethodDecl and the current UnionTypeDecl or
SupportingEntityTypeTemplateMethodDecl, then:

          Let:
            C_MULTI_UPPER_BOUND = %JOIN_CHARACTERISTIC%.upperBound
```

```
            C_MULTI_LOWER_BOUND = %JOIN_CHARACTERISTIC%.lowerBound

        Otherwise:

          Let:
            C_MULTI_UPPER_BOUND = 1
            C_MULTI_LOWER_BOUND = 1

        If R is a SelectedEntity that is a SelectedEntityReference of a
SelectedEntityCharacteristicReference CharacteristicBasis in
%%CURRENT_TEMPLATE%.boundQuery%, then:

          Let:
            C_MULTI_LOWER_BOUND = 0

      Otherwise: // %JOIN_CHARACTERISTIC% is a Composition or Participant of R:

        If %JOIN_CHARACTERISTIC% is not in the EntityTypeReferenceJoinPath
between the MainEntityTypeTemplateMethodDecl and the current UnionTypeDecl or
SupportingEntityTypeTemplateMethodDecl, then:

          If %JOIN_CHARACTERISTIC% is a Composition, then:

            Let:
              C_MULTI_UPPER_BOUND = 1
              C_MULTI_LOWER_BOUND = 1

          Otherwise: // %JOIN_CHARACTERISTIC% is a Participant

            Let:
              C_MULTI_UPPER_BOUND = %JOIN_CHARACTERISTIC%.sourceUpperBound
              C_MULTI_LOWER_BOUND = %JOIN_CHARACTERISTIC%.sourceLowerBound

        Otherwise:

            Let:
              C_MULTI_UPPER_BOUND = 1
              C_MULTI_LOWER_BOUND = 1

        If L is a SelectedEntity that is a SelectedEntityReference of a
SelectedEntityCharacteristicReference CharacteristicBasis in
%%CURRENT_TEMPLATE%.boundQuery%:

          Let:
            C_MULTI_LOWER_BOUND = 0

    The following 4 emit conditional rules are used to determine the IDL type
for a DirectEntityTypeStructuredTemplateElementTypeReferenceStatement's type.
These rules use the C_MULTI_LOWER_BOUND and C_MULTI_UPPER_BOUND determined by
the C_MULTI conditional rule above for each joining Characteristic:

      Emit Conditional Rule DE_REF1:
        If C_MULTI_UPPER_BOUND of any of the joining Characteristics is
unbounded, then the type emitted is an unbounded sequence.

      Emit Conditional Rule DE_REF2:
        If DE_REF1 does not hold and C_MULTI_LOWER_BOUND <> C_MULTI_UPPER_BOUND
for any of the joining Characteristics, then the type emitted is a bounded
sequence whose bound is the product of all of the joining Characteristic’s
C_MULTI_UPPER_BOUND.

        Let: DE_UPPERBOUND_PRODUCT_VALUE = // the calculated product
```

```
      Emit Conditional Rule DE_REF3:
        If DE_REF1 and DE_REF2 do not hold, and C_MULTI_UPPER_BOUND > 1 and
C_MULTI_LOWER_BOUND = C_MULTI_UPPER_BOUND for any of the joining
Characteristics, then the type emitted is an array whose size is the product of
all joining Characteristic's C_MULTI_UPPER_BOUND.

        Let: DE_UPPERBOUND_PRODUCT_VALUE = // the calculated product

      Emit Conditional Rule DE_REF4:
        If DE_REF1, DE_REF2, and DE_REF3 do not hold, then C_MULTI_UPPER_BOUND
= 1 and C_MULTI_LOWER_BOUND = 1 for all of the joining Characteristics, and the
type emitted is unary.

    Let:
      DE_TYPE_NAME = <EntityTypeStructuredTemplateElementTypeReference>
      DE_ROLE_NAME = <StructuredTemplateElementMemberName>

    Variant: EntityTypeStructuredTemplateElementTypeReference LEFT_PAREN
PrimaryStructuredTemplateElementTypeReferenceParameter ( COMMA
OptionalStructuredTemplateElementTypeReferenceParameter )* RIGHT_PAREN
StructuredTemplateElementMemberName

      Condition: If DE_REF1 holds:

        Let:
          MEMBER_IDL_TYPE = "sequence" "<" %DE_TYPE_NAME% ">"
          MEMBER_TYPE_NAME = ( "Seq_" , %DE_TYPE_NAME% )

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"

        Emit2: %MEMBER_TYPE_NAME% %DE_ROLE_NAME% ";"

      Condition: If DE_REF2 holds:

        Let:
          MEMBER_IDL_TYPE = "sequence" "<" %DE_TYPE_NAME% ","
%C_UPPERBOUND_PRODUCT_VALUE% ">"
          MEMBER_TYPE_NAME = ( "Seq_" , %C_UPPERBOUND_PRODUCT_VALUE% , "_" ,
%DE_TYPE_NAME% )

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"

        Emit2: %MEMBER_TYPE_NAME% %DE_ROLE_NAME% ";"

      Condition: If DE_REF3 holds:

        Let:
          MEMBER_IDL_TYPE = %DE_TYPE_NAME%
          MEMBER_TYPE_NAME = ( "Array_" , %C_UPPERBOUND_PRODUCT_VALUE% , "_" ,
%DE_TYPE_NAME% )

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% "["
%C_UPPERBOUND_PRODUCT_VALUE% "]" ";"

        Emit2: %MEMBER_TYPE_NAME% %DE_ROLE_NAME% ";"

      Condition: If DE_REF4 holds:

        Let:
          MEMBER_IDL_TYPE = %DE_TYPE_NAME%
          MEMBER_TYPE_NAME = %MEMBER_IDL_TYPE%

        Emit2: %MEMBER_TYPE_NAME% %DE_ROLE_NAME% ";"
```

```
    Variant: OptionalAnnotation
EntityTypeStructuredTemplateElementTypeReference LEFT_PAREN
PrimaryStructuredTemplateElementTypeReferenceParameter ( COMMA
OptionalStructuredTemplateElementTypeReferenceParameter )* RIGHT_PAREN
StructuredTemplateElementMemberName

      Condition: If DE_REF1 holds:

        Let:
          IDL_TYPE = "sequence" "<" %DE_TYPE_NAME% ">"
          IDL_TYPE_NAME = ( "Seq_" , %DE_TYPE_NAME% )
          MEMBER_IDL_TYPE = "sequence" "<" %IDL_TYPE_NAME%  "," "1" ">"
          MEMBER_TYPE_NAME = ( "Opt_" , %IDL_TYPE_NAME% )

        Emit1: "typedef" %IDL_TYPE% %IDL_TYPE_NAME% ";"

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"

        Emit2: %MEMBER_TYPE_NAME% %DE_ROLE_NAME% ";"

      Condition: If DE_REF2 holds:

        Let:
          IDL_TYPE = "sequence" "<" %DE_TYPE_NAME% ","
%C_UPPERBOUND_PRODUCT_VALUE% ">"
          IDL_TYPE_NAME = ( "Seq_" , %C_UPPERBOUND_PRODUCT_VALUE% , "_" ,
%DE_TYPE_NAME% )
          MEMBER_IDL_TYPE = "sequence" "<" %IDL_TYPE_NAME%  "," "1" ">"
          MEMBER_TYPE_NAME = ( "Opt_" , %IDL_TYPE_NAME% )

        Emit1: "typedef" %IDL_TYPE% %IDL_TYPE_NAME% ";"

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"

        Emit2: %MEMBER_TYPE_NAME% %DE_ROLE_NAME% ";"

      Condition: If DE_REF3 holds:

        Let:
          IDL_TYPE = %DE_TYPE_NAME%
          IDL_TYPE_NAME = ( "Array_" , %C_UPPERBOUND_PRODUCT_VALUE% , "_" ,
%DE_TYPE_NAME% )
          MEMBER_IDL_TYPE = "sequence" "<" %IDL_TYPE_NAME%  "," "1" ">"
          MEMBER_TYPE_NAME = ( "Opt_" , %IDL_TYPE_NAME% )

        Emit1: "typedef" %IDL_TYPE% %IDL_TYPE_NAME% "["
%C_UPPERBOUND_PRODUCT_VALUE% "]" ";"

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"

        Emit2: %MEMBER_TYPE_NAME% %DE_ROLE_NAME% ";"

      Condition: If DE_REF4 holds:

        Let:
          IDL_TYPE = %DE_TYPE_NAME%
          IDL_TYPE_NAME = %IDL_TYPE%
          MEMBER_IDL_TYPE = "sequence" "<" %IDL_TYPE_NAME%  "," "1" ">"
          MEMBER_TYPE_NAME = ( "Opt_" , %IDL_TYPE_NAME% )

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"
```

```
        Emit2: %MEMBER_TYPE_NAME% %DE_ROLE_NAME% ";"

Rule: DirectEquivalentEntityTypeTemplateMethodReferenceStatement

  // This rule was manufactured to facilitate the binding specification.
  // It is not a production rule in the Template grammar specification.
  // Its Expression is a combination of the OptionalAnnotation? Alternate
  // of the StructuredTemplateElementTypeReferenceStatement and the
  // EquivalentEntityTypeTemplateMethodReference-type Alternate of
  // the StructuredTemplateElementTypeReferenceExpression Rules in the grammar.

  Expression: OptionalAnnotation? EquivalentEntityTypeTemplateMethodReference
LEFT_ANGLE_BRACKET StructuredTemplateElementTypeReferenceParameterList
RIGHT_ANGLE_BRACKET StructuredTemplateElementMemberName

    // Generate IDL for each EquivalentEntityTypeTemplateMethodMember of the
    // StructuredTemplateElementTypeDecl referenced by the
    // EquivalentEntityTypeTemplateMethodReference.

    Emit1: { %EquivalentEntityTypeTemplateMethodMember% =>
<GenerateEquivalentEntityTypeTemplateMethodMemberStatementIDL> }+

    // Instantiate the IDL Template Module. Note that the IDL Template Module
instance is
    // specific to the enclosing StructuredTemplateElementType.

    Emit1: "module" ( %<EquivalentEntityTypeTemplateMethodReference>% ,
"_Module" )
    Emit1: "<"

    For each EquivalentEntityTypeTemplateMethodMember of the
StructuredTemplateElementTypeDecl referenced by the
EquivalentEntityTypeTemplateMethodReference:

      Let:
        MEMBER_IDLTYPE = { %EquivalentEntityTypeTemplateMethodMember% =>
<GetInstantiatedEquivalentEntityTypeTemplateMethodMemberStatementIDLType> }

      Condition: If %EquivalentEntityTypeTemplateMethodMember% is the first
EquivalentEntityTypeTemplateMethodMember of the
StructuredTemplateElementTypeDecl referenced by the
EquivalentEntityTypeTemplateMethodReference:

        Emit1: %MEMBER_IDLTYPE%

      Condition: If %EquivalentEntityTypeTemplateMethodMember% is not the first
EquivalentEntityTypeTemplateMethodMember of the
StructuredTemplateElementTypeDecl referenced by the
EquivalentEntityTypeTemplateMethodReference:

        Emit1: "," %MEMBER_IDLTYPE%

    // Close the IDL Template Module instantiation's actual parameter list.

    Emit1: ">"

    // Construct a name for this IDL Template Module instance:

    Let:
      UNDERSCORE_SEPARATED_ENTITY_NAME_LIST = ""

    For each StructuredTemplateElementTypeReferenceParameter in the
StructuredTemplateElementTypeReferenceParameterList:
```

```
      // Get the name of the DesignatedEntityTypeReference behind this
      // StructuredTemplateElementTypeReferenceParameter.

      Let:
        PARAMETERS_ENTITY_NAME =
%%<StructuredTemplateElementTypeReferenceParameter>%.name%

      // Append the DesignatedEntityTypeReference's name.

      UNDERSCORE_SEPARATED_ENTITY_NAME_LIST = (
%UNDERSCORE_SEPARATED_ENTITY_NAME_LIST% , "_" , %PARAMETERS_ENTITY_NAME% )

    Let:
      IDL_TYPE_NAME =  ( %CURRENT_STRUCTURED_TEMPLATE_ELEMENT_NAME% ,
%PARAMETER_LIST_ENTITY_NAMES% , "_Resource" )
      IDL_TEMPLATE_MODULE_INST_NAME = ( %IDL_TYPE_NAME% , "_Module" )

    // Finalize the IDL Template Module instantiation.

    Emit1: %IDL_TEMPLATE_MODULE_INST_NAME% ";"

    Emit1: "typedef" ( %IDL_TEMPLATE_MODULE_INST_NAME% , "::" ,
%<EquivalentEntityTypeTemplateMethodReference>% ) %IDL_TYPE_NAME% ";"

    // Emit the IDL member for this EquivalentEntityTypeTemplateMethodReference
type of
    // StructuredTemplateElementTypeReferenceStatement.

    Emit2: %IDL_TYPE_NAME% %<StructuredTemplateElementMemberName>% ";"

Rule: GenerateEquivalentEntityTypeTemplateMethodMemberStatementIDL

  // This rule was manufactured to facilitate the binding specification.
  // It is not a production rule in the Template grammar specification.
  // Its Expression is the EquivalentEntityTypeTemplateMethodMember's Rule
  // in the grammar.

  Expression: EquivalentEntityTypeTemplateElementMemberStatement SEMICOLON

    Let:
      INSTANTIATED_MEMBER_STATEMENT = {
%EquivalentEntityTypeTemplateElementMemberStatement% =>
<GetInstantiatedEquivalentEntityTypeTemplateElementMemberStatement> }

    Emit1: { %INSTANTIATED_MEMBER_STATEMENT% =>
<GenerateExplicitDesignatedEntityNonEntityTypeCharacteristicReferenceIDL> }

Rule: GetInstantiatedEquivalentEntityTypeTemplateMethodMemberStatementIDLType

  // This rule was manufactured to facilitate the binding specification.
  // It is not a production rule in the Template grammar specification.
  // Its Expression is the EquivalentEntityTypeTemplateMethodMember Rule's
  // Expression in the grammar.

  Expression: EquivalentEntityTypeTemplateElementMemberStatement SEMICOLON

    Let:
      INSTANTIATED_MEMBER_STATEMENT = {
%EquivalentEntityTypeTemplateElementMemberStatement% =>
<GetInstantiatedEquivalentEntityTypeTemplateElementMemberStatement> }
      INSTANTIATED_MEMBER_STATEMENT_IDLTYPE = { %INSTANTIATED_MEMBER_STATEMENT%
=>
```

```
<GetExplicitDesignatedEntityNonEntityTypeCharacteristicReferenceMemberIDLType>
}

    Return: %INSTANTIATED_MEMBER_STATEMENT_IDLTYPE%

Rule: GenerateExplicitDesignatedEntityNonEntityTypeCharacteristicReferenceIDL

  // This rule was manufactured to facilitate the binding specification. It is
not
  // a production rule in the Template grammar specification. Its Expression is
the
  // ExplicitDesignatedEntityNonEntityTypeCharacteristicReferenceExpression
Rule's
  // Expression in the grammar.

  Expression: OptionalAnnotation?
DesignatedEntityNonEntityTypeCharacteristicReference ( DEREF
IDLStructMemberReference )? ( KW_AS StructuredTemplateElementMemberName )?

    // Behind each
ExplicitDesignatedEntityNonEntityTypeCharacteristicReferenceExpression
    // is a DesignatedEntityNonEntityTypeCharacteristicReference, which is a
reference to
    // a Characteristic composed into a DesignatedEntityTypeReference, which is
a reference
    // to a datamodel.platform.Entity (the Entity that composes that
Characteristic.
    // A DesignatedEntityNonEntityTypeCharacteristicReference's type is a
always
    // datamodel.platform.PlatformDataType.

    // Emit the IDL for the
DesignatedEntityNonEntityTypeCharacteristicReference's type.

    Emit1: { %%<DesignatedEntityNonEntityTypeCharacteristicReference>%.type% =>
<PlatformIDLType> }

    // The IDL type emitted for an
    // ExplicitDesignatedEntityNonEntityTypeCharacteristicReferenceExpression
may be
    // a complex type based of the
    // DesignatedEntityNonEntityTypeCharacteristicReference's IDL type, such as
an
    // array or sequence.

    // The C_MULTI and C_REF conditional rules below refer to Characteristics
that
    // join the JoinPathEntityTypeReferences along the
EntityTypeReferenceJoinPath
    // from the PrimaryEntityTypeTemplateMethodParameter's EntityTypeReference
of
    // %CURRENT_STRUCTURED_TEMPLATE_ELEMENT% to the
    // DesignatedEntityNonEntityTypeCharacteristicReference's composing
    // DesignatedEntityTypeReference. An example EntityTypeReferenceJoinPath is
    // A.B.C, where A, B, and C are JoinPathEntityTypeReferences. Each L.R pair
    // in an EntityTypeReferenceJoinPath represents and is unambiguously
traceable to
    // a Entity join in %%CURRENT_TEMPLATE%.boundQuery%. L and R represent the
    // Entitys specified in that join, and the "." represents the
Characteristic used
    // to join them. In the example, there are 2 joins represented: the first
is A.B
```

```
    // and the second is B.C. There are 2 joining Characteristics, one for each
join:
    // one that joins A and B, and the other that joins B and C.

    // The C_MULTI condition determines the multiplicity of a joining
Characteristic
    // for the subsequent C_REF conditional rules. In the C_MULTI condition, L
    // represents the Characteristic's left JoinPathEntityTypeReference (the
Entity
    // to the left of a "."), and R represents the Characteristic's right
    // JoinPathEntityTypeReference (the Entity to the right of that ").

    C_MULTI Conditional Rule:
      Let:
        JOIN_CHARACTERISTIC = datamodel.platform.Characteristic  // a "."

      If %JOIN_CHARACTERISTIC% is a Composition or Participant of L, then:

        If %JOIN_CHARACTERISTIC% is not in the EntityTypeReferenceJoinPath
between the MainEntityTypeTemplateMethodDecl and the current UnionTypeDecl or
SupportingEntityTypeTemplateMethodDecl, then:

          Let:
            C_MULTI_UPPER_BOUND = %JOIN_CHARACTERISTIC%.upperBound
            C_MULTI_LOWER_BOUND = %JOIN_CHARACTERISTIC%.lowerBound

        Otherwise:

          Let:
            C_MULTI_UPPER_BOUND = 1
            C_MULTI_LOWER_BOUND = 1

        If R is a SelectedEntity that is a SelectedEntityReference of a
SelectedEntityCharacteristicReference CharacteristicBasis in
%%CURRENT_TEMPLATE%.boundQuery%, then:

          Let:
            C_MULTI_LOWER_BOUND = 0

      Otherwise: // %JOIN_CHARACTERISTIC% is a Composition or Participant of R:

        If %JOIN_CHARACTERISTIC% is not in the EntityTypeReferenceJoinPath
between the MainEntityTypeTemplateMethodDecl and the current UnionTypeDecl or
SupportingEntityTypeTemplateMethodDecl, then:

          If %JOIN_CHARACTERISTIC% is a Composition, then:

            Let:
              C_MULTI_UPPER_BOUND = 1
              C_MULTI_LOWER_BOUND = 1

          Otherwise: // %JOIN_CHARACTERISTIC% is a Participant

            Let:
              C_MULTI_UPPER_BOUND = %JOIN_CHARACTERISTIC%.sourceUpperBound
              C_MULTI_LOWER_BOUND = %JOIN_CHARACTERISTIC%.sourceLowerBound

        Otherwise:

            Let:
              C_MULTI_UPPER_BOUND = 1
              C_MULTI_LOWER_BOUND = 1
```

```
        If L is a SelectedEntity that is a SelectedEntityReference of a
SelectedEntityCharacteristicReference CharacteristicBasis in
%%CURRENT_TEMPLATE%.boundQuery%:

          Let:
            C_MULTI_LOWER_BOUND = 0

    The following 4 emit conditional rules are used to determine the IDL type
for a DesignatedEntityNonEntityTypeCharacteristicReference's type. These rules
use the C_MULTI_LOWER_BOUND and C_MULTI_UPPER_BOUND determined by the C_MULTI
conditional rule above for each joining Characteristic:

      Emit Conditional Rule C_REF1:
        If C_MULTI_UPPER_BOUND of any of the joining Characteristics is
unbounded, or the DesignatedEntityNonEntityTypeCharacteristicReference's
upperBound is unbounded, then the type emitted is an unbounded sequence.

      Emit Conditional Rule C_REF2:
        If C_REF1 does not hold and C_MULTI_LOWER_BOUND <> C_MULTI_UPPER_BOUND
for any of the joining Characteristics, or
%%DesignatedEntityNonEntityTypeCharacteristicReference%.lowerBound% <>
%%DesignatedEntityNonEntityTypeCharacteristicReference%.upperBound%, then the
type emitted is a bounded sequence whose bound is the product of all of the
joining Characteristic's C_MULTI_UPPER_BOUND and
%%DesignatedEntityNonEntityTypeCharacteristicReference%.upperBound%.

        Let: C_UPPERBOUND_PRODUCT_VALUE = // the calculated product

      Emit Conditional Rule C_REF3:
        If C_REF1 and C_REF2 do not hold, and C_MULTI_UPPER_BOUND > 1 and
C_MULTI_LOWER_BOUND = C_MULTI_UPPER_BOUND for any of the joining
Characteristics, or
%%DesignatedEntityNonEntityTypeCharacteristicReference%.upperBound% > 1 and
%%DesignatedEntityNonEntityTypeCharacteristicReference%.lowerBound% =
%%DesignatedEntityNonEntityTypeCharacteristicReference%.upperBound%, then the
type emitted is an array whose size is the product of all joining
Characteristic's C_MULTI_UPPER_BOUND and
%%DesignatedEntityNonEntityTypeCharacteristicReference%.upperBound%.

        Let: C_UPPERBOUND_PRODUCT_VALUE = // the calculated product

      Emit Conditional Rule C_REF4:
        If C_REF1, C_REF2, and C_REF3 do not hold, then C_MULTI_UPPER_BOUND = 1
and C_MULTI_LOWER_BOUND = 1 for all of the joining Characteristics, and
%%DesignatedEntityNonEntityTypeCharacteristicReference%.upperBound% = 1 and its
%%DesignatedEntityNonEntityTypeCharacteristicReference%.lowerBound% = 1, and
the type emitted is unary.

    Variant: DesignatedEntityNonEntityTypeCharacteristicReference

      Let:
        C_TYPE_NAME =
%<DesignatedEntityNonEntityTypeCharacteristicReference>%.type.name
        C_ROLE_NAME =
%<DesignatedEntityNonEntityTypeCharacteristicReference>%.rolename

      Condition: If C_REF1 holds:

        Let:
          MEMBER_IDL_TYPE = "sequence" "<" %C_TYPE_NAME% ">"
          MEMBER_TYPE_NAME = ( "Seq_" , %C_TYPE_NAME% )

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"
```

```
      Condition: If C_REF2 holds:

        Let:
          MEMBER_IDL_TYPE = "sequence" "<" %C_TYPE_NAME% ","
%C_UPPERBOUND_PRODUCT_VALUE% ">"
          MEMBER_TYPE_NAME = ( "Seq_" , %C_UPPERBOUND_PRODUCT_VALUE% , "_" ,
%C_TYPE_NAME% )

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"

      Condition: If C_REF3 holds:

        Let:
          MEMBER_IDL_TYPE = %C_TYPE_NAME%
          MEMBER_TYPE_NAME = ( "Array_" , %C_UPPERBOUND_PRODUCT_VALUE% , "_" ,
%C_TYPE_NAME% )

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% "["
%C_UPPERBOUND_PRODUCT_VALUE% "]" ";"

      Condition: If C_REF4 holds:

        Let:
          MEMBER_IDL_TYPE = %C_TYPE_NAME%
          MEMBER_TYPE_NAME = %MEMBER_IDL_TYPE%

        Emit1: // Intentionally blank - emit nothing

    Variant: DesignatedEntityNonEntityTypeCharacteristicReference KW_AS
StructuredTemplateElementMemberName

      Let:
        C_TYPE_NAME =
%<DesignatedEntityNonEntityTypeCharacteristicReference>%.type.name
        C_ROLE_NAME = <StructuredTemplateElementMemberName>

      Condition: If C_REF1 holds:

        Let:
          MEMBER_IDL_TYPE = "sequence" "<" %C_TYPE_NAME% ">"
          MEMBER_TYPE_NAME = ( "Seq_" , %C_TYPE_NAME% )

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"

      Condition: If C_REF2 holds:

        Let:
          MEMBER_IDL_TYPE = "sequence" "<" %C_TYPE_NAME% ","
%C_UPPERBOUND_PRODUCT_VALUE% ">"
          MEMBER_TYPE_NAME = ( "Seq_" , %C_UPPERBOUND_PRODUCT_VALUE% , "_" ,
%C_TYPE_NAME% )

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"

      Condition: If C_REF3 holds:

        Let:
          MEMBER_IDL_TYPE = %C_TYPE_NAME%
          MEMBER_TYPE_NAME = ( "Array_" , %C_UPPERBOUND_PRODUCT_VALUE% , "_" ,
%C_TYPE_NAME% )
```

```
        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% "["
%C_UPPERBOUND_PRODUCT_VALUE% "]" ";"

      Condition: If C_REF4 holds:

        Let:
          MEMBER_IDL_TYPE = %C_TYPE_NAME%
          MEMBER_TYPE_NAME = %MEMBER_IDL_TYPE%

        Emit1: // Intentionally blank - emit nothing

    Variant: OptionalAnnotation
DesignatedEntityNonEntityTypeCharacteristicReference

      Let:
        C_TYPE_NAME =
%<DesignatedEntityNonEntityTypeCharacteristicReference>%.type.name
        C_ROLE_NAME =
%<DesignatedEntityNonEntityTypeCharacteristicReference>%.rolename

      Condition: If C_REF1 holds:

        Let:
          IDL_TYPE = "sequence" "<" %C_TYPE_NAME% ">"
          IDL_TYPE_NAME = ( "Seq_" , %C_TYPE_NAME% )
          MEMBER_IDL_TYPE = "sequence" "<" %IDL_TYPE_NAME%  "," "1" ">"
          MEMBER_TYPE_NAME = ( "Opt_" , %IDL_TYPE_NAME% )

        Emit1: "typedef" %IDL_TYPE% %IDL_TYPE_NAME% ";"

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"

      Condition: If C_REF2 holds:

        Let:
          IDL_TYPE = "sequence" "<" %C_TYPE_NAME% ","
%C_UPPERBOUND_PRODUCT_VALUE% ">"
          IDL_TYPE_NAME = ( "Seq_" , %C_UPPERBOUND_PRODUCT_VALUE% , "_" ,
%C_TYPE_NAME% )
          MEMBER_IDL_TYPE = "sequence" "<" %IDL_TYPE_NAME%  "," "1" ">"
          MEMBER_TYPE_NAME = ( "Opt_" , %IDL_TYPE_NAME% )

        Emit1: "typedef" %IDL_TYPE% %IDL_TYPE_NAME% ";"

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"

      Condition: If C_REF3 holds:

        Let:
          IDL_TYPE = %C_TYPE_NAME%
          IDL_TYPE_NAME = ( "Array_" , %C_UPPERBOUND_PRODUCT_VALUE% , "_" ,
%C_TYPE_NAME% )
          MEMBER_IDL_TYPE = "sequence" "<" %IDL_TYPE_NAME%  "," "1" ">"
          MEMBER_TYPE_NAME = ( "Opt_" , %IDL_TYPE_NAME% )

        Emit1: "typedef" %IDL_TYPE% %IDL_TYPE_NAME% "["
%C_UPPERBOUND_PRODUCT_VALUE% "]" ";"

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"

      Condition: If C_REF4 holds:

        Let:
```

```
            IDL_TYPE = %C_TYPE_NAME%
            IDL_TYPE_NAME = %IDL_TYPE%
            MEMBER_IDL_TYPE = "sequence" "<" %IDL_TYPE_NAME%  "," "1" ">"
            MEMBER_TYPE_NAME = ( "Opt_" , %IDL_TYPE_NAME% )

          Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"

      Variant: OptionalAnnotation
DesignatedEntityNonEntityTypeCharacteristicReference KW_AS
StructuredTemplateElementMemberName

        Let:
          C_TYPE_NAME =
%<DesignatedEntityNonEntityTypeCharacteristicReference>%.type.name
          C_ROLE_NAME = <StructuredTemplateElementMemberName>

        Condition: If C_REF1 holds:

          Let:
            IDL_TYPE = "sequence" "<" %C_TYPE_NAME% ">"
            IDL_TYPE_NAME = ( "Seq_" , %C_TYPE_NAME% )
            MEMBER_IDL_TYPE = "sequence" "<" %IDL_TYPE_NAME%  "," "1" ">"
            MEMBER_TYPE_NAME = ( "Opt_" , %IDL_TYPE_NAME% )

          Emit1: "typedef" %IDL_TYPE% %IDL_TYPE_NAME% ";"

          Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"

        Condition: If C_REF2 holds:

          Let:
            IDL_TYPE = "sequence" "<" %C_TYPE_NAME% ","
%C_UPPERBOUND_PRODUCT_VALUE% ">"
            IDL_TYPE_NAME = ( "Seq_" , %C_UPPERBOUND_PRODUCT_VALUE% , "_" ,
%C_TYPE_NAME% )
            MEMBER_IDL_TYPE = "sequence" "<" %IDL_TYPE_NAME%  "," "1" ">"
            MEMBER_TYPE_NAME = ( "Opt_" , %IDL_TYPE_NAME% )

          Emit1: "typedef" %IDL_TYPE% %IDL_TYPE_NAME% ";"

          Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"

        Condition: If C_REF3 holds:

          Let:
            IDL_TYPE = %C_TYPE_NAME%
            IDL_TYPE_NAME = ( "Array_" , %C_UPPERBOUND_PRODUCT_VALUE% , "_" ,
%C_TYPE_NAME% )
            MEMBER_IDL_TYPE = "sequence" "<" %IDL_TYPE_NAME%  "," "1" ">"
            MEMBER_TYPE_NAME = ( "Opt_" , %IDL_TYPE_NAME% )

          Emit1: "typedef" %IDL_TYPE% %IDL_TYPE_NAME% "["
%C_UPPERBOUND_PRODUCT_VALUE% "]" ";"

          Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"

        Condition: If C_REF4 holds:

          Let:
            IDL_TYPE = %C_TYPE_NAME%
            IDL_TYPE_NAME = %IDL_TYPE%
            MEMBER_IDL_TYPE = "sequence" "<" %IDL_TYPE_NAME%  "," "1" ">"
            MEMBER_TYPE_NAME = ( "Opt_" , %IDL_TYPE_NAME% )
```

```
        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"

    Variant: DesignatedEntityNonEntityTypeCharacteristicReference DEREF
IDLStructMemberReference

      Let:
        C_TYPE_NAME = %<IDLStructMemberReference>%.type.name
        C_ROLE_NAME = %<IDLStructMemberReference>%.rolename

      Condition: If C_REF1 holds:

        Let:
          MEMBER_IDL_TYPE = "sequence" "<" %C_TYPE_NAME% ">"
          MEMBER_TYPE_NAME = ( "Seq_" , %C_TYPE_NAME% )

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"

      Condition: If C_REF2 holds:

        Let:
          MEMBER_IDL_TYPE = "sequence" "<" %C_TYPE_NAME% ","
%C_UPPERBOUND_PRODUCT_VALUE% ">"
          MEMBER_TYPE_NAME = ( "Seq_" , %C_UPPERBOUND_PRODUCT_VALUE% , "_" ,
%C_TYPE_NAME% )

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"

      Condition: If C_REF3 holds:

        Let:
          MEMBER_IDL_TYPE = %C_TYPE_NAME%
          MEMBER_TYPE_NAME = ( "Array_" , %C_UPPERBOUND_PRODUCT_VALUE% , "_" ,
%C_TYPE_NAME% )

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% "["
%C_UPPERBOUND_PRODUCT_VALUE% "]" ";"

      Condition: If C_REF4 holds:

        Let:
          MEMBER_IDL_TYPE = %C_TYPE_NAME%
          MEMBER_TYPE_NAME = %MEMBER_IDL_TYPE%

        Emit1: // Intentionally blank - emit nothing

    Variant: DesignatedEntityNonEntityTypeCharacteristicReference DEREF
IDLStructMemberReference KW_AS StructuredTemplateElementMemberName

      Let:
        C_TYPE_NAME = %<IDLStructMemberReference>%.type.name
        C_ROLE_NAME = <StructuredTemplateElementMemberName>

      Condition: If C_REF1 holds:

        Let:
          MEMBER_IDL_TYPE = "sequence" "<" %C_TYPE_NAME% ">"
          MEMBER_TYPE_NAME = ( "Seq_" , %C_TYPE_NAME% )

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"

      Condition: If C_REF2 holds:
```

```
        Let:
          MEMBER_IDL_TYPE = "sequence" "<" %C_TYPE_NAME% ","
%C_UPPERBOUND_PRODUCT_VALUE% ">"
          MEMBER_TYPE_NAME = ( "Seq_" , %C_UPPERBOUND_PRODUCT_VALUE% , "_" ,
%C_TYPE_NAME% )

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"

      Condition: If C_REF3 holds:

        Let:
          MEMBER_IDL_TYPE = %C_TYPE_NAME%
          MEMBER_TYPE_NAME = ( "Array_" , %C_UPPERBOUND_PRODUCT_VALUE% , "_" ,
%C_TYPE_NAME% )

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% "["
%C_UPPERBOUND_PRODUCT_VALUE% "]" ";"

        Emit1: %MEMBER_TYPE_NAME% %C_ROLE_NAME% ";"

      Condition: If C_REF4 holds:

        Let:
          MEMBER_IDL_TYPE = %C_TYPE_NAME%
          MEMBER_TYPE_NAME = %MEMBER_IDL_TYPE%

        Emit1: // Intentionally blank - emit nothing

    Variant: OptionalAnnotation
DesignatedEntityNonEntityTypeCharacteristicReference DEREF
IDLStructMemberReference

      Let:
        C_TYPE_NAME = %<IDLStructMemberReference>%.type.name
        C_ROLE_NAME = %<IDLStructMemberReference>%.rolename

      Condition: If C_REF1 holds:

        Let:
          IDL_TYPE = "sequence" "<" %C_TYPE_NAME% ">"
          IDL_TYPE_NAME = ( "Seq_" , %C_TYPE_NAME% )
          MEMBER_IDL_TYPE = "sequence" "<" %IDL_TYPE_NAME%  "," "1" ">"
          MEMBER_TYPE_NAME = ( "Opt_" , %IDL_TYPE_NAME% )

        Emit1: "typedef" %IDL_TYPE% %IDL_TYPE_NAME% ";"

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"

      Condition: If C_REF2 holds:

        Let:
          IDL_TYPE = "sequence" "<" %C_TYPE_NAME% ","
%C_UPPERBOUND_PRODUCT_VALUE% ">"
          IDL_TYPE_NAME = ( "Seq_" , %C_UPPERBOUND_PRODUCT_VALUE% , "_" ,
%C_TYPE_NAME% )
          MEMBER_IDL_TYPE = "sequence" "<" %IDL_TYPE_NAME%  "," "1" ">"
          MEMBER_TYPE_NAME = ( "Opt_" , %IDL_TYPE_NAME% )

        Emit1: "typedef" %IDL_TYPE% %IDL_TYPE_NAME% ";"

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"

      Condition: If C_REF3 holds:
```

```
        Let:
          IDL_TYPE = %C_TYPE_NAME%
          IDL_TYPE_NAME = ( "Array_" , %C_UPPERBOUND_PRODUCT_VALUE% , "_" ,
%C_TYPE_NAME% )
          MEMBER_IDL_TYPE = "sequence" "<" %IDL_TYPE_NAME%  "," "1" ">"
          MEMBER_TYPE_NAME = ( "Opt_" , %IDL_TYPE_NAME% )

        Emit1: "typedef" %IDL_TYPE% %IDL_TYPE_NAME% "["
%C_UPPERBOUND_PRODUCT_VALUE% "]" ";"

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"

        Emit1: %MEMBER_TYPE_NAME% %C_ROLE_NAME% ";"

      Condition: If C_REF4 holds:

        Let:
          IDL_TYPE = %C_TYPE_NAME%
          IDL_TYPE_NAME = %IDL_TYPE%
          MEMBER_IDL_TYPE = "sequence" "<" %IDL_TYPE_NAME%  "," "1" ">"
          MEMBER_TYPE_NAME = ( "Opt_" , %IDL_TYPE_NAME% )

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"

    Variant: OptionalAnnotation
DesignatedEntityNonEntityTypeCharacteristicReference DEREF
IDLStructMemberReference KW_AS StructuredTemplateElementMemberName

      Let:
        C_TYPE_NAME = %<IDLStructMemberReference>%.type.name
        C_ROLE_NAME = <StructuredTemplateElementMemberName>

      Condition: If C_REF1 holds:

        Let:
          IDL_TYPE = "sequence" "<" %C_TYPE_NAME% ">"
          IDL_TYPE_NAME = ( "Seq_" , %C_TYPE_NAME% )
          MEMBER_IDL_TYPE = "sequence" "<" %IDL_TYPE_NAME%  "," "1" ">"
          MEMBER_TYPE_NAME = ( "Opt_" , %IDL_TYPE_NAME% )

        Emit1: "typedef" %IDL_TYPE% %IDL_TYPE_NAME% ";"

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"

      Condition: If C_REF2 holds:

        Let:
          IDL_TYPE = "sequence" "<" %C_TYPE_NAME% ","
%C_UPPERBOUND_PRODUCT_VALUE% ">"
          IDL_TYPE_NAME = ( "Seq_" , %C_UPPERBOUND_PRODUCT_VALUE% , "_" ,
%C_TYPE_NAME% )
          MEMBER_IDL_TYPE = "sequence" "<" %IDL_TYPE_NAME%  "," "1" ">"
          MEMBER_TYPE_NAME = ( "Opt_" , %IDL_TYPE_NAME% )

        Emit1: "typedef" %IDL_TYPE% %IDL_TYPE_NAME% ";"

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"

      Condition: If C_REF3 holds:

        Let:
          IDL_TYPE = %C_TYPE_NAME%
```

```
          IDL_TYPE_NAME = ( "Array_" , %C_UPPERBOUND_PRODUCT_VALUE% , "_" ,
%C_TYPE_NAME% )
          MEMBER_IDL_TYPE = "sequence" "<" %IDL_TYPE_NAME%  "," "1" ">"
          MEMBER_TYPE_NAME = ( "Opt_" , %IDL_TYPE_NAME% )

        Emit1: "typedef" %IDL_TYPE% %IDL_TYPE_NAME% "["
%C_UPPERBOUND_PRODUCT_VALUE% "]" ";"

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"

      Condition: If C_REF4 holds:

        Let:
          IDL_TYPE = %C_TYPE_NAME%
          IDL_TYPE_NAME = %IDL_TYPE%
          MEMBER_IDL_TYPE = "sequence" "<" %IDL_TYPE_NAME%  "," "1" ">"
          MEMBER_TYPE_NAME = ( "Opt_" , %IDL_TYPE_NAME% )

        Emit1: "typedef" %MEMBER_IDL_TYPE% %MEMBER_TYPE_NAME% ";"

Rule:
GetExplicitDesignatedEntityNonEntityTypeCharacteristicReferenceMemberIDLType

  // This rule was manufactured to facilitate the binding specification. It is
not
  // a production rule in the Template grammar specification. Its Expression is
the
  // ExplicitDesignatedEntityNonEntityTypeCharacteristicReferenceExpression
Rule's
  // Expression in the grammar.

  Expression: OptionalAnnotation?
DesignatedEntityNonEntityTypeCharacteristicReference ( DEREF
IDLStructMemberReference )? ( KW_AS StructuredTemplateElementMemberName )?

    // The IDL type emitted for an
    // ExplicitDesignatedEntityNonEntityTypeCharacteristicReferenceExpression
may be
    // a complex type based of the
    // DesignatedEntityNonEntityTypeCharacteristicReference's IDL type, such as
an
    // array or sequence.

    // The C_MULTI and C_REF conditional rules below refer to Characteristics
that
    // join the JoinPathEntityTypeReferences along the
EntityTypeReferenceJoinPath
    // from the PrimaryEntityTypeTemplateMethodParameter's EntityTypeReference
of
    // %CURRENT_STRUCTURED_TEMPLATE_ELEMENT% to the
    // DesignatedEntityNonEntityTypeCharacteristicReference's composing
    // DesignatedEntityTypeReference. An example EntityTypeReferenceJoinPath is
    // A.B.C, where A, B, and C are JoinPathEntityTypeReferences. Each L.R pair
    // in an EntityTypeReferenceJoinPath represents and is unambiguously
traceable to
    // a Entity join in %%CURRENT_TEMPLATE%.boundQuery%. L and R represent the
    // Entitys specified in that join, and the "." represents the
Characteristic used
    // to join them. In the example, there are 2 joins represented: the first
is A.B
    // and the second is B.C. There are 2 joining Characteristics, one for each
join:
    // one that joins A and B, and the other that joins B and C.
```

```
    // The C_MULTI condition determines the multiplicity of a joining
Characteristic
    // for the subsequent C_REF conditional rules. In the C_MULTI condition, L
    // represents the Characteristic's left JoinPathEntityTypeReference (the
Entity
    // to the left of a "."), and R represents the Characteristic's right
    // JoinPathEntityTypeReference (the Entity to the right of that ".").

    C_MULTI Conditional Rule:
      Let:
        JOIN_CHARACTERISTIC = datamodel.platform.Characteristic  // a "."

      If %JOIN_CHARACTERISTIC% is a Composition or Participant of L, then:

        If %JOIN_CHARACTERISTIC% is not in the EntityTypeReferenceJoinPath
between the MainEntityTypeTemplateMethodDecl and the current UnionTypeDecl or
SupportingEntityTypeTemplateMethodDecl, then:

          Let:
            C_MULTI_UPPER_BOUND = %JOIN_CHARACTERISTIC%.upperBound
            C_MULTI_LOWER_BOUND = %JOIN_CHARACTERISTIC%.lowerBound

        Otherwise:

          Let:
            C_MULTI_UPPER_BOUND = 1
            C_MULTI_LOWER_BOUND = 1

        If R is a SelectedEntity that is a SelectedEntityReference of a
SelectedEntityCharacteristicReference CharacteristicBasis in
%%CURRENT_TEMPLATE%.boundQuery%, then:

          Let:
            C_MULTI_LOWER_BOUND = 0

      Otherwise: // %JOIN_CHARACTERISTIC% is a Composition or Participant of R:

        If %JOIN_CHARACTERISTIC% is not in the EntityTypeReferenceJoinPath
between the MainEntityTypeTemplateMethodDecl and the current UnionTypeDecl or
SupportingEntityTypeTemplateMethodDecl, then:

          If %JOIN_CHARACTERISTIC% is a Composition, then:

            Let:
              C_MULTI_UPPER_BOUND = 1
              C_MULTI_LOWER_BOUND = 1

          Otherwise: // %JOIN_CHARACTERISTIC% is a Participant

            Let:
              C_MULTI_UPPER_BOUND = %JOIN_CHARACTERISTIC%.sourceUpperBound
              C_MULTI_LOWER_BOUND = %JOIN_CHARACTERISTIC%.sourceLowerBound

        Otherwise:

            Let:
              C_MULTI_UPPER_BOUND = 1
              C_MULTI_LOWER_BOUND = 1

        If L is a SelectedEntity that is a SelectedEntityReference of a
SelectedEntityCharacteristicReference CharacteristicBasis in
%%CURRENT_TEMPLATE%.boundQuery%:
```

```
          Let:
            C_MULTI_LOWER_BOUND = 0

    The following 4 emit conditional rules are used to determine the IDL type
for a DesignatedEntityNonEntityTypeCharacteristicReference's type. These rules
use the C_MULTI_LOWER_BOUND and C_MULTI_UPPER_BOUND determined by the C_MULTI
conditional rule above for each joining Characteristic:

      Emit Conditional Rule C_REF1:
        If C_MULTI_UPPER_BOUND of any of the joining Characteristics is
unbounded, or the DesignatedEntityNonEntityTypeCharacteristicReference's
upperBound is unbounded, then the type emitted is an unbounded sequence.

      Emit Conditional Rule C_REF2:
        If C_REF1 does not hold and C_MULTI_LOWER_BOUND <> C_MULTI_UPPER_BOUND
for any of the joining Characteristics, or
%%DesignatedEntityNonEntityTypeCharacteristicReference%.lowerBound% <>
%%DesignatedEntityNonEntityTypeCharacteristicReference%.upperBound%, then the
type emitted is a bounded sequence whose bound is the product of all of the
joining Characteristic's C_MULTI_UPPER_BOUND and
%%DesignatedEntityNonEntityTypeCharacteristicReference%.upperBound%.

        Let: C_UPPERBOUND_PRODUCT_VALUE = // the calculated product

      Emit Conditional Rule C_REF3:
        If C_REF1 and C_REF2 do not hold, and C_MULTI_UPPER_BOUND > 1 and
C_MULTI_LOWER_BOUND = C_MULTI_UPPER_BOUND for any of the joining
Characteristics, or
%%DesignatedEntityNonEntityTypeCharacteristicReference%.upperBound% > 1 and
%%DesignatedEntityNonEntityTypeCharacteristicReference%.lowerBound% =
%%DesignatedEntityNonEntityTypeCharacteristicReference%.upperBound%, then the
type emitted is an array whose size is the product of all joining
Characteristic's C_MULTI_UPPER_BOUND and
%%DesignatedEntityNonEntityTypeCharacteristicReference%.upperBound%.

        Let: C_UPPERBOUND_PRODUCT_VALUE = // the calculated product

      Emit Conditional Rule C_REF4:
        If C_REF1, C_REF2, and C_REF3 do not hold, then C_MULTI_UPPER_BOUND = 1
and C_MULTI_LOWER_BOUND = 1 for all of the joining Characteristics, and
%%DesignatedEntityNonEntityTypeCharacteristicReference%.upperBound% = 1 and its
%%DesignatedEntityNonEntityTypeCharacteristicReference%.lowerBound% = 1, and
the type emitted is unary.

    Let:
      C_TYPE_NAME =
%<DesignatedEntityNonEntityTypeCharacteristicReference>%.type.name

    Variant: DesignatedEntityNonEntityTypeCharacteristicReference
    Variant: DesignatedEntityNonEntityTypeCharacteristicReference KW_AS
StructuredTemplateElementMemberName
    Variant: DesignatedEntityNonEntityTypeCharacteristicReference DEREF
IDLStructMemberReference
    Variant: DesignatedEntityNonEntityTypeCharacteristicReference DEREF
IDLStructMemberReference KW_AS StructuredTemplateElementMemberName

      Condition: If C_REF1 holds:

        Let:
          MEMBER_TYPE_NAME = ( "Seq_" , %C_TYPE_NAME% )

        Return: %MEMBER_TYPE_NAME%
```

```
      Condition: If C_REF2 holds:

        Let:
          MEMBER_TYPE_NAME = ( "Seq_" , %C_UPPERBOUND_PRODUCT_VALUE% , "_" ,
%C_TYPE_NAME% )

        Return: %MEMBER_TYPE_NAME%

      Condition: If C_REF3 holds:

        Let:
          MEMBER_TYPE_NAME = ( "Array_" , %C_UPPERBOUND_PRODUCT_VALUE% , "_" ,
%C_TYPE_NAME% )

        Return: %MEMBER_TYPE_NAME%

      Condition: If C_REF4 holds:

        Let:
          MEMBER_TYPE_NAME = %C_TYPE_NAME%

        Return: %MEMBER_TYPE_NAME%

    Variant: OptionalAnnotation
DesignatedEntityNonEntityTypeCharacteristicReference
    Variant: OptionalAnnotation
DesignatedEntityNonEntityTypeCharacteristicReference KW_AS
StructuredTemplateElementMemberName
    Variant: OptionalAnnotation
DesignatedEntityNonEntityTypeCharacteristicReference DEREF
IDLStructMemberReference
    Variant: OptionalAnnotation
DesignatedEntityNonEntityTypeCharacteristicReference DEREF
IDLStructMemberReference KW_AS StructuredTemplateElementMemberName

      Condition: If C_REF1 holds:

        Let:
          IDL_TYPE_NAME = ( "Seq_" , %C_TYPE_NAME% )
          MEMBER_TYPE_NAME = ( "Opt_" , %IDL_TYPE_NAME% )

        Return: %MEMBER_TYPE_NAME%

      Condition: If C_REF2 holds:

        Let:
          IDL_TYPE_NAME = ( "Seq_" , %C_UPPERBOUND_PRODUCT_VALUE% , "_" ,
%C_TYPE_NAME% )
          MEMBER_TYPE_NAME = ( "Opt_" , %IDL_TYPE_NAME% )

        Return: %MEMBER_TYPE_NAME%

      Condition: If C_REF3 holds:

        Let:
          IDL_TYPE_NAME = ( "Array_" , %C_UPPERBOUND_PRODUCT_VALUE% , "_" ,
%C_TYPE_NAME% )
          MEMBER_TYPE_NAME = ( "Opt_" , %IDL_TYPE_NAME% )

        Return: %MEMBER_TYPE_NAME%

      Condition: If C_REF4 holds:
```

```
        Let:
          IDL_TYPE_NAME = %C_TYPE_NAME%
          MEMBER_TYPE_NAME = ( "Opt_" , %IDL_TYPE_NAME% )

        Return: %MEMBER_TYPE_NAME%

Rule: StructuredTemplateElementTypeReferenceParameter

  Expression: ( EntityTypeStructuredTemplateElementDeclaredParameterReference
EQUALS )? DesignatedEntityTypeReferencePath

    Return: <DesignatedEntityTypeReferencePath>

Rule: ExternalTemplateTypeReference

  Expression: IDENTIFIER

    Return: face.uop.Template[%IDENTIFIER%]

Rule: ExternalTemplateTypeAlias

  Expression: IDENTIFIER

    Emit1: %IDENTIFIER%

Rule: ExternalStructuredTemplateElementTypeReference

  Expression: IDENTIFIER

    Emit1: %IDENTIFIER%

Rule: ExternalStructuredTemplateElementTypeAlias

  Expression: IDENTIFIER

    Emit1: %IDENTIFIER%

Rule: EntityTypeStructuredTemplateElementTypeReference

  Expression: IDENTIFIER

    Return: IDENTIFIER

Rule: StructuredTemplateElementMemberName

  Expression: IDENTIFIER

    Return: IDENTIFIER

Rule: TemplateElementTypeName

  Expression: IDENTIFIER

    Return: IDENTIFIER

Rule: EquivalentEntityTypeTemplateMethodReference

  Expression: IDENTIFIER

    Return: IDENTIFIER

Rule: DesignatedEquivalentEntityNonEntityTypeCharacteristicReference
```

```
  Expression: EquivalentEntityTypeTemplateMethodParameterReference PERIOD
EquivalentEntityTypeTemplateMethodCharacteristicReference

    Return: <EquivalentEntityTypeTemplateMethodCharacteristicReference>

Rule: EquivalentEntityTypeTemplateMethodCharacteristicReference

  Expression: IDENTIFIER

    Return: IDENTIFIER

Rule: DesignatedEntityNonEntityTypeCharacteristicReference

  Alternate: DesignatedEntityTypeReferencePath PERIOD
QueryProjectedNonEntityTypeCharacteristicReference

    Return:
datamodel.platform.Composition[%<QueryProjectedNonEntityTypeCharacteristicRefer
ence>%]

  Alternate: QueryProjectedNonEntityTypeCharacteristicReferenceOrAlias

    Return:
datamodel.platform.Composition[%<QueryProjectedNonEntityTypeCharacteristicRefer
enceOrAlias>%]

Rule: DesignatedEntityEnumerationTypeCharacteristicReference

  Alternate: DesignatedEntityTypeReferencePath PERIOD
QueryProjectedEnumerationTypeCharacteristicReference

    Return:
datamodel.platform.Composition[%<QueryProjectedEnumerationTypeCharacteristicRef
erence>%]

  Alternate: QueryProjectedEnumerationTypeCharacteristicReferenceOrAlias

    Return:
datamodel.platform.Composition[%<QueryProjectedEnumerationTypeCharacteristicRef
erenceOrAlias>%]

Rule: DesignatedEntityTypeReferencePath

  Expression: ExplicitEntityTypeReferenceJoinPath?
DesignatedEntityTypeReference

    Return: <DesignatedEntityTypeReference>

Rule: DesignatedEntityTypeReference

  Expression: QualifiedEntityTypeReference

    Return: <QualifiedEntityTypeReference>

Rule: QualifiedEntityTypeReference

  Expression: EntityTypeReference EntityCharacteristicValueQualifier?

    Return: <EntityTypeReference>

Rule: EntityTypeReference
```

```
  Expression: QuerySelectedEntityTypeReferenceOrAlias

    Return:
datamodel.platform.Entity[%<QuerySelectedEntityTypeReferenceOrAlias>%]

Rule: IDLStructMemberReference

  Expression: IDENTIFIER

    Return: datamodel.platform.StructMember[%IDENTIFIER%]

Rule: EnumLiteralReferenceExpression

  Expression: LEFT_BRACE EnumerationTypeReference COLON
EnumerationLiteralReference RIGHT_BRACE

    Return: %<EnumerationLiteralReference>%.name

Rule: EnumerationLiteralReference

  Expression: IDENTIFIER

    Return: datamodel.logical.EnumerationLabel[%IDENTIFIER%]

Rule: QueryProjectedNonEntityTypeCharacteristicReferenceOrAlias

  Expression: IDENTIFIER

    Return: IDENTIFIER

Rule: QueryProjectedNonEntityTypeCharacteristicReference

  Expression: IDENTIFIER

    Return: IDENTIFIER

Rule: QueryProjectedEnumerationTypeCharacteristicReferenceOrAlias

  Expression: IDENTIFIER

    Return: IDENTIFIER

Rule: QueryProjectedEnumerationTypeCharacteristicReference

  Expression: IDENTIFIER

    Return: IDENTIFIER

Rule: QuerySelectedEntityTypeReferenceOrAlias

  Expression: IDENTIFIER

    Return: IDENTIFIER

// The following Rule specifies the IDL mappings for
datamodel.platform.PlatformDataType.

Rule: PlatformIDLType

  // This rule was manufactured to facilitate the binding specification.
  // It is not a production rule in the Template grammar specification.
  // Its Expression is not in the Template grammar, but is instead a
  // meta-type in the UDDL meta-model.
```

```
  Expression: datamodel.platform.PlatformDataType

    Let:
      PLATFORM_IDLTYPE = THIS
      PLATFORM_IDLTYPE_NAME = %PLATFORM_IDLTYPE%.name

    Condition: If %PLATFORM_IDLTYPE% is a datamodel.platform.Boolean:

      Emit1: "typedef" "boolean" %PLATFORM_IDLTYPE_NAME% ";"

    Condition: If %PLATFORM_IDLTYPE% is a datamodel.platform.Octet:

      Emit1: "typedef" "octet" %PLATFORM_IDLTYPE_NAME% ";"

    Condition: If %PLATFORM_IDLTYPE% is a datamodel.platform.Char:

      Emit1: "typedef" "char" %PLATFORM_IDLTYPE_NAME% ";"

    Condition: If %PLATFORM_IDLTYPE% is a datamodel.platform.CharArray:

      Emit1: "typedef" "char" %PLATFORM_IDLTYPE_NAME% "["
%%PLATFORM_IDLTYPE%.length% "]" ";"

    Condition: If %PLATFORM_IDLTYPE% is a datamodel.platform.String:

      Emit1: "typedef" "string" %PLATFORM_IDLTYPE_NAME% ";"

    Condition: If %PLATFORM_IDLTYPE% is a datamodel.platform.BoundedString:

      Emit1: "typedef" "string" "<" %%PLATFORM_IDLTYPE%.maxLength% ">"
%PLATFORM_IDLTYPE_NAME% ";"

    Condition: If %PLATFORM_IDLTYPE% is a datamodel.platform.Short:

      Emit1: "typedef" "short" %PLATFORM_IDLTYPE_NAME% ";"

    Condition: If %PLATFORM_IDLTYPE% is a datamodel.platform.Long:

      Emit1: "typedef" "long" %PLATFORM_IDLTYPE_NAME% ";"

    Condition: If %PLATFORM_IDLTYPE% is a datamodel.platform.LongLong:

      Emit1: "typedef" "long" "long" %PLATFORM_IDLTYPE_NAME% ";"

    Condition: If %PLATFORM_IDLTYPE% is a datamodel.platform.UShort:

      Emit1: "typedef" "unsigned" "short" %PLATFORM_IDLTYPE_NAME% ";"

    Condition: If %PLATFORM_IDLTYPE% is a datamodel.platform.ULong:

      Emit1: "typedef" "unsigned" "long" %PLATFORM_IDLTYPE_NAME% ";"

    Condition: If %PLATFORM_IDLTYPE% is a datamodel.platform.ULongLong:

      Emit1: "typedef" "unsigned" "long" "long" %PLATFORM_IDLTYPE_NAME% ";"

    Condition: If %PLATFORM_IDLTYPE% is a datamodel.platform.Float:

      Emit1: "typedef" "float" %PLATFORM_IDLTYPE_NAME% ";"

    Condition: If %PLATFORM_IDLTYPE% is a datamodel.platform.Double:
```

```
      Emit1: "typedef" "double" %PLATFORM_IDLTYPE_NAME% ";"

    Condition: If %PLATFORM_IDLTYPE% is a datamodel.platform.LongDouble:

      Emit1: "typedef" "long" "double" %PLATFORM_IDLTYPE_NAME% ";"

    Condition: If %PLATFORM_IDLTYPE% is a datamodel.platform.Fixed:

      Emit1: "typedef" "fixed" "<" %%PLATFORM_IDLTYPE%.digits%,
%%PLATFORM_IDLTYPE%.scale% ">" %PLATFORM_IDLTYPE_NAME% ";"

    Condition: If %PLATFORM_IDLTYPE% is a datamodel.platform.Sequence:

      Emit1: "typedef" "sequence" "<" "octet" "," %%PLATFORM_IDLTYPE%.maxSize%
">" %PLATFORM_IDLTYPE_NAME% ";"

    Condition: If %PLATFORM_IDLTYPE% is a datamodel.platform.Array:

      Emit1: "typedef" "octet" %PLATFORM_IDLTYPE_NAME% "["
%%PLATFORM_IDLTYPE%.size% "]" ";"

    Condition: If %PLATFORM_IDLTYPE% is a datamodel.platform.Enumeration:

      Let:
        VTU = %PLATFORM_IDLTYPE%.realizes.measurementAxis.valueTypeUnit

      Condition: If %%VTU%.constraint% is not the empty set (i.e. is
specified):

        // constaint is datamodel.logical.EnumerationConstraint

        Emit1: { %%VTU%.constraint% => <LogicalEnumerationConstraint> }

      Condition: If %%VTU%.constraint% is the empty set (i.e. not specified):

        // valueType is a datamodel.logical.Enumerated

        Emit1: { %%VTU%.valueType% => <LogicalEnumerated> }

    Condition: If %PLATFORM_IDLTYPE% is a datamodel.platform.Struct:

      // Generate IDL types for its composed member's types

      For each datamodel.platform.StructMember in
%%PLATFORM_IDLTYPE%.composition%:

        Let:
          MEMBER = datamodel.platform.StructMember

        Emit1: { %%MEMBER%.type% => <PlatformIDLType> }

      // Now generate an IDL Struct for this Struct

      Emit1: "struct" %PLATFORM_IDLTYPE_NAME% "{"

      // Generate the Struct's members

      For each datamodel.platform.StructMember in
%%PLATFORM_IDLTYPE%.composition%:

        Let:
          MEMBER = datamodel.platform.StructMember
```

```
        Emit1: %%MEMBER%.type.name% %%MEMBER%.rolename% ";"

      Emit1: "}" ";"

Rule: LogicalEnumerated

  // This rule was manufactured to facilitate the binding specification.
  // It is not a production rule in the Template grammar specification.
  // Its Expression is not in the Template grammar, but is instead a
  // meta-type in the UDDL meta-model.

  Expression: datamodel.logical.Enumerated

    Let:
      ENUM = THIS

    For each datamodel.logical.EnumerationLabel in %%ENUM%.label%

      Let:
        LABEL = datamodel.logical.EnumerationLabel

      Condition: If %LABEL% is the first EnumerationLabel in %%ENUM%.label%:

        Emit1: %%LABEL%.name%

      Condition: If %LABEL% is not the first EnumerationLabel in
%%ENUM%.label%:

        Emit1: "," %%LABEL%.name%

Rule: LogicalEnumerationConstraint

  // This rule was manufactured to facilitate the binding specification.
  // It is not a production rule in the Template grammar specification.
  // Its Expression is not in the Template grammar, but is instead a
  // meta-type in the UDDL meta-model.

  Expression: datamodel.logical.EnumerationConstraint

    Let:
      CONSTRAINT = THIS

    For each datamodel.logical.EnumerationLabel in %%CONSTRAINT%.allowedValue%

      Let:
        LABEL = datamodel.logical.EnumerationLabel

      Condition: If %LABEL% is the first EnumerationLabel in
%%CONSTRAINT%.allowedValue%:

        Emit1: %%LABEL%.name%

      Condition: If %LABEL% is not the first EnumerationLabel in
%%CONSTRAINT%.allowedValue%:

        Emit1: "," %%LABEL%.name%
```

K Supporting Constructs for IDL to Programming Language Mappings

K.1 C Programming Language

K.1.1 Basic Type Mapping

```
//! @file FACE/types.h
//! @brief Definitions of C types for IDL basic types to C mapping
//! @details This file contains editable type definitions for C types that
//! align with the size and range requirements given in the IDL basic types
//! to C mapping. Because C types' sizes and ranges are platform-dependent,
//! implementations are responsible for supplying full type definitions.

#ifndef _FACE_TYPES_H
#define  FACE TYPES H

typedef EDITME FACE_short;
typedef EDITME FACE_long;
typedef EDITME FACE_long_long;
typedef EDITME FACE_unsigned_short;
typedef EDITME FACE_unsigned_long;
typedef EDITME FACE_unsigned_long_long;
typedef EDITME FACE_float;
typedef EDITME FACE_double;
typedef EDITME FACE_long_double;
typedef EDITME FACE_char;
typedef EDITME FACE_boolean;
typedef EDITME FACE_octet;

#endif /* _FACE_TYPES_H */
```

K.1.2 FACE_interface_return Specification

```
//! @file FACE/interface.h
//! @brief Definition of FACE interface return.

#ifndef  FACE INTERFACE H
#define _FACE_INTERFACE_H

/** @brief Return codes used to report certain runtime errors for
    FACE Standardized Interfaces. */
typedef enum FACE_interface_return {
  FACE_INTERFACE_NO_ERROR,            /**< No error has occurred. */
  FACE_INTERFACE_INSUFFICIENT_MEMORY, /**< (ctor only) An implementation is
                                           unable to allocate enough memory
                                           for initialization. */
  FACE_INTERFACE_NULL_THIS,           /**< The "this_obj" parameter is a
                                           NULL pointer */
  FACE_INTERFACE_NULL_PARAM,          /**< One or more other parameters is a
                                           NULL pointer */
} FACE_interface_return;

#endif //_FACE_INTERFACE_H
```

K.1.3 FACE_sequence Specification

```
//! @file FACE/sequence.h
//! @brief Interface for operating on a generic sequence of elements.
```

```
#ifndef _FACE_SEQUENCE_H
#define _FACE_SEQUENCE_H

#include <FACE/types.h>
#include <limits.h>
#include <stddef.h>

/**
 * @brief Interface for operating on a generic sequence of elements.
 * @details A FACE_sequence is defined by three characteristics:
 *  - length - the current number of elements in the FACE_sequence
 *  - element size - the size of each element
 *  - bound - the maximum number of elements the FACE_sequence can ever
 *            hold. This bound is logical, and is independent from the size
 *            of any underlying memory. A FACE_sequence's bound is fixed
 *            throughout the lifetime of the FACE_sequence. An "unbounded"
 *            FACE_sequence has an infinite bound, represented by
 *            FACE_SEQUENCE_UNBOUNDED_SENTINEL.
 *  - capacity - the number of elements a FACE_sequence has currently
 *               allocated memory for. This may vary by implementation, but
 *               length <= capacity <= bound is always true.
 *
 * A "managed" FACE_sequence is responsible for and manages the lifetime of
 * the memory for the data it represents. An "unmanaged" FACE_sequence
 * essentially wraps a pointer to memory whose lifetime is managed
 * elsewhere.
 *
 * A FACE_sequence is "initialized" if it is in a state that could have
 * resulted from successful initialization by one of the "_init" functions.
 * Any other state makes the FACE_sequence "uninitialized".
 *
 * When a memory allocation failure or precondition violation occurs, a
 * FACE_sequence is put into a known "invalid state". In this invalid
 * state:
 *  - length, capacity, and bound are 0
 *  - FACE_sequence_buffer() will return NULL
 *  - FACE_sequence_is_managed() and FACE_sequence_is_bounded() will
 *    return FALSE
 * The FACE_sequence_is_valid() function indicates whether or not a
 * FACE_sequence is in this state.
 *
 * Global preconditions:
 * - In every function, if the @p this_obj parameter is NULL, the function
 *   does nothing and returns FACE_SEQUENCE_NULL_THIS.
 * - In every _init function, if this_obj is already initialized,
 *   FACE_SEQUENCE_PRECONDITION_VIOLATED is returned and the state of
 *   this_obj is not modified.
 * - In every non _init function, if this_obj has not been initialized,
 *   FACE_SEQUENCE_PRECONDITION_VIOLATED is returned and the state of
 *   this_obj is not modified.
 */
typedef struct {
  /* implementation-specific */
} FACE_sequence;

/** @brief Return codes used to report certain runtime errors. */
typedef enum FACE_sequence_return {
 FACE_SEQUENCE_NO_ERROR,              /**< No error has occurred. */
 FACE_SEQUENCE_INSUFFICIENT_BOUND,    /**< Executing a function would cause
                                           a FACE_sequence's length to
                                           exceed its bound. */
 FACE_SEQUENCE_INSUFFICIENT_MEMORY,   /**< A FACE_sequence is unable to
```

```
                                        allocate enough memory to
                                        perform some function. */
 FACE_SEQUENCE_PRECONDITION_VIOLATED, /**< A precondition of some function
                                        has been violated. */
 FACE_SEQUENCE_NULL_THIS,             /**< The "this_obj" parameter is a
                                        NULL pointer */
 FACE_SEQUENCE_NULL_PARAM,            /**< One or more other parameters is
                                        a NULL pointer */
 FACE_SEQUENCE_INVALID_PARAM          /**< A FACE_sequence parameter is
                                        invalid. */
} FACE_sequence_return;

/** @brief Value representing the bound of an unbounded FACE_sequence. */
#define FACE_SEQUENCE_UNBOUNDED_SENTINEL ((FACE_unsigned_long) UINT_MAX)

/**
 * @brief Managed unbounded initialization - initializes empty managed
 * unbounded FACE_sequence
 * @details (see #FACE_string_init_managed_unbounded)
 *
 * After initialization, FACE_sequence_buffer() will return NULL.
 *
 * @param this_obj the FACE_sequence to be initialized
 * @param sizeof_T the size of each element in @p this_obj
 */
FACE_sequence_return FACE_sequence_init_managed_unbounded(
  FACE_sequence* this_obj,
  size_t         sizeof_T
);

/**
 * @brief Managed bounded initialization - initializes empty managed
 * FACE_sequence of specified bound
 * @details (see #FACE_string_init_managed_bounded)
 *
 * If allocation is successful, FACE_sequence_buffer() will return NULL.
 *
 * @param this_obj the FACE_sequence to be initialized
 * @param sizeof_T the size of each element in @p this_obj
 * @param bound the specified bound for @p this_obj to be initialized with
 */
FACE_sequence_return FACE_sequence_init_managed_bounded(
  FACE_sequence*     this_obj,
  size_t             sizeof_T,
  FACE_unsigned_long bound
);

/**
 * @brief Managed copy initialization
 * @details (see #FACE_string_init_managed_copy)
 */
FACE_sequence_return FACE_sequence_init_managed_copy(
  FACE_sequence* this_obj,
  FACE_sequence* src
);

/**
 * @brief Managed array initialization
 * @details After initialization, this FACE_sequence manages its own data,
 * which is a copy of the @p length elements of size @p sizeof_T in the
 * array pointed to by @p arr, and the bound of @p this_obj is equal to
 * @p length.
 *
```

```
 * Preconditions:
 *  - arr != NULL
 *  - arr is not empty
 *  - sizeof_T != 0
 *  - length != 0
 * When calling this function, if any of these preconditions are false,
 *  - FACE_SEQUENCE_NULL_PARAM will be returned (if arr is NULL) or
 *    FACE_SEQUENCE_PRECONDITION_VIOLATED will be returned (if any other
 *    precondition is violated)
 *  - @p this_obj is put into the invalid state
 *
 * If no preconditions are violated and memory allocation fails:
 *  - FACE_SEQUENCE_INSUFFICIENT_MEMORY will be returned
 *  - @p this_obj is put into the invalid state
 *
 * The caller must ensure @p length * @p sizeof_T is not greater than the
 * size of the memory allocated at @p arr. If this condition is violated,
 * the result is implementation-defined behavior and may result in an
 * attempt to access restricted memory.
 *
 * @param this_obj the FACE_sequence to be initialized
 * @param arr a pointer to the array
 * @param sizeof_T the size of each element in the array
 * @param length the number of elements in the array
 *
 * @retval FACE_SEQUENCE_NULL_THIS if @p this_obj is null
 * @retval FACE_SEQUENCE_PRECONDITION_VIOLATED if @p this_obj is already
 *         initialized or any other preconditions are false
 * @retval FACE_SEQUENCE_NULL_PARAM if @p arr is null
 * @retval FACE_SEQUENCE_INSUFFICIENT_MEMORY if memory allocation fails
 * @retval FACE_SEQUENCE_NO_ERROR otherwise.
 */
FACE_sequence_return FACE_sequence_init_managed_data(
  FACE_sequence*     this_obj,
  const void *       arr,
  size_t             sizeof_T,
  FACE_unsigned_long length
);

/**
 * @brief Unmanaged initialization
 * @details (see #FACE_string_init_unmanaged)
 *
 * The caller must ensure @p bound * @p sizeof_T is not greater than the
 * size of the memory allocated at @p src. If this condition is violated,
 * the result is implementation-defined behavior and may result in an
 * attempt to access restricted memory.
 *
 * Preconditions:
 *  - src != NULL
 *  - length <= bound
 *  - bound != 0 (no empty unmanaged sequences)
 *  - bound != UNBOUNDED_SENTINEL (no unbounded unmanaged sequences)
 *  - sizeof_T != 0
 * When calling this function, if any of these preconditions are false,
 *  - FACE_SEQUENCE_NULL_PARAM will be returned (if src is NULL) or
 *  - FACE_SEQUENCE_PRECONDITION_VIOLATED will be returned (if any other
 *    preconditions are violated)
 *  - @p this_obj is put into the invalid state
 *
 * @param this_obj a pointer to the FACE_sequence to be initialized
 * @param src pointer to externally managed memory
 * @param length the number of elements in the memory pointed to by @p src
```

```
 * @param sizeof_T the size of each element in the memory pointed to by
 *        @p src
 * @param bound the number of elements the externally managed memory can
 *        hold.
 * Also serves as a capacity.
 */
FACE_sequence_return FACE_sequence_init_unmanaged(
  FACE_sequence*     this_obj,
  void *             src,
  size_t             sizeof_T,
  FACE_unsigned_long length,
  FACE_unsigned_long bound
);

/**
 * @brief Frees any data managed by @p this_obj.
 * @details (see #FACE_string_free)
 */
FACE_sequence_return FACE_sequence_free(FACE_sequence* this_obj);

/**
 * @brief Clears @p this_obj's data.
 * @details (see #FACE_string_clear)
 */
FACE_sequence_return FACE_sequence_clear(FACE_sequence* this_obj);

/**
 * @brief Adds a copy of @p src's data to the @p this_obj's data
 * @details (see #FACE_string_append)
 */
FACE_sequence_return FACE_sequence_append(
  FACE_sequence*       this_obj,
  const FACE_sequence* src
);

/**
 * @brief Gets the element at a given index.
 * @details (see #FACE_sequence_at)
 *
 * @retval NULL if @p this_obj is null, not initialized, or if index is out
 * of range
 * @retval a const pointer to the element at the given index otherwise.
 */
const void * FACE_sequence_at(
  const FACE_sequence* this_obj,
  FACE_unsigned_long   index
);

/**
 * @brief Returns pointer to @p this_obj's underlying data
 * @details To avoid accessing restricted memory, the caller should avoid
 * dereferencing memory beyond buffer + length*(the size of each element).
 *
 * @retval NULL if @p this_obj is null or not initialized
 * @retval a pointer to contiguous memory for @p this_obj's data otherwise
 */
const void * FACE_sequence_buffer(const FACE_sequence* this_obj);

/**
 * @brief Gets the length of @p this_obj.
 * @details (see #FACE_string_length)
 */
FACE_sequence_return FACE_sequence_length(
```

```
  const FACE_sequence* this_obj,
  FACE_unsigned_long*  length
);

/**
 * @brief Gets the capacity of @p this_obj.
 * @details (see #FACE_string_capacity)
 */
FACE_sequence_return FACE_sequence_capacity(
  const FACE_sequence* this_obj,
  FACE_unsigned_long*  capacity
);

/**
 * @brief Gets the bound of @p this_obj.
 * @details (see #FACE_string_bound)
 */
FACE_sequence_return FACE_sequence_bound(
  const FACE_sequence* this_obj,
  FACE_unsigned_long*  bound
);

/**
 * @brief Gets whether or not @p this_obj is managed.
 * @details (see #FACE_string_is_managed)
 */
FACE_sequence_return FACE_sequence_is_managed(
  const FACE_sequence* this_obj,
  FACE_boolean*        is_managed
);

/**
 * @brief Gets whether or not @p this_obj is bounded.
 * @details (see #FACE_string_is_bounded)
 */
FACE_sequence_return FACE_sequence_is_bounded(
  const FACE_sequence* this_obj,
  FACE_boolean*        is_bounded
);

/**
 * @brief Gets whether or not @p this_obj is in the invalid state.
 * @details (see #FACE_string_is_valid)
 */
FACE_sequence_return FACE_sequence_is_valid(
  const FACE_sequence* this_obj,
  FACE_boolean*        is_valid
);

#endif /* _FACE_SEQUENCE_H */
```

K.1.4 FACE_string Specification

```
//! @file FACE/string.h
//! @brief Interface for operating on a sequence of characters.

#ifndef _FACE_STRING_H
#define _FACE_STRING_H

#include <FACE/types.h>
#include <limits.h>

/**
```

```
 * @brief Interface for operating on a sequence of characters.
 * @details A FACE_string is defined by three characteristics:
 *  - length - the current number of characters (excluding NUL)
 *             in the FACE_string
 *  - bound - the maximum number of characters (excluding NUL)
 *            the FACE_string can ever hold. This bound is logical, and is
 *            independent from the size of any underlying memory.
 *            A FACE_string's bound is fixed throughout the lifetime of the
 *            FACE_string. An "unbounded" FACE_string has an infinite
 *            bound, represented by FACE_STRING_UNBOUNDED_SENTINEL.
 *  - capacity - the number of characters (excluding NUL)
 *               a FACE_string has currently allocated memory for. This may
 *               vary by implementation, but length <= capacity <= bound is
 *               always true.
 *
 * A "managed" FACE_string is responsible for and manages the lifetime of
 * the memory for the data it represents. An "unmanaged" FACE_string
 * essentially wraps a pointer to memory whose lifetime is managed
 * elsewhere.
 *
 * A FACE_string is "initialized" if it is in a state that could have
 * resulted from successful initialization by one of the "_init" functions.
 * Any other state makes the FACE_string "uninitialized".
 *
 * When a memory allocation failure or precondition violation occurs, a
 * FACE_string is put into a known "invalid state". In this invalid state:
 *  - length, capacity, and bound are 0
 *  - FACE_string_buffer() will return NULL
 *  - FACE_string_is_managed() and FACE_string_is_bounded() will return
 *    FALSE
 * The FACE_string_is_valid() function indicates whether or not a
 * FACE_string is in this state.
 *
 * Global preconditions:
 * - In every function, if the @p this_obj parameter is NULL, the function
 *   does nothing and returns FACE_STRING_NULL_THIS.
 * - In every _init function, if this_obj is already initialized,
 *   FACE_STRING_PRECONDITION_VIOLATED is returned and the state of
 *   this_obj is not modified.
 * - In every non _init function, if this_obj has not been initialized,
 *   FACE_STRING_PRECONDITION_VIOLATED is returned and the state of
 *   this_obj is not modified.
 */
typedef struct {
  /* implementation-specific */
} FACE_string;

/** @brief Return codes used to report certain runtime errors. */
typedef enum FACE_string_return {
  FACE_STRING_NO_ERROR,             /**< No error has occurred. */
  FACE_STRING_INSUFFICIENT_BOUND,   /**< Executing a function would cause
                                         a FACE_string's length to exceed
                                         its bound. */
  FACE_STRING_INSUFFICIENT_MEMORY,  /**< A FACE_string is unable to
                                         allocate enough memory to perform
                                         some function. */
  FACE_STRING_PRECONDITION_VIOLATED, /**< A precondition of some function
                                         has been violated. */
  FACE_STRING_NULL_THIS,            /**< The "this_obj" parameter is a
                                         NULL pointer */
  FACE_STRING_NULL_PARAM,           /**< One or more other parameters is a
                                         NULL pointer */
  FACE_STRING_INVALID_PARAM         /**< A FACE_string parameter is
```

```
                                          invalid. */
} FACE_string_return;

/** @brief Value representing the bound of an unbounded FACE_string. */
#define FACE_STRING_UNBOUNDED_SENTINEL ((FACE_unsigned_long) UINT_MAX)

/**
 * @brief Unmanaged initialization
 * @details After initialization, @p this_obj does not manage its own data,
 * but instead serves as a wrapper to the data pointed to by @p src.
 *
 * The caller must ensure @p str is a NULL terminated string
 * If this condition is violated, the result is implementation-defined
 * behavior and may result in an attempt to access restricted memory.
 *
 * The capacity of this string is equal to the length of the NULL
 * terminated string @p str not counting the NULL terminator, because
 * the externally managed memory has a fixed size, which is both a bound
 * and a capacity.
 *
 * After construction the following are true:
 *  - length is the length of the current string not counting the NULL
 *    terminator
 *  - capacity is the capacity which is equal to the length of the
 *    original string not counting the NULL terminator
 *  - bound() is the same as capacity()
 *  - FACE_string_buffer() will return the address specified by @p str
 * @param str pointer to externally managed memory
 * (must be NULL terminated)
 * @retval this_obj
 *
 */
FACE_string *FACE_string_init_unmanaged_bounded(
  FACE_string* this_obj,
  const char * str
);

/**
 * @brief Managed unbounded initialization - initializes empty managed
 * unbounded FACE_string
 * @details No memory is allocated. After initialization,
 *  - length will be 0
 *  - capacity will be 0
 *  - bound will be FACE_STRING_UNBOUNDED_SENTINEL
 *  - FACE_string_buffer() will get the empty string
 *
 * @param this_obj the FACE_string to be initialized.
 *
 * @retval FACE_STRING_NULL_THIS if @p this_obj is null
 * @retval FACE_STRING_PRECONDITION_VIOLATED if @p this_obj is already
 *         initialized
 * @retval FACE_STRING_NO_ERROR otherwise.
 */
FACE_string_return FACE_string_init_managed_unbounded(
  FACE_string* this_obj
);

/**
 * @brief Managed bounded initialization - initializes empty managed
 * FACE_string of specified bound
 * @details Memory may or may not be allocated.
 *
 * Preconditions:
```

```
 *  - bound != 0
 *  - bound != FACE_STRING_UNBOUNDED_SENTINEL
 * When calling this function, if any of these preconditions are false,
 *  - FACE_STRING_PRECONDITION_VIOLATED will be returned
 *  - @p this_obj is put into the invalid state
 *
 * While the implementation does not have to allocate memory equal in
 * size to the requested bound, memory allocation may still fail. If no
 * preconditions are violated and memory allocation fails:
 *  - FACE_STRING_INSUFFICIENT_MEMORY will be returned
 *  - @p this_obj is put into the invalid state
 *
 * Otherwise:
 *  - length will be 0
 *  - capacity will be the current capacity
 *  - bound will be the specified bound
 *  - FACE_string_buffer() will get the empty string
 *
 * @param this_obj the FACE_string to be initialized
 * @param bound the specified bound for the @p this_obj to be initialized
 *        with
 *
 * @retval FACE_STRING_NULL_THIS if @p this_obj is null
 * @retval FACE_STRING_PRECONDITION_VIOLATED if @p this_obj is already
 *         initialized or if any other preconditions are false
 * @retval FACE_STRING_INSUFFICIENT_MEMORY if memory allocation fails
 * @retval FACE_STRING_NO_ERROR otherwise.
 */
FACE_string_return FACE_string_init_managed_bounded(
  FACE_string*       this_obj,
  FACE_unsigned_long bound
);

/**
 * @brief Managed copy initialization
 * @details After initialization, @p this_obj manages its own data, which
 * is a copy of @p src's data, and has the same bound as @p src.
 *
 * Preconditions:
 *  - @p src != NULL
 *  - @p src is initialized
 * When calling this function, if any of these preconditions are false,
 *  - FACE_STRING_NULL_PARAM will be returned (if src is NULL) or
 *    FACE_STRING_PRECONDITION_VIOLATED will be returned (if src is not
 *    initialized)
 *  - @p this_obj is put into the invalid state
 *
 * If no preconditions are violated and memory allocation fails:
 *  - FACE_STRING_INSUFFICIENT_MEMORY will be returned
 *  - @p this_obj is put into the invalid state
 *
 * @param this_obj the FACE_string to be initialized
 * @param src the FACE_string to initialize @p this_obj with
 *
 * @retval FACE_STRING_NULL_THIS if @p this_obj is null
 * @retval FACE_STRING_PRECONDITION_VIOLATED if @p this_obj is already
 *         initialized or if @p src is not initialized
 * @retval FACE_STRING_NULL_PARAM if @p src is null
 * @retval FACE_STRING_INVALID_PARAM if @p src is in the invalid state
 * @retval FACE_STRING_INSUFFICIENT_MEMORY if memory allocation fails
 * @retval FACE_STRING_NO_ERROR otherwise.
 */
FACE_string_return FACE_string_init_managed_copy(
```

```
  FACE_string* this_obj,
  FACE_string* src
);

/**
 * @brief Managed C-string initialization
 * @details After initialization, @p this_obj manages its own data, which
 * is a copy of @p cstr, and the bound of @p this_obj is equal to @p
 * cstr's length.
 *
 * Preconditions:
 *  - cstr != NULL
 *  - cstr is not empty
 * When calling this function, if any of these preconditions are false,
 *  - FACE_STRING_NULL_PARAM will be returned (if cstr is NULL) or
 *    FACE_STRING_PRECONDITION_VIOLATED will be returned (if cstr is not
 *    empty)
 *  - @p this_obj is put into the invalid state
 *
 * If no preconditions are violated and memory allocation fails:
 *  - FACE_STRING_INSUFFICIENT_MEMORY will be returned
 *  - @p this_obj is put into the invalid state
 *
 * @param this_obj the FACE_string to be initialized
 * @param cstr a NUL-terminated string to initialize @p this_obj's data
 * with
 *
 * @retval FACE_STRING_NULL_THIS if @p this_obj is null
 * @retval FACE_STRING_PRECONDITION_VIOLATED if @p this_obj is already
 *         initialized or if @p cstr is empty
 * @retval FACE_STRING_NULL_PARAM if @p cstr is null
 * @retval FACE_STRING_INSUFFICIENT_MEMORY if memory allocation fails
 * @retval FACE_STRING_NO_ERROR otherwise.
 */
FACE_string_return FACE_string_init_managed_cstring(
  FACE_string* this_obj,
  const char * cstr
);

/**
 * @brief Unmanaged initialization
 * @details After initialization, @p this_obj does not manage its own data,
 * but instead serves as a wrapper to the data pointed to by @p src.
 *
 * The caller must ensure @p bound (plus space for NUL)
 * is not greater than the size of the memory allocated at @p src.
 * If this condition is violated, the result is implementation-defined
 * behavior and may result in an attempt to access restricted memory.
 *
 * The capacity of @p this_obj will be equal to its bound, because the
 * externally managed memory has a fixed size, which is both a bound and a
 * capacity.
 *
 * Preconditions:
 *  - src != NULL
 *  - length <= bound
 *  - bound != 0 (no empty unmanaged strings)
 *  - bound != UNBOUNDED_SENTINEL (no unbounded unmanaged strings)
 * When calling this function, if any of these preconditions are false,
 *  - FACE_STRING_NULL_PARAM will be returned (if src is NULL) or
 *  - FACE_STRING_PRECONDITION_VIOLATED will be returned (if any other
 *    preconditions are violated)
 *  - @p this_obj is put into the invalid state
```

```
 *
 * Otherwise:
 *  - FACE_STRING_NO_ERROR will be returned
 *  - length will be the specified length
 *  - capacity will return the specified capacity (bound)
 *  - bound() will return the specified bound
 *  - FACE_string_buffer() will return a pointer to the externally managed
 *    memory
 *
 * @param this_obj a pointer to the FACE_string to be initialized
 * @param src pointer to externally managed memory
 * @param length the number of characters (excluding the NUL character) in
 *        the memory pointed to by @p src
 * @param bound the number of characters (excluding the NUL character)
 *        the externally managed memory can hold. Also serves as a
 * capacity.
 *
 * @retval FACE_STRING_NULL_THIS if @p this_obj is null
 * @retval FACE_STRING_PRECONDITION_VIOLATED if @p this_obj is already
 *         initialized or any other preconditions are false
 * @retval FACE_STRING_NULL_PARAM if @p src is null
 * @retval FACE_STRING_NO_ERROR otherwise.
 */
FACE_string_return FACE_string_init_unmanaged(
  FACE_string*       this_obj,
  char*              src,
  FACE_unsigned_long length,
  FACE_unsigned_long bound
);

/**
 * @brief Unmanaged initialization from a C-string literal
 * @details After initialization, @p this_obj does not manage its own data,
 * but instead serves as a wrapper to the data pointed to by @p src.
 *
 * This function has exactly the same behavior, preconditions, and possible
 * return codes as calling FACE_string_init_unmanaged where the values for
 * @p length and &p bound are both strlen(&p src).
 *
 * The motivation for this function is to initialize a FACE_string from a
 * C-string literal. C-string literals are often stored in read-only
 * memory. Calling a FACE_string interface function that modifies the
 * string value on a string instance that was initialized from read-only
 * memory results in implementation-defined behavior such as as access
 * violation.
 *
 * @param this_obj a pointer to the FACE_string to be initialized
 * @param src pointer to externally managed memory
 */
FACE_string_return FACE_string_init_unmanaged_cstring(
  FACE_string* this_obj,
  char* src
);

/**
 * @brief Frees any data managed by @p this_obj.
 * @details If any preconditions are violated, @p this_obj's state remains
 * unchanged.
 *
 * Preconditions:
 *  - @p this_obj is managed
 *
 * @retval FACE_STRING_NULL_THIS if @p this_obj is null
```

```
 * @retval FACE_STRING_PRECONDITION_VIOLATED if @p this_obj is not
 *         initialized or any other preconditions are false
 * @retval FACE_STRING_NO_ERROR otherwise.
 */
FACE_string_return FACE_string_free(FACE_string* this_obj);

/**
 * @brief Clears @p this_obj's data.
 * @details If any preconditions are violated, @p this_obj's state remains
 * unchanged.
 *
 * Otherwise, all data is cleared, and @p this_obj's length will be set
 * to 0. Memory allocation remains unchanged.
 *
 * @retval FACE_STRING_NULL_THIS if @p this_obj is null
 * @retval FACE_STRING_PRECONDITION_VIOLATED if @p this_obj is not
 *         initialized or any other preconditions are false
 * @retval FACE_STRING_NO_ERROR otherwise.
 */
FACE_string_return FACE_string_clear(FACE_string* this_obj);

/**
 * @brief Adds a copy of @p src's data to the @p this_obj's data
 * @details This is the only FACE_string function that may reallocate
 * managed memory. If append is successful, the length of this String
 * changes accordingly; capacity may or may not be changed.
 * If append is unsuccessful, @p this_obj's state remains unchanged.
 *
 * Preconditions:
 *  - @p src != NULL
 *  - @p src is initialized
 * When calling this function, if any of these preconditions are false,
 *  - FACE_STRING_NULL_PARAM will be returned (if src is NULL) or
 *    FACE_STRING_PRECONDITION_VIOLATED will be returned (if any other
 *    preconditions are violated)
 *
 * @retval FACE_STRING_NULL_THIS if @p this_obj is null
 * @retval FACE_STRING_PRECONDITION_VIOLATED if @p this_obj is not
 *         initialized or if @p src is not initialized
 * @retval FACE_STRING_NULL_PARAM if @p src is null
 * @retval FACE_STRING_INSUFFICIENT_BOUND if append would exceed logical
 *         bound
 * @retval FACE_STRING_INSUFFICIENT_MEMORY if append exceeds available
 *         memory
 * @retval FACE_STRING_NO_ERROR otherwise
 */
FACE_string_return FACE_string_append(
  FACE_string*       this_obj,
  const FACE_string* src
);

/**
 * @brief Gets the character at a given index.
 * @details FACE_strings use a zero-based index.
 *
 * @param this_obj a const pointer to the FACE_string being indexed.
 * @param index The index of the element to be retrieved.
 *
 * @retval NULL if @p this_obj is null or not initialized
 * @retval '\0' if index is out of range
 * @retval a const pointer to the character at the given index otherwise
 */
const char * FACE_string_at(
```

```
  const FACE_string* this_obj,
  FACE_unsigned_long index
);

/**
 * @brief Returns C-string representation of @p this_obj's data
 *
 * @retval NULL if @p this_obj is null or not initialized
 * @retval a pointer to a NUL-terminated (C-style) string equivalent
 * to @p this_obj's underlying string data otherwise
 */
const char * FACE_string_buffer(const FACE_string* this_obj);

/**
 * @brief Gets the length of @p this_obj.
 *
 * @param this_obj a const pointer to the FACE_string to get the length of
 * @param length A pointer where the length will be stored.
 *
 * @retval FACE_STRING_NULL_THIS if @p this_obj is null
 * @retval FACE_STRING_PRECONDITION_VIOLATED if @p this_obj is not
 *         initialized
 * @retval FACE_STRING_NULL_PARAM if @p length is null
 * @retval FACE_STRING_NO_ERROR otherwise.
 */
FACE_string_return FACE_string_length(
  const FACE_string*  this_obj,
  FACE_unsigned_long* length
);

/**
 * @brief Gets the capacity of @p this_obj.
 *
 * @param this_obj a const pointer to the FACE_string to get the capacity
 * of
 * @param capacity A pointer where the capacity will be stored.
 *
 * @retval FACE_STRING_NULL_THIS if @p this_obj is null
 * @retval FACE_STRING_PRECONDITION_VIOLATED if @p this_obj is not
 *         initialized
 * @retval FACE_STRING_NULL_PARAM if @p capacity is null
 * @retval FACE_STRING_NO_ERROR otherwise.
 */
FACE_string_return FACE_string_capacity(
  const FACE_string*  this_obj,
  FACE_unsigned_long* capacity
);

/**
 * @brief Gets the bound of @p this_obj.
 *
 * @param this_obj a const pointer to the FACE_string to get the capacity
 * of
 * @param bound A pointer where the bound will be stored.
 *
 * @retval FACE_STRING_NULL_THIS if @p this_obj is null
 * @retval FACE_STRING_PRECONDITION_VIOLATED if @p this_obj is not
 *         initialized
 * @retval FACE_STRING_NULL_PARAM if @p bound is null
 * @retval FACE_STRING_NO_ERROR otherwise.
 */
FACE_string_return FACE_string_bound(
  const FACE_string*  this_obj,
```

```
  FACE_unsigned_long* bound
);

/**
 * @brief Gets whether or not @p this_obj is managed.
 *
 * @param this_obj a const pointer to the FACE_string to check
 * @param is_managed A pointer where the result will be stored.
 * @p is_managed will be 1 if @p this_obj manages its own memory, and 0
 * otherwise.
 *
 * @retval FACE_STRING_NULL_THIS if @p this_obj is null
 * @retval FACE_STRING_PRECONDITION_VIOLATED if @p this_obj is not
 *         initialized
 * @retval FACE_STRING_NULL_PARAM if @p is_managed is null
 * @retval FACE_STRING_NO_ERROR otherwise.
 */
FACE_string_return FACE_string_is_managed(
  const FACE_string* this_obj,
  FACE_boolean*      is_managed
);

/**
 * @brief Gets whether or not @p this_obj is bounded.
 *
 * @param this_obj a const pointer to the FACE_string to check
 * @param is_bounded A pointer where the result will be stored.
 *        @p is_bounded will be 1 if @p this_obj is bounded, and 0
 *        otherwise.
 *
 * @retval FACE_STRING_NULL_THIS if @p this_obj is null
 * @retval FACE_STRING_PRECONDITION_VIOLATED if @p this_obj is not
 *         initialized
 * @retval FACE_STRING_NULL_PARAM if @p is_bounded is null
 * @retval FACE_STRING_NO_ERROR otherwise.
 */
FACE_string_return FACE_string_is_bounded(
  const FACE_string* this_obj,
  FACE_boolean*      is_bounded
);

/**
 * @brief Gets whether or not @p this_obj is in the invalid state.
 * @details (see FACE_string details)
 *
 * @param this_obj a const pointer to the FACE_string to check
 * @param is_valid A pointer where the result will be stored. @p is_valid
 *        will be 0 if @p this_obj is in the invalid state, and 1
 *        otherwise.
 *
 * @retval FACE_STRING_NULL_THIS if @p this_obj is null
 * @retval FACE_STRING_PRECONDITION_VIOLATED if @p this_obj is not
 *         initialized
 * @retval FACE_STRING_NULL_PARAM if @p is_valid is null
 * @retval FACE_STRING_NO_ERROR otherwise.
 */
FACE_string_return FACE_string_is_valid(
  const FACE_string* this_obj,
  FACE_boolean*      is_valid
);

#endif /* _FACE_STRING_H */
```

K.1.5 FACE_fixed Specification

```
//! @file FACE/fixed.h
//! @brief Interface for a generic fixed type

#ifndef _FACE_FIXED_H
#define _FACE_FIXED_H

#include <FACE/types.h>

/**
 * @brief The maximum number of digits in a fixed-point number
 * (enforced by IDL).
 */
#define FACE FIXED DIGITS MAX ((FACE short) 31)

/**
 * @brief Data structure representing generic fixed type.
 */
typedef struct {
  /* implementation-specific */
} FACE_fixed;

/**
 * @brief Return codes for FACE_fixed functions
 *
 */
typedef enum {
  FACE_FIXED_NO_ERROR,    /**< No error has occurred. */
  FACE_FIXED_NULL_THIS,   /**< The "this_obj" parameter is a NULL pointer */
  FACE_FIXED_NULL_PARAM,  /**< One or more other parameters is a NULL
                               pointer */
  FACE_FIXED_INVALID_PARAM, /**< One or more fixed-point number parameters
                               are uninitialized or in an inconsistent
                               state. */
  FACE_FIXED_TOO_LARGE      /**< A value is too large to fit in a
                               fixed-point number. */
} FACE_fixed_return;

/**
 * @brief Initializes a fixed-point number with specified digits and scale.
 * @details After initialization, the value of @p this_obj is zero.
 *
 * @param this obj A pointer to the fixed-point number
 * @param digits Total number of digits
 * @param scale Number of significant fractional digits
 *
 * @retval FACE_FIXED_NULL_THIS if @p this_obj is null.
 * @retval FACE FIXED INVALID PARAM if any of the conditions in
 *         FACE fixed valid are violated
 * @retval FACE_FIXED_NO_ERROR Operation was successful.
 */
FACE_fixed_return FACE_fixed_init(FACE_fixed* this_obj,
                                  FACE_unsigned_short digits,
                                  FACE unsigned short scale
);

/**
 * @brief Gets the total number of digits of a fixed-point number.
 *
 * @param this obj A const pointer to the fixed-point number.
 * @param digits Pointer to digits; is set if call successful.
 *
 * @retval FACE_FIXED_NULL_THIS if @p this_obj is null.
 * @retval FACE FIXED NULL PARAM if @p digits is null.
 * @retval FACE FIXED INVALID PARAM if @p this obj is invalid.
 * @retval FACE FIXED NO ERROR Operation was successful.
 */
FACE_fixed_return FACE_fixed_digits(FACE_fixed*          this_obj,
                                    FACE_unsigned_short* digits);
```

```
/**
 * @brief Gets the scale (number of significant fractional digits) of a
 * fixed-point number.
 *
 * @param this_obj A const pointer to the fixed-point number.
 * @param scale Pointer to scale; is set if call successful.
 *
 * @retval FACE FIXED NULL THIS if @p this obj is null.
 * @retval FACE FIXED NULL PARAM if @p scale is null.
 * @retval FACE FIXED INVALID PARAM if @p this obj is invalid.
 * @retval FACE_FIXED_NO_ERROR Operation was successful.
 */
FACE_fixed_return FACE_fixed_scale(FACE_fixed*          this_obj,
                                   FACE_unsigned_short* scale);

/**
 * @brief Checks that the contents of a fixed-point number are consistent.
 * @details A fixed-point number is consistent if:
 *   - digits does not exceed FACE_FIXED_DIGITS_MAX.
 *   - scale does not exceed digits.
 *
 * @param this_obj a pointer to the fixed-point number.
 *
 * @retval FACE_FIXED_NULL_THIS if @p this_obj is null.
 * @retval FACE FIXED INVALID PARAM if any of the above conditions are
 *         violated.
 * @retval FACE_FIXED_NO_ERROR otherwise.
 */
FACE_fixed_return FACE_fixed_valid(FACE_fixed* this_obj);

/**
 * @brief Makes a copy of a fixed-point number.
 * @param this_obj a pointer to the destination fixed-point number
 * @param src a pointer to the source fixed-point number
 * @retval FACE_FIXED_NULL_THIS if @p this_obj is null.
 * @retval FACE FIXED NULL PARAM if @p src is null.
 * @retval FACE FIXED INVALID PARAM if @p this obj is an invalid fixed-point
 *         number.
 * @retval FACE_FIXED_NO_ERROR otherwise.
 */
FACE_fixed_return FACE_fixed_dup(FACE_fixed*       this_obj,
                                 const FACE_fixed* src
);


/**
 * @name Conversion Initializers
 * @anchor Conversion Initializers
 * @brief Initializes a fixed-point number by converting from another
 * representation.
 *
 * @param this obj a pointer to the fixed-point number
 * @param val the value to convert from
 *
 * @retval FACE_FIXED_NULL_PARAM if this_obj is null.
 * @retval FACE_FIXED_NO_ERROR otherwise.
 */
///@{
FACE_fixed_return FACE_fixed_init_short(
  FACE_fixed*      this_obj,
  const FACE_short val
);
FACE_fixed_return FACE_fixed_init_long(
  FACE_fixed*     this_obj,
  const FACE long val
);
FACE_fixed_return FACE_fixed_init_long_long(
  FACE_fixed*          this_obj,
  const FACE_long_long val
);
FACE_fixed_return FACE_fixed_init_unsigned_short(
```

```
  FACE_fixed*               this_obj,
  const FACE unsigned short val
);
FACE_fixed_return FACE_fixed_init_unsigned_long(
  FACE_fixed*              this_obj,
  const FACE_unsigned_long val
);
FACE_fixed_return FACE_fixed_init_unsigned_long_long(
  FACE_fixed*                   this_obj,
  const FACE unsigned long long val
);
FACE_fixed_return FACE_fixed_init_short(
  FACE_fixed*      this_obj,
  const FACE short val
);
FACE_fixed_return FACE_fixed_init_float(
  FACE_fixed*      this_obj,
  const FACE_float val
);
FACE_fixed_return FACE_fixed_init_double(
  FACE_fixed*       this_obj,
  const FACE_double val
);
FACE_fixed_return FACE_fixed_init_long_double(
  FACE_fixed*            this_obj,
  const FACE long double val
);

/**
 * @brief Initializes a fixed-point number with a string representation of a
 * fixed-point literal.
 * @details (see \ref Conversion Initializers)
 * A valid fixed-point literal string contains at least 1 and no more
 * than FACE_FIXED_DIGITS_MAX decimal digits, an optional decimal point, an
 * optional leading +/-, and an optional trailing d/D. The string must be
 * non-empty and null-terminated.
 *
 * @retval FACE FIXED NULL THIS if @p this obj is null.
 * @retval FACE_FIXED_NULL_PARAM if @p val is null.
 * @retval FACE_FIXED_INVALID_PARAM if @p val is an invalid fixed-point
 *         literal
 * @retval FACE FIXED NO ERROR otherwise.
 */
FACE_fixed_return FACE_fixed_init_str(FACE_fixed*      this_obj,
                                      const FACE_char* val);
///@}


/** @name Conversions */
///@{
/**
 * @brief Converts a fixed-point number to an integer number.
 * @details Digits to the right of the decimal point are truncated.
 *
 * @param this_obj a pointer to the fixed-point number source of the
 * conversion
 * @param dst a pointer to the destination of the conversion
 *
 * @retval FACE FIXED NULL THIS if @p this obj is null.
 * @retval FACE FIXED NULL PARAM if @p dst is null.
 * @retval FACE_FIXED_INVALID_PARAM if @p this_obj is an invalid fixed-point
 *         number.
 * @retval FACE_FIXED_TOO_LARGE if the magnitude of the fixed-point value
 *         does not fit in the destination of the conversion.
 * @retval FACE FIXED NO ERROR otherwise.
 */
FACE_fixed_return FACE_fixed_convert_integer(const FACE_fixed* this_obj,
                                             FACE_long_long*   dst);

/**
 * @brief Converts a fixed-point number to a floating-point number.
```

```
 *
 * @param this_obj a pointer to the fixed-point number source of the
 * conversion
 * @param dst a pointer to the destination of the conversion
 *
 * @retval FACE_FIXED_NULL_THIS if @p this_obj is null.
 * @retval FACE_FIXED_NULL_PARAM if @p dst is null.
 * @retval FACE_FIXED_INVALID_PARAM if @p this_obj is an invalid fixed-point
 *         number.
 * @retval FACE_FIXED_TOO_LARGE if the magnitude of the fixed-point value
 *         does not fit in the destination of the conversion.
 * @retval FACE_FIXED_NO_ERROR otherwise.
 */
FACE_fixed_return FACE_fixed_convert_floating(const FACE_fixed* this_obj,
                                              FACE_long_double* dst);

/**
 * @brief Converts a fixed-point number to a string representation of a
 * fixed-point literal.
 * @details The string starts with a '-' if negative and nothing if
 * positive, and always ends with a 'd'. Leading zeros are dropped, but
 * trailing fractional zeros are preserved. For example, a fixed-point
 * number with digits=4 and scale=2 with the value 1.1 is converted to
 * "1.10d".) (See #FACE_fixed_init_str for more details on valid string
 * representations of fixed point literals.)
 *
 * @param this_obj a pointer to the fixed-point number source of the
 * conversion @param dst a pointer to the destination of the conversion
 *
 * @retval FACE_FIXED_NULL_THIS if @p this_obj is null.
 * @retval FACE_FIXED_NULL_PARAM if @p dst is null.
 * @retval FACE_FIXED_INVALID_PARAM if @p this_obj is an invalid fixed-point
 *         number.
 * @retval FACE_FIXED_NO_ERROR otherwise.
 */
FACE_fixed_return FACE_fixed_convert_str(const FACE_fixed* this_obj,
                                         FACE_char*        dst);
///@}


/**
 * @name Precision Modifiers
 * @brief Converts a fixed value to a new value with a specified scale.
 * @details If the value currently has more digits on the right than the new
 * scale, the @p round function rounds away from values halfway or more the
 * to  the next absolute value for the new scale. If the value currently has
 * fewer or equal digits on the right relative to the new scale, both
 * functions return the value unmodified. The truncate function always
 * truncates the value towards zero. For example:
 *
 * f1: 0.1
 *   Round with scale 0 => 0
 *   Trunc with scale 0 => 0
 * f2: 0.05
 *   Round with scale 1 => 0.1
 *   Trunc with scale 1 => 0.0
 * f3: -0.005
 *   Round with scale 1 => -0.01
 *   Trunc with scale 1 =>  0.00
 *
 * @param this_obj a pointer to the source fixed-point number
 * @param result a pointer to the result
 * @param scale the new scale
 *
 * @retval FACE FIXED NULL THIS if @p this obj is null.
 * @retval FACE FIXED NULL PARAM if @p result is null.
 * @retval FACE_FIXED_INVALID_PARAM if @p this_obj is an invalid fixed-point
 *         number or if @p scale would cause @p result to be an invalid
 *         fixed-point number.
 * @retval FACE FIXED NO ERROR otherwise.
 */
```

```
///@{
FACE_fixed_return FACE_fixed_round(const FACE_fixed*         this_obj,
                                   FACE_fixed*               result,
                                   const FACE unsigned short scale
);
FACE_fixed_return FACE_fixed_truncate(const FACE_fixed*         this_obj,
                                      FACE_fixed*               result,
                                      const FACE unsigned short scale
);
///@}


/**
 * @name Arithmetic Operations (Binary)
 * @brief Performs an arithmetic operation on two fixed-point numbers.
 * @details These functions calculate a result exactly using double
 * precision arithmetic and truncating the result to fit into the smallest
 * valid fixed-point number that can represent the result.
 *
 * The following table summarizes the type resulting from an operation on
 * two fixed point numbers:
 *  - one with _digits=d1 and _scale=s1 (fixed<d1, s1>)
 *  - another with _digits=d2 and _scale=s2 (fixed<d2, s2>).
 * (s_inf denotes an arbitrary number of decimal places.)
 *
 * Operation | Result (fixed<d, s>)
 * --------- | ------
 *     +     |  fixed<max(d1-s1,d2-s2) + max(s1,s2) + 1, max(s1,s2)>
 *     -     |  fixed<max(d1-s1,d2-s2) + max(s1,s2) + 1, max(s1,s2)>
 *     *     |  fixed<d1+d2, s1+s2>
 *     /     |  fixed<(d1-s1+s2) + s inf, s inf>
 * remainder |  fixed<max(s1,s2), max(s1,s2)>
 *
 * If the actual result is more than FACE_FIXED_DIGITS_MAX significant
 * digits, the result is retained as:
 *  fixed<d,s> => fixed<FACE FIXED DIGITS MAX, FACE FIXED DIGITS MAX-d+s>.
 *
 * Any of the three parameters may be equal (i.e. point to the same
 * fixed-point number in memory); local temporary copies ensure the result
 * is as expected.
 *
 * @param result a pointer the result of the operation.
 * @param operand1 a pointer to the first operand.
 * @param operand2 a pointer to the second operand.

 * @retval FACE_FIXED_NULL_PARAM if any parameter is null.
 * @retval FACE FIXED INVALID PARAM if @p operand1 or @p operand2 is an
 *         invalid fixed-point number.
 * @retval FACE FIXED TOO LARGE if the magnitude of the actual result does
 *         not fit in a valid fixed-point number.
 * @retval FACE_FIXED_NO_ERROR otherwise.
 */
///@{
/** @brief Calculates the sum of two fixed-point numbers. */
FACE_fixed_return FACE_fixed_add(FACE_fixed* result,
                                 const FACE_fixed* operand1,
                                 const FACE_fixed* operand2
);
/** @brief Calculates the difference of two fixed-point numbers. */
FACE_fixed_return FACE_fixed_subtract(FACE_fixed* result,
                                      const FACE_fixed* operand1,
                                      const FACE_fixed* operand2
);
/** @brief Calculates the product of two fixed-point numbers. */
FACE_fixed_return FACE_fixed_multiply(FACE_fixed* result,
                                      const FACE_fixed* operand1,
                                      const FACE_fixed* operand2
);
/** @brief Calculates the quotient of two fixed-point numbers. */
FACE_fixed_return FACE_fixed_divide(FACE_fixed* result,
                                    const FACE_fixed* operand1,
```

```
                                    const FACE_fixed* operand2
);
/**
 * @brief Calculates the remainder after division of two fixed-point
 * numbers.
 */
FACE_fixed_return FACE_fixed_remainder(FACE_fixed* result,
                                       const FACE_fixed* operand1,
                                       const FACE_fixed* operand2);
///@}


/**
 * @name Arithmetic Operations (In-place)
 * @brief Performs an arithmetic operation on a fixed-point number.
 *
 * @param this_obj a pointer to the fixed-point number.
 *
 * @retval FACE_FIXED_NULL_THIS if @p this_obj is null.
 * @retval FACE FIXED INVALID PARAM if @p this obj is an invalid fixed-point
 *         number.
 * @retval FACE_FIXED_TOO_LARGE if the magnitude of the operation result
 *         does not fit in a valid fixed-point number.
 * @retval FACE_FIXED_NO_ERROR otherwise.
 */
///@{
/** @brief Increments a fixed-point number by 1. */
FACE_fixed_return FACE_fixed_inc(FACE_fixed* this_obj);
/** @brief Decrements a fixed-point number by 1. */
FACE_fixed_return FACE_fixed_dec(FACE_fixed* this_obj);
/** @brief Makes a fixed-point number positive. */
FACE_fixed_return FACE_fixed_pos(FACE_fixed* this_obj);
/** @brief Makes a fixed-point number negative. */
FACE_fixed_return FACE_fixed_neg(FACE_fixed* this_obj);
///@}

/**
 * @brief Indicates whether or not a fixed-point value is 0.
 * @details @p result will be set to 0 if the value is 0, and 1 otherwise
 *
 * @param this obj a pointer to the fixed-point number.
 * @param result a pointer to the result
 *
 * @retval FACE FIXED NULL THIS if @p this obj is null.
 * @retval FACE_FIXED_NULL_PARAM if @p result is null.
 * @retval FACE_FIXED_INVALID_PARAM if @p this_obj is an invalid fixed-point
 *         number.
 * @retval FACE FIXED NO ERROR otherwise.
 */
FACE_fixed_return FACE_fixed_not(const FACE_fixed* this_obj,
                                 FACE_boolean*     result);

/** @name Comparison Operations
 * @brief Performs comparisons of fixed-point numbers.
 * @details @p return_code will be:
 *  - FACE_FIXED_NULL_PARAM if any parameter is null.
 *  - FACE_FIXED_INVALID_PARAM if @p lhs or @p rhs is an invalid fixed-point
 *    number.
 *  - FACE FIXED NO ERROR otherwise.
 *
 * @param lhs the left-hand side of the comparison.
 * @param rhs the right-hand side of the comparison.
 * @param return_code pointer to a return code resulting from the function's
 *        invocation.
 */
///@{
/** @brief Returns 1 if lhs greater than rhs; 0 otherwise. */
FACE_boolean FACE_fixed_gt(const FACE_fixed*  lhs,
                           const FACE_fixed*  rhs,
                           FACE_fixed_return* return_code
);
```

```
/** @brief Returns 1 if lhs less than rhs; 0 otherwise. */
FACE_boolean FACE_fixed_lt(const FACE_fixed*  lhs,
                           const FACE_fixed*  rhs,
                           FACE_fixed_return* return_code
);
/** @brief Returns 1 if lhs greater than or equal to rhs; 0 otherwise. */
FACE_boolean FACE_fixed_gteq(const FACE_fixed*  lhs,
                             const FACE_fixed*  rhs,
                             FACE_fixed_return* return_code
);
/** @brief Returns 1 if lhs less than or equal to rhs; 0 otherwise. */
FACE_boolean FACE_fixed_lteq(const FACE_fixed*  lhs,
                             const FACE_fixed*  rhs,
                             FACE_fixed_return* return_code
);
/** @brief Returns 1 if lhs is equal to rhs; 0 otherwise. */
FACE_boolean FACE_fixed_eq(const FACE_fixed*  lhs,
                           const FACE_fixed*  rhs,
                           FACE_fixed_return* return_code
);
/** @brief Returns 1 if lhs is not equal to rhs; 0 otherwise. */
FACE_boolean FACE_fixed_neq(const FACE_fixed*  lhs,
                            const FACE_fixed*  rhs,
                            FACE_fixed_return* return_code);
///@}


#endif /* _FACE_FIXED_H */
```

K.2 C++ Programming Language

K.2.1 Basic Type Mapping

```
//! @file FACE/types.hpp
//! @brief Definitions of C++ types for IDL basic types to C++ mapping
//! @details This file contains editable type definitions for C++ types that
//! align with the size and range requirements given in the IDL basic types
//! to C++ mapping. Because C++ types' sizes and ranges are
//! platform-dependent, implementations are responsible for supplying full
//! type definitions.

#ifndef _FACE_TYPES_HPP
#define _FACE_TYPES_HPP

namespace FACE
{
  typedef EDITME Short;
  typedef EDITME Long;
  typedef EDITME LongLong;
  typedef EDITME UnsignedShort;
  typedef EDITME UnsignedLong;
  typedef EDITME UnsignedLongLong;
  typedef EDITME Float;
  typedef EDITME Double;
  typedef EDITME LongDouble;
  typedef EDITME Char;
  typedef EDITME Boolean;
  typedef EDITME Octet;
}

#endif /* _FACE_TYPES_HPP */
```

K.2.2 FACE::Sequence Specification

```
//! @file FACE/Sequence.hpp
//! @brief Template class representing a sequence of elements.
//! @details Because template classes must be implemented in header files,
```

```
//! this class specification includes empty-bodied definitions for its
//! methods.

#ifndef _FACE_SEQUENCE_HPP
#define _FACE_SEQUENCE_HPP

#include <FACE/types.hpp>
#include <limits.h>

namespace FACE
{
  /**
   * @brief Class representing a sequence of elements of type T
   * @details A FACE::Sequence is defined by three characteristics:
   *  - length - the current number of elements in the Sequence
   *  - bound - the maximum number of elements the Sequence can ever hold.
   *            This bound is logical, and is independent from the size of
   *            any underlying memory. A Sequence's bound is fixed
   *            throughout the lifetime of the Sequence. An "unbounded"
   *            sequence has an infinite bound, represented by
   *            FACE::Sequence::UNBOUNDED_SENTINEL.
   *  - capacity - the number of elements the Sequence has currently
   *               allocated memory for. This may vary by implementation,
   *               but length <= capacity <= bound is always true.
   * A "managed" Sequence is responsible for and manages the lifetime of
   * the memory for the data it represents. An "unmanaged" Sequence
   * essentially wraps a pointer to memory whose lifetime is managed
   * elsewhere.
   *
   * In general, Sequence method behavior is identical to String method
   * behavior, except where otherwise noted. FACE::Sequence<T>::RETURN_CODE
   * is used in place of FACE::String::RETURN_CODE.
   *
   * This class does not throw exceptions, but precondition violations and
   * memory allocation failures can occur in constructors and other methods
   * that cannot return a value. In these situations, a Sequence object is
   * put into a known "invalid state", used to indicate that an object has
   * been constructed but is not valid and should not be used. In this
   * invalid state:
   *  - length(), bound(), capacity() will return 0
   *  - buffer() will return NULL
   *  - is_managed() and is_bounded() will return FALSE
   * The is_valid() method indicates whether or not an object is in this
   * state.
   *
   * @tparam the element type
   */
  template <typename T>
  class Sequence {
  public:
    /** @brief Return codes used to report certain runtime errors. */
    enum RETURN_CODE {
        NO_ERROR,             /**< No error has occurred. */
        INSUFFICIENT_BOUND,   /**< Executing a function would cause a
                                   Sequence's length to exceed its bound.
                                   */
        INSUFFICIENT_MEMORY,  /**< A Sequence is unable to allocate enough
                                   memory to perform some function. */
        PRECONDITION_VIOLATED /**< A precondition of some function has been
                                   violated. */
    };

    /** @brief Constant representing the bound of an unbounded Sequence. */
```

```
static const unsigned int UNBOUNDED_SENTINEL = UINT_MAX;

/**
 * @brief Default constructor - creates empty managed unbounded
 * Sequence
 * @details (see #FACE::String Default constructor)
 *
 * After construction, the Sequence will be empty.
 */
Sequence()
{ /* insert definition */ }

/**
 * @brief Managed constructor - creates empty Sequence of specified
 * bound
 * @details (see #FACE::String Managed constructor)
 *
 * If allocation is successful, the Sequence will be empty.
 */
Sequence(FACE::UnsignedLong bound, RETURN_CODE& return_code)
{ /* insert definition */ }

/**
 * @brief Managed copy constructor
 * @details (see #FACE::String Managed copy constructor)
 */
Sequence(const Sequence& seq)
{ /* insert definition */ }

/**
 * @brief Managed assignment operator
 * @details ( see #FACE::String::operator=)
 */
Sequence& operator=(const Sequence& seq)
{ /* insert definition */ }

/**
 * @brief Managed C-style array constructor
 * @details After construction, this Sequence manages its own data,
 * which is a copy of the @p length elements pointed to by @p arr, and
 * bound() will return @p length.
 *
 * Preconditions:
 *  - arr != NULL
 * When calling this function, if any of these preconditions are false,
 *  - return_code will be set to PRECONDITION_VIOLATED
 *  - this String is put into the invalid state
 *
 * If no preconditions are violated and memory allocation fails:
 *  - return_code will be set to INSUFFICIENT_MEMORY
 *  - this String is put into the invalid state
 *
 * The caller must ensure @p length * sizeof(T) is not greater than the
 * size of the memory allocated at @p arr. If this condition is
 * violated, the result is undefined behavior and may result in an
 * attempt to access restricted memory.
 *
 * @param arr A pointer to the C-style array
 * @param length The number of elements in the array
 * @param return_code (see details)
 */
Sequence(const T * arr,
         FACE::UnsignedLong length,
```

```
         RETURN_CODE& return_code)
{ /* insert definition */ }

/**
 * @brief Unmanaged constructor
 * @details (see #FACE::String::String)
 *
 * The caller must ensure @p bound + sizeof(T) is not greater than the
 * size of the memory allocated at @p seq. If this condition is
 * violated, the result is undefined behavior and may result in an
 * attempt to access restricted memory.
 *
 * @param seq pointer to externally managed memory
 * @param length the number of elements in the memory pointed to by
 *        @p seq @param bound the number of elements the externally
 *        managed memory can hold. Also serves as a capacity.
 * @param return_code (see details)
 */
Sequence(T * seq,
         FACE::UnsignedLong length,
         FACE::UnsignedLong bound,
         RETURN_CODE& return_code)
{ /* insert definition */ }

/**
 * @brief Frees any data managed by this Sequence.
 */
~Sequence()
{ /* insert definition */ }

/**
 * @brief Clears this String's data.
 * @details (see #FACE::String::clear)
 */
void clear()
{ /* insert definition */ }

/**
 * @brief Adds a copy of @p seq's data to the current data.
 * @details (see #FACE::String::append)
 */
RETURN_CODE append(const Sequence& seq)
{ /* insert definition */ }

/**
 * @brief Returns a reference to the element at a given index.
 * @details (see #FACE::String::operator[])
 *
 * If @p index is out of range, the behavior is implementation-defined.
 */
///@{
T& operator[](FACE::UnsignedLong index)
{ /* insert definition */ }
const T& operator[](FACE::UnsignedLong index) const
{ /* insert definition */ }
///@}

/**
 * @brief Returns pointer to contiguous memory for underlying data
 * @details To avoid accessing restricted memory, the caller should
 * avoid dereferencing memory beyond buffer() + length() * sizeof(T).
 */
///@{
```

```
    T * buffer()
    { /* insert definition */ }
    const T * buffer() const
    { /* insert definition */ }
    ///@}

    /** @brief Returns the length of this Sequence */
    FACE::UnsignedLong length() const
    { /* insert definition */ }

    /** @brief Returns the capacity of this Sequence */
    FACE::UnsignedLong capacity() const
    { /* insert definition */ }

    /** @brief Returns the bound of this Sequence */
    FACE::UnsignedLong bound() const
    { /* insert definition */ }

    /**
     * @brief Returns whether or not this Sequence is managed.
     * @details (see #FACE::String::is_managed)
     */
    FACE::Boolean is_managed() const
    { /* insert definition */ }

    /**
     * @brief Returns whether or not this Sequence is bounded.
     * @details (see #FACE::String::is_bounded)
     */
    FACE::Boolean is_bounded() const
    { /* insert definition */ }

    /**
     * @brief Returns whether or not this Sequence is in the invalid state.
     * @details (see class details)
     */
    FACE::Boolean is_valid() const
    { /* insert definition */ }

  private:
    /* implementation-specific */
  };
}

#endif /* _FACE_SEQUENCE_HPP */
```

K.2.3 FACE::String Specification

```
//! @file FACE/String.hpp
//! @brief Class representing a sequence of strings.

#ifndef _FACE_STRING_HPP
#define _FACE_STRING_HPP

#include <FACE/types.hpp>
#include <limits.h>

namespace FACE
{
  /**
   * @brief Class representing a sequence of characters.
   * @details A FACE::String is defined by three characteristics:
   *  - length - the current number of characters (excluding NUL)
```

```
 *             in the String
 *  - bound - the maximum number of characters (excluding NUL)
 *            the String can ever hold. This bound is logical, and is
 *            independent from the size of any underlying memory.
 *            A String's bound is fixed throughout the lifetime of the
 *            String. An "unbounded" String has an infinite bound,
 *            represented by FACE::String::UNBOUNDED_SENTINEL.
 *  - capacity - the number of characters (excluding NUL)
 *               a String has currently allocated memory for. This may
 *               vary by implementation, but length <= capacity <= bound
 *               is always true.
 *
 * A "managed" String is responsible for and manages the lifetime of the
 * memory for the data it represents. An "unmanaged" String essentially
 * wraps a pointer to memory whose lifetime is managed elsewhere.
 *
 * This class does not throw exceptions, but precondition violations and
 * memory allocation failures can occur in constructors and other methods
 * that cannot return a value. In these situations, a String object is
 * put into a known "invalid state", used to indicate that an object has
 * been constructed but is not valid and should not be used. In this
 * invalid state:
 *  - length(), bound(), capacity() will return 0
 *  - buffer() will return NULL
 *  - is_managed() and is_bounded() will return FALSE
 * The is_valid() method indicates whether or not an object is in this
 * state.
 */
class String {
public:
  /** @brief Return codes used to report certain runtime errors. */
  enum RETURN_CODE {
    NO_ERROR,             /**< No error has occurred. */
    INSUFFICIENT_BOUND,   /**< Executing a function would cause a
                               String's
                               length to exceed its bound. */
    INSUFFICIENT_MEMORY,  /**< A String is unable to allocate enough
                               memory to perform some function. */
    PRECONDITION_VIOLATED /**< A precondition of some function has been
                               violated. */
  };

  /** @brief Constant representing the bound of an unbounded String. */
  static const unsigned int UNBOUNDED_SENTINEL = UINT_MAX;

  /**
   * @brief Default constructor - creates empty managed unbounded String
   * @details No memory is allocated. After construction,
   *  - length() will return 0
   *  - capacity() will return 0
   *  - bound() will return UNBOUNDED_SENTINEL
   *  - buffer() will return the empty string
   */
  String();

  /**
   * @brief Managed constructor - creates empty managed bounded String
   * of specified bound
   * @details Memory may or may not be allocated.
   *
   * Preconditions:
   *  - bound != 0
   *  - bound != UNBOUNDED_SENTINEL
```

```
     * When calling this function, if any of these preconditions are false,
     *  - return_code will be set to PRECONDITION_VIOLATED
     *  - this String is put into the invalid state
     *
     * While the implementation does not have to allocate memory equal in
     * size to the requested bound, memory allocation may still fail. If no
     * preconditions are violated and memory allocation fails:
     *  - return_code will be set to INSUFFICIENT_MEMORY
     *  - this String is put into the invalid state
     *
     * Otherwise:
     *  - return_code will be set to NO_ERROR
     *  - length() will return 0
     *  - capacity() will return the current capacity
     *  - bound() will return the specified bound
     *  - buffer() will return the empty string
     */
    String(FACE::UnsignedLong bound, RETURN_CODE& return_code);

    /**
     * @brief Unmanaged constructor
     * @details After construction, this String does not manage its own
     * data, but instead serves as a wrapper to the data pointed to by
     * @p str.
     *
     * The caller must ensure @p str is a NULL terminated string
     * If this condition is violated, the result is implementation-defined
     * behavior and may result in an attempt to access restricted memory.
     *
     * The capacity of this String is equal to the length of the NULL
     * terminated string @p str not counting the NULL terminator, because
     * the externally managed memory has a fixed size, which is both a
     * bound and a capacity.
     *
     * After construction the following are true:
     *  - length() will return the length of the current string not
     *    counting the NULL terminator
     *  - capacity() will return the capacity which is equal to the
     *    length of the original string not counting the NULL terminator
     *  - bound() will return the same as capacity()
     *  - buffer() will return the address specified by @p str
     * @param str pointer to externally managed memory (must be NULL
     *    terminated)
     */
    String(const char * str);

    /**
     * @brief Managed copy constructor
     * @details After construction, this String manages its own data, which
     * is a copy of @p str's data, and has the same bound as @p str.
     * If sufficient memory cannot be allocated, this String is put into
     * the invalid state.
     */
    String(const String& str);

    /**
     * @brief Managed assignment operator
     * @details After assignment, this String's data is a copy of @p str's
     * data, and bound() will return @p str's bound. After assignment,
     * this String's data is managed.
     * If sufficient memory cannot be allocated, this String is put into
     * the invalid state.
     *
```

```
 * @return a reference to this String
 */
String& operator=(const String& str);

/**
 * @brief Managed C-string constructor
 * @details After successful construction, this String manages its own
 * data, which is a copy of @p str, and bound() will return @p str's
 * length.
 *
 * Preconditions:
 *  - str != NULL
 * When calling this function, if any of these preconditions are false,
 *  - return_code will be set to PRECONDITION_VIOLATED
 *  - this String is put into the invalid state
 *
 * If no preconditions are violated and memory allocation fails:
 *  - return_code will be set to INSUFFICIENT_MEMORY
 *  - this String is put into the invalid state
 *
 * @param str A NUL-terminated string.
 * @param return_code (see details)
 */
String(const char *str, RETURN_CODE& return_code);

/**
 * @brief Unmanaged constructor
 * @details After construction, this String does not manage its own
 * data, but instead serves as a wrapper to the data pointed to by
 * @p str.
 *
 * The caller must ensure @p bound (plus space for NUL)
 * is not greater than the size of the memory allocated at @p str.
 * If this condition is violated, the result is implementation-defined
 * behavior and may result in an attempt to access restricted memory.
 *
 * The capacity of this String is equal to its bound, because the
 * externally managed memory has a fixed size, which is both a bound
 * and a capacity.
 *
 * Preconditions:
 *  - str != NULL
 *  - length <= bound
 *  - bound != 0 (no empty unmanaged strings)
 *  - bound != UNBOUNDED_SENTINEL (no unbounded unmanaged strings)
 * When calling this function, if any of these preconditions are false:
 *  - return_code will be set to PRECONDITION_VIOLATED
 *  - this String is put into the invalid state
 *
 * Otherwise:
 *  - return_code will be set to NO_ERROR
 *  - length() will return the specified length
 *  - capacity() will return the specified capacity (bound)
 *  - bound() will return the specified bound
 *  - buffer() will return a pointer to the externally managed memory
 *
 * @param str pointer to externally managed memory
 * @param length the number of characters (excluding the NUL character)
 *        in the memory pointed to by @p str
 * @param bound the number of characters (excluding the NUL character)
 *        the externally managed memory can hold. Also serves as a
 *        capacity.
 * @param return_code (see details)
```

```
 */
String(char * str,
       FACE::UnsignedLong length,
       FACE::UnsignedLong bound,
       RETURN_CODE& return_code);

/**
 * @brief Frees any data managed by this String.
 */
~String();

/**
 * @brief Clears this String's data.
 * @details All data is cleared, and this String's length will be set
 * to 0. Memory allocation remains unchanged.
 */
void clear();

/**
 * @brief Adds a copy of @p str's data to the current data
 * @details This is the only String function that may reallocate
 * managed memory. If append is successful, the length of this String
 * changes accordingly; capacity may or may not be changed.
 * If append is unsuccessful, the state of this String is unchanged.
 *
 * @retval INSUFFICIENT_BOUND if append would exceed logical bound
 * @retval INSUFFICIENT_MEMORY if append exceeds available memory
 * @retval NO_ERROR otherwise
 */
RETURN_CODE append(const String& str);

/**
 * @brief Returns a reference to the character at a given index.
 * @details If @p index is out of range, '\0' is returned.
 * Strings use a zero-based index.
 *
 * @param index The index of the element to be retrieved
 * @return a reference to the desired element.
 */
///@{
char& operator[](FACE::UnsignedLong index);
const char& operator[](FACE::UnsignedLong index) const;
///@}

/**
 * @brief Returns C-string representation of string data
 * @details Returns a pointer to a NUL-terminated (C-style) string
 * equivalent to this String's underlying string data.
 */
///@{
char * buffer();
const char * buffer() const;
///@}

/** @brief Returns the length of this String */
FACE::UnsignedLong length() const;

/** @brief Returns the capacity of this String */
FACE::UnsignedLong capacity() const;

/** @brief Returns the bound of this String */
FACE::UnsignedLong bound() const;
```

```
    /**
     * @brief Returns whether or not this String is managed.
     *
     * @retval TRUE if this String manages its own memory
     * @retval FALSE otherwise
     */
    FACE::Boolean is_managed() const;

    /**
     * @brief Returns whether or not this String is bounded.
     * @details Equivalent to bound() != UNBOUNDED_SENTINEL
     * Unmanaged strings are always bounded.
     */
    FACE::Boolean is_bounded() const;

    /**
     * @brief Returns whether or not this String is in the invalid state.
     * @details (see class details)
     */
    FACE::Boolean is_valid() const;

  private:
    /* implementation-specific */
  };
}

#endif /* _FACE_STRING_HPP */
```

K.2.4 FACE::Fixed Specification

```
//! @file FACE/Fixed.hpp
//! @brief Class representing a generic fixed type

#ifndef _FACE_FIXED_HPP
#define  FACE_FIXED_HPP

#include <FACE/types.hpp>

namespace FACE
{

/**
 * @brief Class representing an IDL fixed type.
 */
class Fixed
{
public:
  /**
   * @brief The maximum number of digits in a fixed-point number
   * (enforced by IDL).
   */
  static const FACE::Short FACE_FIXED_DIGITS_MAX = (FACE::Short)31;

  /**
   * @brief Constructs a fixed point number with a specified digits and
   * scale.
   *
   * @param digits the total number of digits
   * @param scale the total number of significant fractional digits
   */
  Fixed(FACE::UnsignedLong digits, FACE::UnsignedLong scale);

  /**
   * @brief Copy constructor - creates a copy of a fixed type, maintaining
   * its digits, scale, and value.
   */
  Fixed(const Fixed& fx);
```

```
/**
 * @brief Assignment operator - assigns the digits, scale, and value of
 * one fixed type to another.
 */
Fixed& operator=(const Fixed& fx);

/**
 * @brief Destructor - performs any necessary implementation-specific
 * cleanup.
 */
~Fixed();

/**
 * @name Constructors (Conversion)
 * @anchor Constructors
 * @brief Constructs a fixed-point number by converting from another
 * representation.
 *
 * @param val the value to convert from
 */
///@{
explicit Fixed(FACE::Short val = 0);
explicit Fixed(FACE::UnsignedShort val);
explicit Fixed(FACE::Long val);
explicit Fixed(FACE::UnsignedLong val);
explicit Fixed(FACE::LongLong val);
explicit Fixed(FACE::UnsignedLongLong val);
explicit Fixed(FACE::Double val);
explicit Fixed(FACE::LongDouble val);

/**
 * @brief Constructs a fixed-point number with a string representation of
 * a  fixed-point literal.
 * @details (see \ref Constructors)
 * A valid fixed-point literal string contains at least 1 and no more
 * than FACE FIXED DIGITS MAX decimal digits, an optional decimal point,
 * an  optional leading +/-, and an optional trailing d/D. An invalid
 * fixed-point literal string results in implementation-defined behavior.
 */
explicit Fixed(const FACE::Char* val);
///@}

/** @name Conversions */
///@{
/**
 * @brief Converts a fixed-point number to an integer number.
 * @details Digits to the right of the decimal point are truncated.
 * If the magnitude of the fixed-point value does not fit, the result is
 * implementation-defined.
 */
operator FACE::LongLong() const;
/**
 * @brief Converts a fixed-point number to a floating-point number.
 * @details If the magnitude of the fixed-point value does not fit, the
 * result is implementation-defined.
 */
operator FACE::LongDouble() const;
/**
 * @brief Converts a fixed-point number to a string representation of a
 * fixed-point literal.
 * @details The string starts with a '-' if negative and nothing if
 * positive, and always ends with a 'd'. Leading zeros are dropped, but
 * trailing fractional zeros are preserved. For example, a fixed-point
 * number with digits=4 and scale=2 with the value 1.1 is converted to
 * "1.10d".) (See \ref Constructors for more details on valid string
 * representations of fixed point literals.)
 */
operator FACE::Char*() const;
///@}
```

```
/**
 * @name Precision Modifiers
 * @brief Converts a fixed value to a new value with a specified scale.
 * @details If the value currently has more digits on the right than the
 * new scale, the @p round function rounds away from values halfway or
 * more the to  the next absolute value for the new scale. If the value
 * currently has fewer or equal digits on the right relative to the new
 * scale, both functions return the value unmodified. The truncate
 * function always truncates the value towards zero. For example:
 *
 * f1: 0.1
 *   Round with scale 0 => 0
 *   Trunc with scale 0 => 0
 * f2: 0.05
 *   Round with scale 1 => 0.1
 *   Trunc with scale 1 => 0.0
 * f3: -0.005
 *   Round with scale 1 => -0.01
 *   Trunc with scale 1 =>  0.00
 *
 * If the specified scale would result in an invalid fixed-point number,
 * the result is implementation-defined.
 */
///@{
Fixed round(FACE::UnsignedShort scale) const;
Fixed truncate(FACE::UnsignedShort scale) const;
///@}


/**
 * @name Arithmetic Operations (In-place)
 * @brief Performs an arithmetic operation on a fixed-point number.
 * If the magnitude of the fixed-point value does not fit, the result is
 * implementation-defined.
 */
///@{
/** @brief Prefix increment */
Fixed& operator++();
/** @brief Postfix increment */
Fixed operator++(int);
/** @brief Prefix decrement */
Fixed& operator--();
/** @brief Postfix decrement */
Fixed operator--(int);
///@}

/**
 * @name Arithmetic Operations (Binary)
 * @brief Performs an arithmetic operation on two fixed-point numbers.
 * @details These functions calculate a result exactly using double
 * precision arithmetic and truncating the result to fit into the smallest
 * valid fixed-point number that can represent the result.
 *
 * The following table summarizes the type resulting from an operation on
 * two fixed point numbers:
 *  - one with _digits=d1 and _scale=s1 (fixed<d1, s1>)
 *  - another with _digits=d2 and _scale=s2 (fixed<d2, s2>).
 * (s inf denotes an arbitrary number of decimal places.)
 *
 * Operation | Result (fixed<d, s>)
 * --------- | ------
 *     +     |  fixed<max(d1-s1,d2-s2) + max(s1,s2) + 1, max(s1,s2)>
 *     -     |  fixed<max(d1-s1,d2-s2) + max(s1,s2) + 1, max(s1,s2)>
 *     *     |  fixed<d1+d2, s1+s2>
 *     /     |  fixed<(d1-s1+s2) + s inf, s inf>
 * remainder |  fixed<max(s1,s2), max(s1,s2)>
 *
 * If the actual result is more than FACE_FIXED_DIGITS_MAX significant
 * digits, the result is retained as:
 *  fixed<d,s> => fixed<FACE FIXED DIGITS MAX, FACE FIXED DIGITS MAX-d+s>.
 *
```

```
     * Any of the three parameters may be equal (i.e. point to the same
     * fixed-point number in memory); local temporary copies ensure the result
     * is as expected.
     *
     * If the magnitude of the result does not fit in a valid fixed-point
     * number, the result is implementation-defined.
     */
    ///@{
    /** @brief Calculates the in-place sum of two fixed-point numbers. */
    Fixed& operator += (const Fixed& rhs);
    /**
     * @brief Calculates the in-place difference of two fixed-point numbers.
     */
    Fixed& operator -= (const Fixed& rhs);
    /** @brief Calculates the in-place product of two fixed-point numbers. */
    Fixed& operator *= (const Fixed& rhs);
    /** @brief Calculates the in-place quotient of two fixed-point numbers. */
    Fixed& operator /= (const Fixed& rhs);
    /**
     * @brief Calculates the remainder after division of two
     * fixed-point numbers.
     */
    Fixed& operator % (const Fixed& rhs);
    ///@}

    /**
     * @name Comparison Operations
     * @brief Performs comparisons of fixed-point numbers.
     *
     * @param rhs the right-hand side of the comparison.
     */
    ///@{
    /** @brief Returns 1 if lhs greater than rhs; 0 otherwise. */
    FACE::Boolean operator > (const Fixed& rhs) const;
    /** @brief Returns 1 if lhs less than rhs; 0 otherwise. */
    FACE::Boolean operator < (const Fixed& rhs) const;
    /** @brief Returns 1 if lhs greater than or equal to rhs; 0 otherwise. */
    FACE::Boolean operator >= (const Fixed& rhs) const;
    /** @brief Returns 1 if lhs less than or equal to rhs; 0 otherwise. */
    FACE::Boolean operator <= (const Fixed& rhs) const;
    /** @brief Returns 1 if lhs is equal to rhs; 0 otherwise. */
    FACE::Boolean operator == (const Fixed& rhs) const;
    /** @brief Returns 1 if lhs is not equal to rhs; 0 otherwise. */
    FACE::Boolean operator != (const Fixed& rhs) const;
    ///@}


    /** @brief Calculates the absolute value of a fixed-point number. */
    Fixed operator+() const;
    /**
     * @brief Calculates the absolute value of a fixed-point number,
     * multiplied by -1.
     */
    Fixed operator-() const;


    /** @brief Returns false if value is 0; true otherwise. */
    FACE::Boolean operator!() const;

    /**
     * @brief Returns the smallest digits value that can hold the complete
     * fixed-point value.
     */
    FACE::UnsignedShort digits() const;
    /**
     * @brief Returns the smallest scale value that can hold the complete
     * fixed-point value.
     */
    FACE::UnsignedShort scale() const;
};
```

```
}

#endif /* _FACE_FIXED_HPP */
```

K.3 Ada Programming Language

K.3.1 Sequence Packages

The packages FACE.Sequences, FACE.Sequences.Bounded, and FACE.Sequences.Unbounded provide definitions of sequence types and interfaces to their operations. Implementations are responsible for providing the bodies of these packages. The constructs defined in the FACE.Sequences package mimic constructs in the Ada.Strings package, as does FACE.Sequences.Bounded mimic Ada.Strings.Bounded and FACE.Sequences.Unbounded mimic Ada.Strings.Unbounded, with the following modifications and clarifications:

- The Element generic formal parameter is the type of each element in the sequence
- Any reference to “string” in the Ada.Strings package specification can be read as “sequence”; any reference to “character” can be read as “element”

The behavior of each subprogram in the FACE.Sequences packages mimics the behavior of the subprogram with the same name and signature in the Ada.Strings package, with the following modifications and clarifications:

- All references to a Truncation value of Error are considered to be FACE.Sequences.Right
- For all subprograms that propagate Index_Error under some conditions, Constraint_Error is propagated under those conditions instead
- Comparison operators (“=”, “<”, “>”, “<=”, and “>=”) are an element-by-element comparison
- Get_Element – this function mimics the behavior of the Element function
- Delete – if From <= Through, the deleted elements are replaced with implementation-defined values representing a null element
- FACE.Sequences.Bounded."*" – if the result would exceed the sequence’s bound, truncation occurs by dropping elements from the right
- FACE.Sequences.Unbounded.Copy – this function allocates a new Sequence and copies the value of Source to the new Sequence

  (This supports definitions of the Sequence type where the Ada assignment (:=) operation would only perform a shallow copy.)
- FACE.Sequences.Unbounded.Is_Null – this function returns True if the sequence is uninitialized, and False otherwise
- FACE.Sequences.Unbounded.Free – this procedure does not necessarily perform an unchecked de-allocation

In general, FACE.Sequences.Unbounded functions that return a Sequence always return a newly allocated Sequence. Procedures that modify (mode is “in out”) a parameter that is a Sequence attempt to use the space, if any, that is pre-allocated and available to be used before allocating

new space. Allocation can occur if the allocated space for the Sequence is too small to hold a new Sequence, but also if this space is too large, so that the reduced space requirement of the used portion is considered a waste of memory.

Note: Ada.Strings is only referenced here as a means of specification, and is not intended to imply the use of that package.

K.3.1.1 FACE.Sequences Specification

```
package FACE.Sequences is

    type Truncation is (Left, Right);
    type Direction  is (Forward, Backward);

end FACE.Sequences;
```

K.3.1.2 FACE.Sequences.Bounded Specification

```
generic
   type Element is private;
package FACE.Sequences.Bounded is

   type Sequence (Max_Length : Positive) is private;

   function Length (Source : in Sequence) return Natural;
   pragma Inline (Length);

   ------------------------------------------------------------
   -- Conversion, Concatenation, and Selection Functions --
   ------------------------------------------------------------

   function Append
     (Left, Right : in Sequence;
      Drop        : in Truncation := FACE.Sequences.Right)
      return Sequence;

   function Append
     (Left  : in Sequence;
      Right : in Element;
      Drop  : in Truncation := FACE.Sequences.Right)
      return Sequence;

   function Append
     (Left  : in Element;
      Right : in Sequence;
      Drop  : in Truncation := FACE.Sequences.Right)
      return Sequence;

   procedure Append
     (Source   : in out Sequence;
      New_Item : in Sequence;
      Drop     : in Truncation := FACE.Sequences.Right);

   procedure Append
     (Source   : in out Sequence;
      New_Item : in Element;
      Drop     : in Truncation := FACE.Sequences.Right);

   function "&"
     (Left, Right : in Sequence)
      return Sequence;

   function "&"
     (Left  : in Sequence;
      Right : in Element)
      return Sequence;

   function "&"
```

```
   (Left  : in Element;
    Right : in Sequence)
    return Sequence;

function Get_Element
   (Source : in Sequence;
    Index  : in Positive)
    return Element;

procedure Replace_Element
   (Source : in out Sequence;
    Index  : in Positive;
    By     : in Element);

function "=" (Left, Right : in Sequence) return Boolean;

----------------------------------------
-- Sequence transformation subprograms --
----------------------------------------

function Delete
   (Source  : in Sequence;
    From    : in Positive;
    Through : in Positive)
    return Sequence;

procedure Delete
   (Source  : in out Sequence;
    From    : in Positive;
    Through : in Positive);

-----------------------------------
-- Sequence selector subprograms --
-----------------------------------

function Head
   (Source : in Sequence;
    Count  : in Natural;
    Pad    : in Element;
    Drop   : in Truncation := FACE.Sequences.Right)
    return Sequence;

procedure Head
   (Source : in out Sequence;
    Count  : in Natural;
    Pad    : in Element;
    Drop   : in Truncation := FACE.Sequences.Right);

function Tail
   (Source : in Sequence;
    Count  : in Natural;
    Pad    : in Element;
    Drop   : in Truncation := FACE.Sequences.Right)
    return Sequence;

procedure Tail
   (Source : in out Sequence;
    Count  : in Natural;
    Pad    : in Element;
    Drop   : in Truncation := FACE.Sequences.Right);

--------------------------------------
-- Sequence constructor subprograms --
--------------------------------------

function "*"
   (Left  : in Natural;
    Right : in Element)
    return Sequence;

function "*"
```

```
      (Left  : in Natural;
       Right : in Sequence)
       return Sequence;

   function Replicate
     (Count : in Natural;
      Item  : in Element;
      Drop  : in Truncation := FACE.Sequences.Right)
      return Sequence;

   function Replicate
     (Count : in Natural;
      Item  : in Sequence;
      Drop  : in Truncation := FACE.Sequences.Right)
      return Sequence;


private
   -- implementation-defined
end FACE.Sequences.Bounded;
```

K.3.1.3 FACE.Sequences.Unbounded Specification

```
generic
   type Element is private;
package FACE.Sequences.Unbounded is

   subtype Index_Range  is Positive;
   subtype Length_Range is Natural;

   type Sequence is private;

   Null_Sequence : constant Sequence;

   function Length (Source : in Sequence) return Length_Range;
   pragma Inline (Length);

   ------------------------------------------------------------
   -- Conversion, Concatenation, and Selection Functions --
   ------------------------------------------------------------

   function Copy
     (Source : in Sequence)
      return Sequence;

   function To_Sequence
     (Length : in Length_Range)
      return Sequence;

   procedure Append
     (Source   : in out Sequence;
      New_Item : in Sequence);

   procedure Append
     (Source   : in out Sequence;
      New_Item : in Element);

   function "&"
     (Left, Right : in Sequence)
      return Sequence;

   function "&"
     (Left  : in Sequence;
      Right : in Element)
      return Sequence;

   function "&"
     (Left  : in Element;
      Right : in Sequence)
      return Sequence;
```

```
   function Get_Element
     (Source : in Sequence;
      Index  : in Index_Range)
      return Element;

   procedure Replace_Element
     (Source : in out Sequence;
      Index  : in Index_Range;
      By     : in Element);

   function "="
     (Left, Right : in Sequence) return Boolean;

   function Is_Null (Source : in Sequence) return Boolean;

   ----------------------------------------
   -- Sequence transformation subprograms --
   ----------------------------------------

   function Delete
     (Source  : in Sequence;
      From    : in Index_Range;
      Through : in Index_Range)
      return Sequence;

   procedure Delete
     (Source  : in out Sequence;
      From    : in Index_Range;
      Through : in Index_Range);

   -----------------------------------
   -- Sequence selector subprograms --
   -----------------------------------

   function Head
     (Source : in Sequence;
      Count  : in Length_Range;
      Pad    : in Element )
      return Sequence;

   procedure Head
     (Source : in out Sequence;
      Count  : in Length_Range;
      Pad    : in Element );

   function Tail
     (Source : in Sequence;
      Count  : in Length_Range;
      Pad    : in Element )
      return Sequence;

   procedure Tail
     (Source : in out Sequence;
      Count  : in Length_Range;
      Pad    : in Element );

   --------------------------------------
   -- Sequence constructor subprograms --
   --------------------------------------

   function "*"
     (Left  : in Length_Range;
      Right : in Element)
      return Sequence;

   function "*"
     (Left  : in Length_Range;
      Right : in Sequence)
      return Sequence;

private
```

```
  -- implementation-defined
end FACE.Sequences.Unbounded;
```

K.4 Java Programming Language

K.4.1 us.opengroup.FACE.Holder<T> Specification

```
package us.opengroup.FACE;

/**
 * @brief Class facilitating in and inout parameter passing of immutable types.
 * @param <T> The immutable type.
 */
public class Holder<T>{
  /** @brief Attribute containing the immutable type to be assigned a value. */
  public T value;

  /** @brief Default constructor */
  public Holder() {}

  /**
   * @brief Value constructor
   * @details Constructs a Holder of type T and initializes member attribute
   * value with the parameter passed in.
   */
  public Holder(T initial) {value = initial;}
}
```

K.4.2 us.opengroup.FACE.BAD_PARAM Specification

```
package us.opengroup.FACE;

/** @brief Exception thrown when a parameter violates the IDL semantics. */
public class BAD_PARAM extends Exception
{
  /** @brief Constructs a BAD_PARAM exception with a default reason. */
  public BAD_PARAM()
  {
    super("");
  }

  /** @brief Constructs a BAD_PARAM exception with the specified reason. */
  public BAD_PARAM(String reason)
  {
    super(reason);
  }
}
```

K.4.3 us.opengroup.FACE.DATA_CONVERSION Specification

```
package us.opengroup.FACE;
/** @brief Exception thrown when a parameter's data violates the IDL definition of the
data type when converted. */
public class DATA_CONVERSION extends Exception
{
  /** @brief Constructs a DATA_CONVERSION exception with a default reason. */
  public DATA_CONVERSION()
  {
    super("");
  }

  /** @brief Constructs a DATA_CONVERSION exception with the specified reason. */
  public DATA_CONVERSION(String reason)
  {
    super(reason);
  }
}
```

L Acronyms

Acronym	Description
2D	Two-Dimensional
3D	Three-Dimensional
AMRDEC	Aviation and Missile Research, Development, and Engineering Center
ANSI	American National Standards Institute
API	Application Programming Interface
ARINC	Aeronautical Radio Inc.
ARP	Aerospace Recommended Practice
BAE	British Aerospace
BBC	Backup Bus Controller
BC	Bus Controller
BM	Bus Monitor
BSP	Board Support Package
CALIPSO	Common Architecture Label Internet Protocol Version 6 Security Option
CCB	Configuration Control Board
CDM	Conceptual Data Model
CDS	Cockpit Display System
CMC	Canadian Marconi Company
COE	Common Operating Environment
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off-The-Shelf
CPI	Critical Program Information
CPU	Central Processing Unit
CSP	Component State Persistence
DDS	Data Distribution Service
DF	Definition File

Acronym	Description
DITS	Digital Information Transfer System
DO	Document
DoD	Department of Defense
DPM	Device Protocol Mediation
DSDM	Domain-Specific Data Model
EGL	Embedded Graphics Library
EMOF	Essential Meta-Object Facility
ES	Embedded Systems
FACE	Future Airborne Capability Environment
FM	Fault Management
FTP	File Transfer Protocol
FSC	Framework Support Capability
GPS	Global Positioning System
GPU	Graphics Processing Unit
GUID	Globally Unique Identifier
HM	Health Monitor
HMFM	Health Monitoring and Fault Management
HMI	Human Machine Interface
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
I/O	Input/Output
ICD	Interface Control Document
ID	Identification
IDL	Interface Definition Language
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IIOP	Internet Inter-ORB Protocol
IOS	Input/Output Services
IP	Internet Protocol
IPC	Inter-Process Communication

Acronym	Description
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
ISO/IEC	International Organization for Standardization/International Electrotechnical Commission
IT	Information Technology
LCM	Life Cycle Management
Java EE	Java Enterprise Edition
Java SE	Java Standard Edition
LDM	Logical Data Model
MCDU	Multi-purpose Control Display Unit
MIL-STD	Military Standard
MISRA	Motor Industry Software Reliability Association
MOF	Meta-Object Facility
MOSA	Modular Open Systems Approach
MPEG	Moving Picture Experts Group
MSG	Message
NAVAIR	Naval Air Systems Command
NIST	National Institute of Standards and Technology
NSA	National Security Agency
OCL	Object Constraint Language
OFP	Operation Flight Program
OMG	Object Management Group
OpenGL	Open Graphics Language
OS	Operating System
OSGi	Open Services Gateway initiative
PCS	Portable Components Segment
PDM	Platform Data Model
POSIX	Portable Operating System Interface
PSCS	Platform-Specific Common Services
PSDS	Platform-Specific Device Services

Acronym	Description
PSGS	Platform-Specific Graphics Services
PSSS	Platform-Specific Services Segment
QoS	Quality of Service
RFC	Request for Comments
RMF	Risk Management Framework
RS	Recommended Standard
RT	Run-Time (Language)
RT	Remote Terminal
RTI	Real Time Innovations
RTPS	Real Time Publish Subscribe
RTSC	Real Time Safety-Critical
SFTP	Secure File Transfer Protocol
SW	Software
SA	Subaddress
SC	Safety-Critical
SDM	Shared Data Model
SMPTE	Society of Motion Picture and Television Engineers
SNMP	Simple Network Management Protocol
STL	Standard Template Libraries
TCP	Transmission Control Protocol
TPM	Transport Protocol Module
TS	Transport Services
TSS	Transport Services Segment
TWG	Technical Working Group
UA	User Application
UDP	User Datagram Protocol
UoC	Unit of Conformance
UoP	Unit of Portability
U.S.	United States
USM	Unit of Portability Supplied Model

Acronym	Description
UUID	Universally Unique Identifier
W3C	World Wide Web Consortium
XMI	Extensible Markup Language Metadata Interchange
XML	Extensible Markup Language
XSD	Extensible Markup Language Schema Definition

Index