Date: February 2020

Space Telecommunication Interface

Version 0

OMG Document Number: mars/20-01-01

Normative reference: https://www.omg.org/spec/sti/1.0/

Machine readable file(s): https://www.omg.org/sti/20200101

Normative: https://www.omg.org/spec/acronym/20200101/foo.xmi

Informative: https://www.omg.org/spec/acronym/20200101/non_normative_foo.xmi

---

Object Management Group

109 Highland Avenue
Needham, MA 02494
USA

Telephone: +1-781-444-0404
Facsimile: +1-781-444-0320
rfp@omg.org

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

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

LICENSES

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

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

PATENTS

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

GENERAL USE RESTRICTIONS

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

DISCLAIMER OF WARRANTY

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

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

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c) (1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 109 Highland Avenue, Needham, MA 02494, U.S.A.

TRADEMARKS

CORBA®, CORBA logos®, FIBO®, Financial Industry Business Ontology®, FINANCIAL INSTRUMENT GLOBAL IDENTIFIER®, IIOP®, IMM®, Model Driven Architecture®, MDA®, Object Management Group®, OMG®, OMG Logo®, SoaML®, SOAML®, SysML®, UAF®, Unified Modeling Language®, UML®, UML Cube Logo®, VSIPL®, and XMI® are registered trademarks of the Object Management Group, Inc.

For a complete list of trademarks, see: https://www.omg.org/legal/tm_list.htm. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

COMPLIANCE

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

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

OMG's Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page http://www.omg.org, under Documents, Report a Bug/Issue.

Table of Contents

0 Submission-Specific Material....vi
0.1 Submission Preface....vi
0.2 Copyright Waiver....vi
0.3 Submitter Representative....vi
0.4 Author Team....vi
0.5 Proof of Concept....vi
0.6 Submission Contents....vii
0.7 Applicability to RFP....vii
1 Scope....1
2 Conformance....1
3 References....1
3.1 Normative References....1
3.2 Non-normative References....2
4 Terms and Definitions....2
5 Symbols....3
6 Additional Information....6
6.1 Acknowledgments....6
7 Specification....7
7.1 Overview....7
7.2 Purpose....7
7.3 Key Architecture Requirements....7
7.4 Fundamental Design....8
7.5 Roles and Responsibilities....9
8 Hardware Architecture....12
8.1 Generalized Hardware Architecture....12
8.2 Module Specification....17
8.3 Hardware Interface Description....25
9 Application Architecture....28
9.1 Configurable Hardware Design....28
9.2 Specialized Hardware Interfaces....29
10 Software Architecture....32
10.1 Software Layer Model....32
10.2 Infrastructure....37
10.3 API Overview....38
10.4 Data Types and Constants....42
10.5 Application and Device Control Interface....48
10.6 STI API....61
10.7 Non-STI Software Interfaces....94
11 External Command and Telemetry Interfaces....98
Annex A: Language Translations....100
A.1 C Language Mapping....100
A.2 C++ Language Mapping....102
A.3 Python Mapping....104
Annex B: Conformance Table....107

Index of Tables

Table 1: Module Interface Characterization....26
Table 2: Example Operating Power Interface....27
Table 3: Software Component Descriptions....37
Table 4: Platform-Specific Naming Conventions....41
Table 5: Infrastructure-provided Data Types....43
Table 6: Access Constants....44
Table 7: CalendarKind Constants....44
Table 8: HandleID Constants....45
Table 9: Result Constants....46
Table 10: Handle Name Constants....46
Table 11: Property Name Constants....46
Table 12: Size Limit Constants....47
Table 13: TimeWarp Constants....47
Table 14: APP_GetHandleID() Definition....48
Table 15: APP_GetHandleName() Definition....49
Table 16: APP_Instance() Definition....50
Table 17: APP_Destroy() Definition....50
Table 18: APP_Initialize() Definition....51
Table 19: APP_ReleaseObject() Definition....51
Table 20: APP_Query() Definition....52
Table 21: APP_Configure() Definition....52
Table 22: APP_RunTest() Definition....53
Table 23: APP_Start() Definition....54
Table 24: APP_Stop() Definition....54
Table 25: DEV_Open() Definition....55
Table 26: DEV_Load() Definition....55
Table 27: DEV_Reset() Definition....56
Table 28: DEV_Flush() Definition....56
Table 29: DEV_Unload() Definition....57
Table 30: DEV_Close() Definition....57
Table 31: APP_Read() Definition....58
Table 32: APP_Write() Definition....59
Table 33: APP_AddressRead() Definition....60
Table 34: APP_AddressWrite() Definition....61
Table 35: IsOK() Definition....62
Table 36: ValidateHandleID() Definition....63
Table 37: ValidateSize() Definition....63
Table 38: GetErrorQueue() Definition....63
Table 39: GetHandleName() Definition....64
Table 40: HandleRequest() Definition....64
Table 41: InstantiateApp() Definition....66
Table 42: AbortApp() Definition....67
Table 43: Initialize() Definition....68
Table 44: Release() Definition....68
Table 45: Configure() Definition....69
Table 46: Query() Definition....69
Table 47: RunTest() Definition....70
Table 48: Start() Definition....70
Table 49: Stop() Definition....71

Table 50: DeviceOpen() Definition....71
Table 51: DeviceLoad() Definition....71
Table 52: DeviceReset() Definition....72
Table 53: DeviceFlush() Definition....72
Table 54: DeviceUnload() Definition....72
Table 55: DeviceClose() Definition....73
Table 56: Read() Definition....74
Table 57: Write() Definition....74
Table 58: AddressRead() Definition....75
Table 59: AddressWrite() Definition....76
Table 60: Log() Definition....76
Table 61: FileOpen() Definition....77
Table 62: FileClose() Definition....78
Table 63: FileGetSize() Definition....78
Table 64: FileRemove() Definition....79
Table 65: FileRename() Definition....79
Table 66: FileGetFreeSpace() Definition....80
Table 67: MessageQueueCreate() Definition....81
Table 68: MessageQueueDelete() Definition....81
Table 69: PubSubCreate() Definition....82
Table 70: PubSubDelete() Definition....82
Table 71: Register() Definition....83
Table 72: Unregister() Definition....83
Table 73: GetNanoseconds() Definition....84
Table 74: GetSeconds() Definition....85
Table 75: GetTimeWarp() Definition....85
Table 76: TimeAdd() Definition....85
Table 77: TimeSubtract() Definition....86
Table 78: GetTime() Definition....86
Table 79: SetTime() Definition....87
Table 80: GetCalendarTime() Definition....88
Table 81: CalendarValueCivil Structure Definition....89
Table 82: CalendarValueGPS Structure Definition....89
Table 83: CalendarValueDayNumber Structure Definition....90
Table 84: CalendarTime Union Definition....90
Table 85: SetTimeAdjust() Definition....91
Table 86: GetTimeAdjust() Definition....92
Table 87: TimeSynch() Definition....92
Table 88: Sleep() Definition....93
Table 89: DelayUntil() Definition....94
Table 90: Function Alternatives....97
Table 91: C Language Header Files....101
Table 92: C Language Data Type Mapping....101
Table 93: C++ Language Header Files....103
Table 94: C++ Language Data Type Mapping....104
Table 95: Python Language Data Type Mapping....106

Figures

Figure 1: Roles and Responsibilities....10
Figure 2: Hardware Architecture Diagram Key....12
Figure 3: Notional STI Hardware Architecture....13
Figure 4: GPM Architecture Details....17
Figure 5: SPM Architecture Details....20
Figure 6: RFM Architecture Details....23
Figure 7: Waveform Component Distribution....28
Figure 8: Notional High Level Software and Configurable Hardware Design....30
Figure 9: Software Execution Model....33
Figure 10: Layered Structure in UML....34
Figure 11: Operating Environment....35
Figure 12: Standards-Compliant vs Standards-Conformant OS....37
Figure 13: Infrastructure....38
Figure 14: Application and Device Structure....39
Figure 15: Sequence Diagram for Application Control....65
Figure 16: Sequence Diagram for InstantiateApp....66
Figure 17: Sequence Diagram for AbortApp....67
Figure 18: Calendar Time Value Representations....88
Figure 19: Profile Building Blocks....96
Figure 20: Command and Telemetry Interfaces....98

Preface

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable, and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies, and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at http://www.omg.org.

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. All OMG Specifications are available from the OMG website at:

http://www.omg.org/spec

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
109 Highland Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: pubs@omg.org

Certain OMG specifications are also available as ISO standards. Please consult http://www.iso.org

0 Submission-Specific Material

0.1 Submission Preface

This submission is in response to the Space Telecommunication Interface (STI) RFP, document number mars/19-09-21.

The proposal describes an architecture for software-defined radios (SDRs) based on existing NASA standards. The proposed Space Telecommunication Interface provides a common, consistent framework to abstract the application software from the platform hardware to increase portability and reduce the cost and risk of using complex reconfigurable and reprogrammable radio interfaces across environments from Earth to outer space.

0.2 Copyright Waiver

0.3 Submitter Representative

Janette C. Briones, PhD.
Louis M. Handler
NASA Glenn Research Center, Cleveland, OH

Joseph P. Hickey
William T. Dark
Vantage Partners, LLC, Brook Park, OH

Jeffrey Smith, PhD.
Sierra Nevada Corporation, Herndon, VA

0.4 Author Team

Janette C. Briones, PhD.
Louis M. Handler
NASA Glenn Research Center, Cleveland, OH

Joseph P. Hickey
William T. Dark
Vantage Partners, LLC, Brook Park, OH

Jeffrey Smith, PhD.
Sierra Nevada Corporation, Herndon, VA

0.5 Proof of Concept

The foundation of this proposal evolved from a National Aeronautics and Space Administration (NASA) project whose goal was to improve the portability of components utilized in software defined radio (SDR) deployments. The intent is to improve the return on investment in software development by allowing the related components to be deployed in more than one project/mission without incurring significant additional development time.

A predecessor to this STI specification was developed by NASA as part of a technology demonstration of software-defined radio technology. The use of SDRs for NASA missions was a new concept in 2002, made possible by the development of reconfigurable components suitable for use in space radios. A need to reduce the cost and risk of using SDRs was identified and the development of a common SDR architecture was initiated as a means to achieve this.

In 2007, the architecture was determined to be ready for flight implementation in a technology development project. This project was originally called the Communication, Navigation, and Networking reConfigurable Testbed (CoNNeCT), and later renamed the Space Communications and Navigation (SCaN) Testbed. Three SDRs were procured in 2008 and 2009 for the SCaN Testbed, using the architecture defined in a technical memorandum and referred to in the procurement specifications as Space Telecommunications Radio System version 1.02.1.

The SCaN Testbed was launched in July 2012 and operates on an external truss on the International Space Station (ISS). The SCaN Testbed is an experimental communications system that provides the capability for S-Band, Ka-Band, and L-Band communication with space and ground assets. Investigation of SDR technology and the STI architecture are the primary focus of the SCaN Testbed. As a completely reconfigurable testbed, the SCaN Testbed provides experimenters an opportunity to develop and demonstrate experimental waveforms and applications for communication, networking, and navigation concepts and to advance the understanding of operating SDRs in space.

Lessons learned from the SDR platform provider, application developers, and integrators of the SCaN Testbed provided critical insight for the development of future versions of the Space Telecommunications Radio System (STRS), which was released as NASA standard 4009. The most recent revision to the NASA standard, designated as 4009A, serves as the basis for this STI specification.

As part of this effort, NASA has developed a reference implementation of this architecture as a C/C++ library. Additionally, NASA has deployed several complete STRS operating environments on different radio platforms from different vendors, and maintains a library of portable applications that are compliant with the architecture.

0.6 Submission Contents

TODO: Fill this in

0.7 Applicability to RFP

It addresses the issues presented in the STI RFP in the following ways:

- This proposal submission is based on NASA STRS, which is an architecture for SDRs that has been deployed and used in space flight applications.
- STRS has been successfully implemented on space-grade, radiation tolerant hardware which has limited computing resources (memory capacity, processing power, etc.). It has relatively small footprint compared to other software radio architectures, and minimal external software dependencies.
- This proposal does not require a specific middleware to exist between components, permitting these interfaces these interfaces to be a direct path (e.g. a function call) with minimal overhead. However, the model also does not exclude the use of middleware/remote procedure calls where warranted, so it has the wide applicability to a variety of system footprints.
- It facilitates re-use of investments in signal processing software and FPGA designs for software defined radios through abstraction layers.
- It contains provisions for independent unit testing of the SDR components in isolation.

- It allows for independent updates/lifecycles of each SDR component, allowing the radio functionality to evolve over time.
- It allows for components to be developed in parallel by different vendors, and facilitates vendor independence.
- It provides a common, abstract interface for higher level software applications to communicate with the radio, such as cognitive link optimization algorithms.
- It permits the interfaces to be accessed via an external Remote Procedure Call (RPC) mechanism, where warranted, to provide an external command/control interface for the radio.
- The relationship to NASA STRS means that existing software/programmable logic already developed in compliance with STRS will be portable to STI with minimal changes, thereby offering an upgrade path for existing products.

The general system/hardware architecture is prescribed in section 8, Hardware Architecture, and the software API is documented in section 10, Software Architecture.

This page intentionally left blank.

1 Scope

This document, the Space Telecommunication Interface (STI), specifies the data types, application programming interface, and associated operational patterns that compliant software defined radio (SDR) platforms are required to implement. This is intended to promote portability of SDR applications between radio platform providers by providing a common programming interface.

In order to be adaptable to a wide variety of platforms and applications, this specification focuses on a metamodel for the hardware and software architecture of an SDR, rather than prescribing a specific implementation. As such, an adequate level of knowledge capture must be documented to facilitate portability and reuse of hardware and software architecture.

2 Conformance

The primary point of conformance is support of the given platform independent model (PIM). This specification concerns multiple aspects of SDRs, with different specific points of conformance for each aspect. For hardware architecture, conformance is indicated through a hardware interface document, which specifies how the PIM is realized in a given design. For the software architecture, conformance is based on the implementation and usage of the various software interfaces prescribed in this document.

A complete conformance and requirements matrix is included in appendix 11, Conformance Table.

3 References

3.1 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply.

Object Management Group (OMG):

IDL	Interface Definition Language Specification
CPP	C++ Language Mapping Specification
PYTH	Python Language Mapping Specification
UML	Unified Modeling Language Specification

Institute of Electrical and Electronics Engineers (IEEE):

1003.13	IEEE Standard for Information Technology—Standardized Application Environment Profile (AEP)—POSIX® Realtime and Embedded Application Support

International Organization for Standardization (ISO):

9899	Information technology—Programming languages—C
9945	Information technology—Portable Operating System Interface (POSIX®) Base Specifications
14882	Information technology—Programming languages—C++

3.2 Non-normative References

The following documents provide additional guidelines, historical context or rationale for elements of this specification.

Object Management Group (OMG):

ORMSC/14-06-01	Model Driven Architecture (MDA) Guide

National Aeronautics and Space Administration (NASA):

NASA-STD-4009A	Space Telecommunications Radio Systems (STRS) Architecture Standard
NASA-HDBK-4009A	Space Telecommunications Radio Systems (STRS) Architecture Standard Rationale
NASA/TM—2007-215042	Space Telecommunications Radio System (STRS) Architecture Goals/Objectives and Level 1 Requirements
NASA/TP—2008-214813	Space Telecommunications Radio System Software Architecture Concepts and Analysis

United States Department of Defense:

MIL-STD-1553	Digital Time Division Command/Response Multiplex Data Bus
SCA	Software Communications Architecture Specification, Version 2.2.2

4 Terms and Definitions

For the purposes of this specification, the following terms and definitions apply.

Component

A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. A component exposes a set of provided and required interfaces that specify the component behavior and operation.

Facility

The realization of certain functionality through a set of well defined interfaces.

Logical Device

A software component that is an abstraction of a hardware device it represents.

Mapping

The specification of a mechanism for transforming the elements of a model conforming to a particular metamodel into elements of another model that conforms to another (possibly the same) metamodel.

Metamodel

A model of models.

Model

A formal specification of the function, structure and/or behavior of an application or system.

Model Driven Architecture (MDA)

An approach to IT system specification that separates the specification of functionality from the specification of the implementation of that functionality on a specific technology platform.

Platform

A set of subsystems or technologies that provide a coherent set of functionality through interfaces and specified usage patterns.

Platform Independent Model (PIM)

A model of a subsystem that contains no information specific to the platform, or the technology that is used to realize it.

Platform Specific Model (PSM)

A model of a subsystem that includes information about the specific technology that is used in the realization of it on a specific platform, and hence possibly contains elements that are specific to the platform.

Radio Platform

The Radio Platform is a platform that provides radio functionality.

Service

A software program that provides functionality available for use by other applications.

5 Symbols

The following acronyms and abbreviations are used in this document

A	Ampere
ADC	Analog-to-Digital Converter
AEP	Application Environment Profile
AGC	Automatic Gain Control

ANSI	American National Standards Institute
API	Application Programming Interface
APP	Application
ASCII	American Standard Code for Information Interchange
ASIC	Application-Specific Integrated Circuit
BIT	Built-in Test
BSP	Board Support Package
C&DH	Command and Data Handling
CCSDS	Consultative Committee for Space Data Systems
COTS	Commercial Off the Shelf
DAC	Digital-to-Analog Converter
DEC	Digital Equipment Corporation
DLL	Dynamic Link Library
DSP	Digital Signal Processor
EDIF	Electronic Design Interchange Format
EEPROM	Electrically Erasable, Programmable Read-Only Memory
FFRDC	Federally Funded Research and Development Center
FIFO	First In, First Out
FIPS	Federal Information Processing Standard
FPGA	Field Programmable Gate Array
GPIO	General Purpose Input Output
GPM	General Purpose Processing Module
GPP	General Purpose Processor
GPS	Global Positioning System
HAL	Hardware Abstraction Layer
HDBK	Handbook
HDL	Hardware Description Language
HID	Hardware Interface Description
HW	Hardware
I/O	Input/Output
I^2C	Inter-Integrated Circuit
ID	Identification, Identifier
IEC	International Electrotechnical Commission
IEEE	The Institute of Electrical and Electronics Engineers
IF	Intermediate Frequency
INCITS	Inter-National Committee for Information Technology Standards
IP	Internet Protocol
ISO	International Organization for Standardization
ISR	Interrupt Service Routine
LLC	Logical Link Control
LNA	Low Noise Amplifier
MAC	Media Access Control
MIL	Military
MMU	Memory Management Unit

NASA	National Aeronautics and Space Administration
NM	Network Module
OAL	OEM adaptation layer
OE	Operating Environment
OEM	Original Equipment Manufacturer
OM	Optical Module
OMG	Object Management Group
ORMSC	Operational Research MSc Programmes
OS	Operating System
OSS	Open Source Software
PIM	Platform-Independent Model
POSIX®	Portable Operating System Interface
PROM	Programmable Read-Only Memory
RAM	Random Access Memory
RF	Radio Frequency
RFM	Radio Frequency Module
ROM	Read-Only Memory
RTOS	Real-Time Operating System
SCA	Software Communications Architecture
SDR	Software-Defined Radio
SEC	Security Module
SEU	Single Event Upset
SPM	Signal Processing Module
SRAM	Static Random Access Memory
STD	Standard
STI	Space Telecommunication Interface
TCP	Transmission Control Protocol
TMR	Triple-Mode Redundancy
TT&C	Telemetry, tracking, and command
UML	Unified Modeling Language
V	Volt
V&V	Verification and Validation
VDD	Version Description Document
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
XML	Extensible Markup Language

6 Additional Information

6.1 Acknowledgments

The following companies and individuals contributed to the development of this specification:

Principal Authors of NASA-STD-4009A (STRS)

Richard C. Reinhart, Thomas J. Kacpura, Louis M. Handler, Sandra K. Johnson, Janette C. Briones, Jennifer M. Nappier, and Joseph A. Downey
NASA Glenn Research Center, Cleveland, OH

C. Steve Hall
Analex Corporation, Cleveland, OH

Dale J. Mortensen
ASRC Aerospace Corporation, Cleveland, OH

James P. Lux
Jet Propulsion Laboratory, Pasadena, CA

Key Industry Participants

Carl Smith, John Liebetreu
General Dynamics Corporation, C4-I

Vince Kovarik
Harris Corporation, Melbourne, FL

Mark Scoville
L-3 Communications, Salt Lake City, UT

Jerry Bickle
Prism Tech, Woburn, MA

Key Reviewers and Contributors

David J. Israel
NASA Goddard Space Flight Center, Greenbelt, MD

Allen Farrington, Yong Chong, Kenneth J. Peters
Jet Propulsion Laboratory, Pasadena, CA

Andrew L. Benjamin
NASA Johnson Space Center, Houston, TX

Eric A. Eberly, Terry M. Luttrell
NASA Marshall Space Flight Center, Huntsville, AL

SDR Forum Contributing Member Companies

General Dynamics
Prism Tech
Boeing Corporation
Cincinnati Electronics

Harris Corporation
L-3 Communications
Lockheed Martin

7 Specification

7.1 Overview

This specification describes the Space Telecommunication Interface (STI) architecture for software-defined radios (SDRs). STI provides a common, consistent framework to abstract the application software from the platform hardware to reduce the cost and risk of using complex reconfigurable and reprogrammable radio interfaces across different space and satellite projects. It achieves this objective by defining an architecture to enable the reuse of applications (waveforms and services implemented on the SDR) across heterogeneous SDR platforms and thereby reduces dependence on a single vendor or platform type.

The specification provides a detailed description and set of requirements to implement the architecture. The specification focuses on the key components and facilities by prescribing their functionality and interfaces for both the hardware and the software. The intended audience for this specification is composed of software and hardware developers who need architecture specification details to develop an STI platform or application.

7.2 Purpose

The purpose of this specification is to establish an open architecture specification for space and ground SDRs. Many space projects either use hardware radios, which cannot be modified once deployed, or software-defined radios with an architecture that depends on the radio provider and involves significant effort to add new applications.

This specification is intended to assist in the development of software-defined, reconfigurable technology to meet future space communications and navigation system needs. Software-based SDRs enable advanced operations that potentially reduce mission life-cycle costs for space or ground platforms. Since SDR technology allows radios to be reconfigured to perform different functions, it may reduce the number of discrete radio devices required to achieve desired objectives, which also decreases mass and power requirements for the overall system.

7.3 Key Architecture Requirements

The key goals in the development of the STI architecture are to decrease the development time, cost, and risk of using SDRs while still accommodating advances in technology. The advent of software-based applications allows minimal rework to reuse applications and to adapt to evolving requirements.

The requirements for the architecture are derived from the following STI goals and objectives:

- Usable across most space project types (scalability and flexibility).
- Decrease development time and cost.
- Increase reliability of SDRs.
- Accommodate advances in technology with minimal rework (extensibility).
- Adaptable to evolving requirements (adaptability).
- Leverage existing or developing standards, resources, and experience (state-of-the-art and state-of-practices).
- Maintain vendor independence.
- Enhance waveform application portability and re-usability.

Conversely, the architecture does **not** specify mission-specific functional and performance requirements such as:

- Any specific hardware
- Contents or format of the external interfaces to the SDR
- Waveform-specific requirements such as data rate, coding scheme, and modulation and demodulation techniques.
- Security, fault tolerance, redundancy, and fault mitigation approaches.

Instead, the architecture is careful to enable all solutions that the project might require as they relate to the mission-specific functional and performance specifications. The architecture does not preclude the implementation of mission-developed services on the SDR, including but not limited to:

- Multiple waveforms operating simultaneously across any RF band defined in the SDR specification.
- Commanded built-in-test (BIT) and status reporting.
- Real-time operational diagnostics.
- Automated system recovery and initialization.
- Networking and navigation within the SDR.
- Secure transmission.
- Shared processing among on-board elements.

To meet these goals and objectives, the STI architecture has an open architecture design that accommodates a varying range of radio form factors.

7.4 Fundamental Design

This STI Standard consists of hardware, configurable hardware design, and software architectures with accompanying description, guidance, and requirements.

The terms “software” and “configurable hardware design” are used in this specification to distinguish the architecture items that apply to code (source code, object code, executables, etc.) implemented on a processor; and designs (hardware description language/HDL source, loadable files, data tables, etc.) implemented in a configurable hardware device such as a field programmable gate array (FPGA). Both items can change the functionality of the radio in-situ using program control. The term “software” is also used in a generic sense in this specification to discuss all configurable items of the radio, including configurable hardware design. The terminology used is not meant to imply design and implementation process.

The STI hardware architecture is specified at a facility level. The hardware architecture requirements are written so that the hardware provider defines the functional breakdown (modules or components) of the system and publishes the functions and interfaces for each module and for the entire platform in a hardware interface description (HID) document. This information enables others developing applications or additional modules, or interfacing to the platform, to have the knowledge to integrate and test the hardware interfaces and understand the features and limitations of the platform. This specification encourages the development of applications that are modular, portable, reconfigurable, and reusable.

The software architecture is the focus of this STI Standard. STI applications use the STI infrastructure-provided application program interfaces (APIs) and services to load, verify, execute, change parameters, terminate, or unload an application. The software architectural model describes the relationship between the software elements, defined in layers, in an STI-compliant radio. The model illustrates the different software elements used in the software

execution and defines the API layers between an STI application and the Operational Environment (OE), and between the OE and the hardware platform.

The STI software layers are separated to enable developers to implement the software layers differently according to their requirements while still complying with the STI architecture. A key aspect is the abstraction of the STI application, which is either a waveform or service, from the underlying OE software to promote portability and reusability of the STI application. Interfaces in STI software architecture can be divided into three general categories, as follows:

- The STI APIs, defined in this document, and the application-specific data structures associated with these APIs.
- The operating system interface, such as POSIX®.
- The interface to external software modules, libraries or dependencies, such as third-party signal processing software, mathematical toolkits, or an interface to any application-specific hardware.

The STI APIs provide the interfaces that allow applications to be instantiated and use platform services. These APIs also enable communication between STI applications and the STI infrastructure. The hardware abstraction layer (HAL) provides a software view of the specialized hardware by abstracting the physical hardware of interfaces. It is to be published so that software and configurable hardware design running on the platform's specialized hardware can integrate with the STI infrastructure.

7.5 Roles and Responsibilities

The final configuration of an SDR and its applications is generally a product of multiple organizations performing various tasks. The separation of requirements, responsibilities, and resulting tasks is assigned in this specification by logical role where each role has requirements that may be satisfied by an individual or delegated to a subordinate organization(s). As figure 1, Roles and Responsibilities, illustrates, the effort begins with a mission need for a radio, which could support communications, navigation, and in some instances even networking functions. The mission system engineer defines radio interface requirements. For each mission, the system integrators, platform providers, and application developers are selected. Eventually, the platform and applications are integrated into the STI-compliant radio product. Both the hardware and software are tailored to meet mission-specific needs.

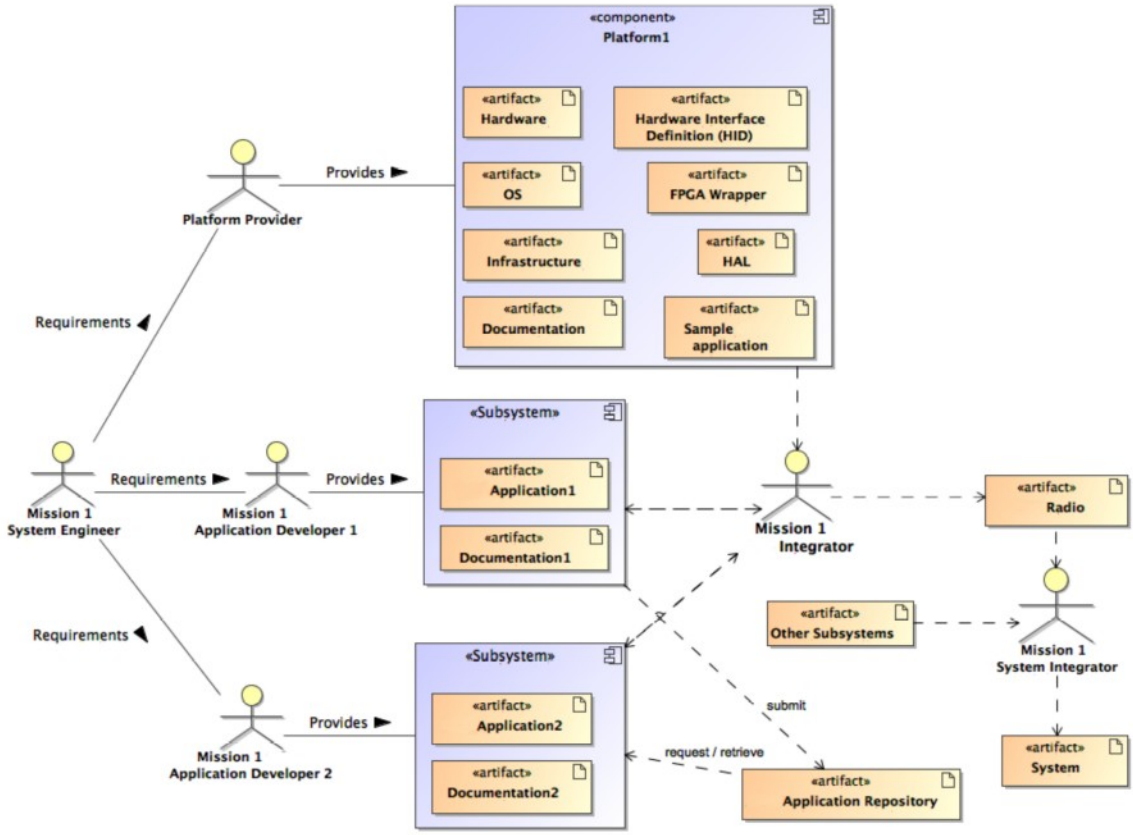

Figure 1: Roles and Responsibilities

The STI platform provider is the organization responsible for the design and development of the SDR hardware platform, including the STI OE (e.g., infrastructure, OS), and associated documentation. The OE and hardware platform are a unique set and become the SDR platform.

The STI platform provider is responsible for following:

- All documentation associated with the platform.
- Any platform-specific FPGA wrapper for adaptation of FPGA code to the platform
- Software header files specifying the required interface, including constants, type definitions, and structures.
- Script or software configuration file formats, any extensible markup language (XML) schema, and any transformation tool for controlling instantiation, and their associated documentation, if deemed necessary.

If the STI platform provider delegates responsibility for part of the OE to a separate infrastructure provider, the responsibility for the appropriate files and documentation may be delegated to that provider as well. If the STI platform provider delegates responsibility for part of the hardware to a separate hardware provider, the responsibility for the pertinent HID documentation may be delegated to that hardware provider as well. The STI platform provider is ultimately responsible to integrate and deliver all aspects of the platform and OE documentation.

A primary objective of STI is facilitate the re-use of SDR components, and as such, one or more repositories containing existing, previously-developed STI components may be available for project development efforts. Any such components may be publicly available and distributed under an open-source license or a

commercial/proprietary license, or may be held in a private, non-public repository that is maintained internally within the same organization.

The project design team and the STI application developer have the responsibility to evaluate the contents of any available component repositories against the SDR application requirements to determine if an existing application in a repository may be re-used by porting it to the target platform. Depending on the results of this decision, the STI application developer either creates a new application or ports an existing STI application. The STI application developer performs unit tests, and documents the functionality.

The STI integrator brings the hardware platform and software application together on the SDR platform. The STI integrator could be the STI platform provider, the STI application developer(s), a mission engineer, or even a third party. The STI integrator's role is to have the application properly running on the SDR platform to meet the communication, navigation, or other functions of the mission. Once the STI radio integration is complete, it is delivered to a system integrator who incorporates it into the mission spacecraft system.

8 Hardware Architecture

In addition to providing benefits by defining a standard software infrastructure for software defined radios, this specification also defines standards for the hardware portion of the radio. Hardware technologies may change more rapidly than software, and each radio implementation generally has very specific spacecraft dependencies and requirements. Therefore, the STI hardware architecture is specified as an abstract set of facilities level rather than at the physical implementation level.

The architecture does not prescribe a specific hardware implementation approach. An STI hardware platform is to be delivered with a complete HID, which is described in section 8.3, Hardware Interface Description. The HID specifies the electrical interfaces, logic interfaces, connector requirements, and physical requirements for the delivered radio. Each module's HID abstracts and defines the module functionality and performance.

8.1 Generalized Hardware Architecture

Figure 2, Hardware Architecture Diagram Key, illustrates the symbols and terminology used within the hardware architecture diagrams. The hardware diagram illustrates the radio functions and the interconnects for each module.

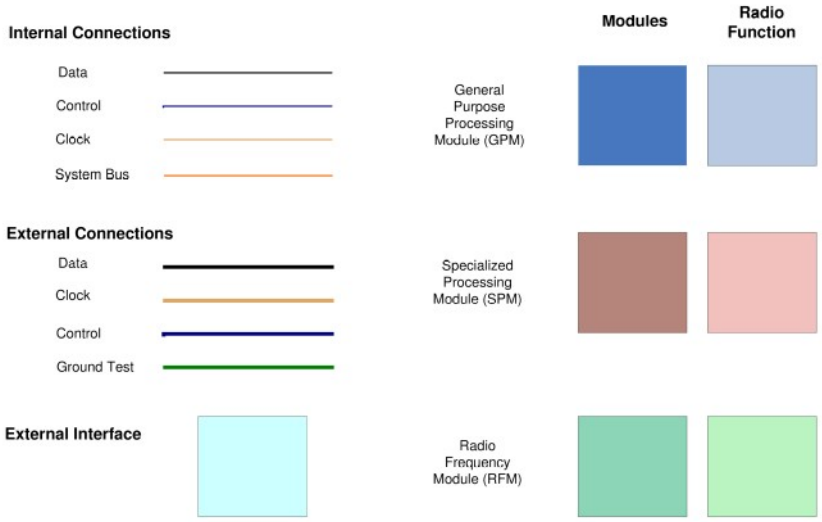

Figure 2: Hardware Architecture Diagram Key

The modules are a logical and functional division of common radio functions that comprise an STI platform. Modules are not intended to represent physical entities of the platform. As developers choose how to distribute and implement the radio functions among hardware elements, the specification provides the guidance on the interfaces and abstractions that are to be provided to comply with the architecture. The module and function connections provided in the diagrams are data path, control, signal clock, and external interfaces.

Figure 3, Notional STI Hardware Architecture, shows the high-level STI hardware architecture. The figure illustrates the functional attributes and interfaces for each module. A module is a combination of logical and functional representations of platform and applications implemented in a radio. The modules are divided into their typical functions to provide a common description and terminology reference. Each STI platform provider has the flexibility to combine these modules and their functionality as necessary during the radio design process to meet the specific mission requirements.

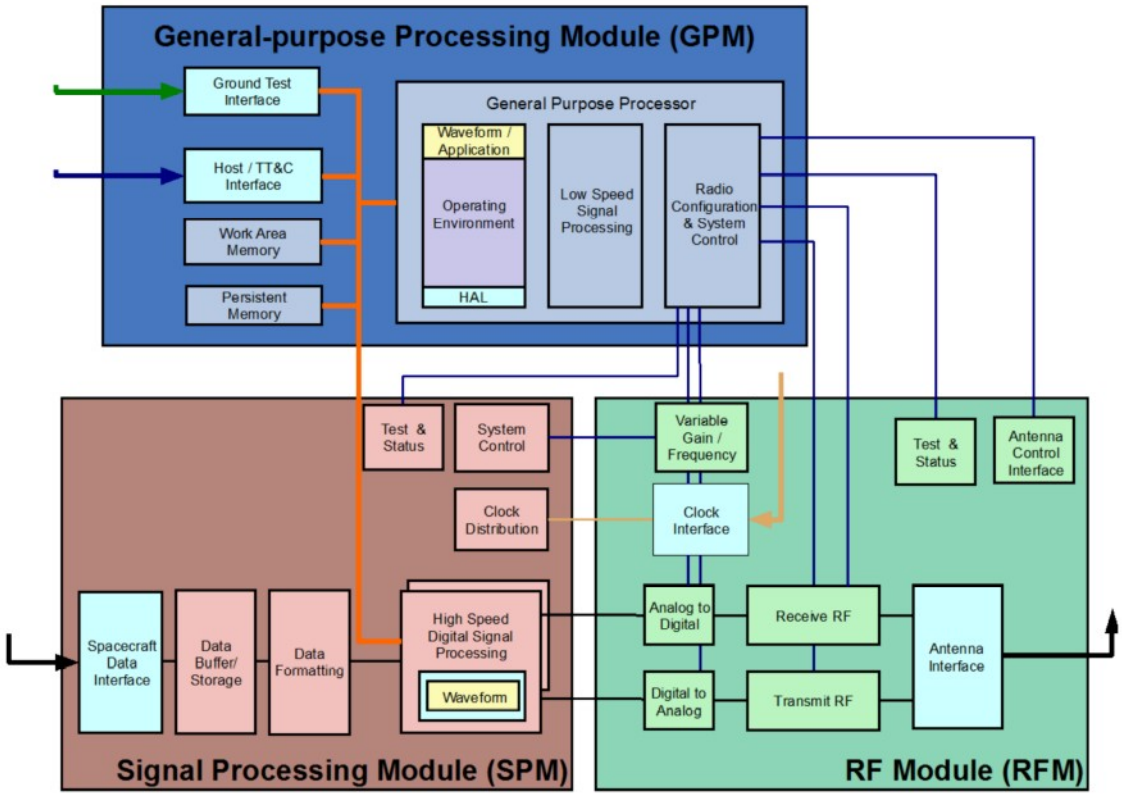

Figure 3: Notional STI Hardware Architecture

Additional modules can be added for increased capability. The hardware architecture does not specify a physical implementation internally on each module, nor does it mandate the standards or ratings of the hardware used to construct the radios. Thus, a radio supplier can encapsulate company proprietary circuit or software designs, provided the modules meet the specific architecture rules and expose the interfaces defined for each module. There is flexibility to physically combine or split these modules as necessary during the radio design process to meet the specific mission requirements or to optimize the design. For example, all RF and signal-processing components or functions may be integrated onto a single printed circuit board, easing footprint, interface, and integration issues, or an approach with multiple boards and enclosures could be used. Similarly, an FPGA could potentially contain both the Signal Processing Module (SPM) functions and the General Purpose Processor (GPP), or the SPM functions could be split between an FPGA and the GPM.

Each project or organization may choose to further standardize certain interfaces and physical packaging. This approach provides organizations with the flexibility to adopt different implementation standards for various project classes. Thus, if a series of radios are required with common operating requirements, physical construction details, such as bus chassis or card slice, can be part of the acquisition strategy. In turn, this modularity may improve the overall cost-effectiveness of a radio system over its service lifetime.

Another example of the flexibility is where a large organization or space mission may choose to standardize the details of the RF-to-signal-processing interface. This might be done to facilitate the use of different RF modules, but the same signal processing module, for radios used for several similar missions. Figure 3 depicts radio facilities, or elements, expected for each module in a notional sense. It should be noted that not all the elements shown in each module are necessarily required for implementation. This architecture specifies the functionality of each module, but it does not necessarily specify how they are implemented. Mission requirements will dictate the implementation approach to each module, and the modules required in each radio.

8.1.1 Components

The approach taken in the STI is to describe the radio hardware architecture in a modular fashion. The generic hardware architecture diagram identifies three main functional components or modules of the STI radio. Although not shown in figure 3, additional modules (e.g., optical, networking, and security) can be added for increased capability and will be included in the specification as it matures.

The hardware architecture currently consists of the following modules:

- **General-Purpose Processing Module** (GPM), which consists of:
  - A suitable general purpose processor (GPP),
  - Appropriate memory (both volatile and nonvolatile),
  - System bus,
  - The spacecraft or host telemetry, tracking, and command (TT&C) interface,
  - Ground test interface,
  - Any required components to support the radio configuration.
- **Signal-Processing Module** (SPM), which consists of:
  - The signal processing used to handle the transformation of received digitally formatted signals into data packets, and/or
  - The conversion of data packets into digitally formatted signals to be transmitted.
  - The spacecraft data interface, which represents any required Application-Specific Integrated Circuits (ASICs), Digital Signal Processors (DSPs), FPGAs, memory, and connection fabric or bus.
- **Radio Frequency Module** (RFM), which consists of:
  - The interfaces that control the final stage of transmission or the first stage of reception of the wireless signals, including antennas.
  - Any required RF functionality to provide the SPM with the filtered, amplified, and correctly formatted signal if acting as a receiver, and/or
  - Any required RF functionality to format, filter, and amplify the signal from the SPM if acting as a transmitter.
  - Its associated components include filters, radio frequency (RF) switches, diplexer, low noise amplifiers (LNAs), power amplifiers, analog to digital converters (ADCs), and digital to analog converters (DACs).
- **Security Module** (SEC). Though not directly identified in the generic hardware diagram, an SEC is also being proposed to allow STI radios to support future security requirements. The details of this module will be defined in later revisions of the architecture.
- **Network Module** (NM): The architecture supports Consultative Committee for Space Data Systems (CCSDS) and Internet Protocol (IPs) networking functions. However, the Network Module (NM) may be realized as a combination of both the GPM and SPM.
- **Optical Module** (OM): This module supports the integration of optical equipment when used. The detail of this module will be defined in later revisions of the architecture. (It has many similarities to RFM, but for optical carriers)

8.1.2 Functions

Test and status, fault monitoring and recovery, and radio and TT&C data-handling functions are to be implemented on all modules to some level. The details of the implementation are mission specific. The related control and interface requirements for the shared module functions are stated in the corresponding module section.

Test and Status

Each module (or combination of modules) should provide a means to query the current health of the module and run diagnostics.

Fault Monitoring and Recovery

Each module (or combination of modules) should incorporate detection of operational errors, upsets, and major component failures. These may be caused by the radiation environment, for example, including single-event upsets (SEUs), temperature fluctuations, or power supply anomalies. In addition to detection, mitigation and fail-safe techniques should be employed. Each module should have a default power-up mode to provide the minimal functionality required by the mission. This fail-safe mode should have minimal software and/or configurable hardware design dependency.

Radio Data Path

SDRs can be implemented with or without the GPM in the data path. The STI architecture supports the separation of the RFM and SPM data paths from the GPM. Giving the GPM access to the data path as an optional capability rather than a required capability allows for a more efficient implementation for medium and small mission classes and improves the overall performance for near-term implementations. If space-qualified GPM components mature with the performance capabilities required for signal processing, the GPM can exist within the data path and take on more signal-processing functionality, increasing flexibility.

Radio Startup Process

The startup of the STI infrastructure is expected to be initiated by the STI platform boot process, so that it can receive and send external commands and instantiate applications. The startup process might include built-in tests for self-diagnostics to verify nominal system functionality. In order to control an STI platform at power-up and to recover from error conditions, an STI platform is to have a known power-up condition that sets the state of all modules. To support upgrades to the OE, an STI platform requires the ability to alter the state (boot parameters) and/or select a boot image. The exact mechanisms and procedures used will be platform and mission specific but need to be sufficient to support upgrades to OE components, such as the OS, BSP, and STI infrastructure.

8.1.3 External Interfaces

There may be several external interfaces in this architecture:

Host TT&C

The host TT&C interface represents the typically low-latency, low-rate interface for the spacecraft (or other host) to communicate with the radio. The host telemetry typically carries all information sourced within the radio. This type of information traditionally is called the telemetry data and includes health, status, and performance parameters of the radio as well as the link in use. In addition, this telemetry often includes radiometric tracking and navigation data. The command portion of this interface contains the information that has the radio itself as the destination of the information. Configuration parameters, configuration data files, new software data files, and operational commands are the typical types of information found on this interface.

Ground Test

The Ground Test Interface provides a “development-level” view of the radio and is exclusively used for ground-based integration and testing functions. It typically provides low-level access to internal parameters not typically

available to the Spacecraft TT&C Interface. It can also provide access when the GPM is not functioning (i.e., during boot).

Data

The Data Interface is the primary interface for data that are sourced from the other end of the link and for data that are sunk to the other end of the link. This interface is separate from the TT&C interface because it typically has a different set of transfer parameters (protocol, speeds, volumes, etc.) than the TT&C information. A common interface point in the spacecraft for this type of interface is the spacecraft solid-state recorder rather than the spacecraft command and data-handling (C&DH) subsystem. This interface is also characterized by medium to high latency and high data rates.

Clock

The Clock Interface is used to input to the radio the frequency reference sufficient for supporting navigation and tracking. This type of input frequency reference is essential to the operation of the radio and provides references to the SPM and RFM. There does not have to be an external clock interface if the SPM or RFM contains an oscillator that performs this function

Antenna

The Antenna Interface is used to connect the electromagnetic signal (input or output) to the radiating element or elements of the spacecraft. It also includes the necessary capability for switching among the elements if required by the mission. Steering the elements, if a function of the overall telecommunications system, is possible through this interface, but it is not typically employed because of overall operational constraints.

Power

The Power Interface, which is not included on the diagram, is described as part of this specification at the highest levels. The Power Interface defines the types and conditions of the input energy to power the radio.

Mission defined

The Mission-defined Interface, which is not included in the diagram, could monitor conditions that the radio encounters such as external temperature, solar radiation, magnetic field strength, attitude, etc. The mission would assign what to do with these values. A thermal interface that monitors temperature could be used to activate a heating element or adjust dynamic factors dependent on temperature in a known way.

8.1.4 Networking Interface

A networking interface does not necessarily map directly to the SPM, GPM, or RFM. The networking interface might handle only spacecraft TT&C data or both spacecraft TT&C data and radio data. This architecture allows for those capabilities.

8.1.5 Internal Interfaces

To support the overall goals of the architecture, the internal interfaces (GPM system bus, GPM RFM control, SPM-to-GPM test, frequency reference, and data path) should be well documented and available without restriction. The GPM system bus (orange lines in figure 3) provides the primary interconnect between elements of the GPM.

The GPM system bus may provide an interface between the microprocessor, the memory elements, and the external interfaces (TT&C and Test) of the GPM. The GPM system bus is the primary interface between the GPM and the SPM, as shown in the interconnection with the major SPM processing elements. Finally, the GPM system bus provides the interface by which the re-programmable and re-configurable elements of the SDR are modified. It supports both the read and write access to the SPM elements, as well as the reloading of hardware configuration files to the FPGAs.

The interface between the GPM and the RFM is primarily a control/status interface. Various RFM elements are controlled by the set of GPM RFM control lines (blue lines in figure 3). Coming from the System Control block in the GPM, this control bus can be either fixed by the System Control function or programmed by the GPM software and validated and routed by the System Control function. It is important to have a hardware-based confirmation and limit-check on the software controlling any RFM elements. The System Control module of the GPM provides this functionality, thus keeping the GPM RFM Control bus within operational limits.

The Ground Test Interface (green line in figure 3) provides specific control and status signals from different modules or functions to the Ground Test Interface block. This interface is used during development and testing to validate the operation of the various radio functions. This interface is very specific to the implementation and realization of the different modules.

The Frequency Reference Interface provides an important interface between the RFM and the SPM functions. It ties the two modules together in a way that allows for the SDR to implement tracking and navigation functions. The characteristics of this interface are defined by the various amounts of tracking accuracy required by the mission for the SPM to accomplish. This interface can be as simple as a single, common frequency reference that is conditioned from an outside source and distributed in the least degrading fashion possible to the SPM. Finally, the data paths are the various streams of bits, symbols, and RF waves connecting the major blocks of the primary data path. For any particular implementation, the data path or bit streams are defined by the particular application implemented in the functional blocks.

The interface between the RFM and SPM should be well defined and have characteristics suitable for that level of conversion between the analog and digital domains. The hardware architecture can be further specified in a manner that is important for implementers to consider and follow, if the implementation dictates the necessity of particular components. Details of the GPM, SPM, and RFM are provided in subsequent sections.

8.2 Module Specification

8.2.1 General Purpose Processing Module

Figure 4, GPM Architecture Details, provides a closeup of the GPM. The GPM consists of one or more general purpose or digital signal-processing elements and support hardware components, embedded OS, software applications and interfaces to support the configuration, control, and status of the radio. The number of processing elements and the extent of support hardware will vary depending on the mission-class processing and data-handling requirements from a single system on a chip implementation for smaller mission classes to multiple logical replaceable units (LRUs) for the largest mission classes. In addition, fault tolerance requirements can also increase

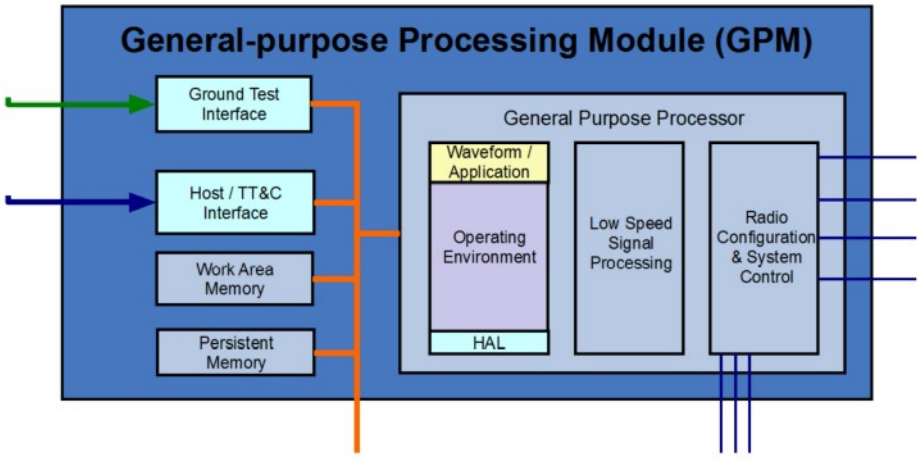

Figure 4: GPM Architecture Details

the number of hardware processing elements, support hardware components, and interface points required to meet the range of mission classes. The majority of processing functions of the GPM will be under software control and supported by an OS. Mission-specific data handling speeds may require the use of separate specialized support hardware (FPGA or ASIC chips) to alleviate the burden on the processing elements. Such specialized support hardware could include encryption, packet routing, and network processing functions.

GPM Components

The GPM contains, as necessary, a GPP and various memory elements as shown in figure 4. Depending on the particular project requirements, not all memory elements are required. The GPP will typically be implemented as a microprocessor, but it could take many forms, depending on the type of deployment. Because the GPM is the primary control component of the radio, it is a required module for an STI radio. A description of each element follows.

The GPP functions include the OE, the Hardware Abstraction Layer (HAL), and potentially application functions. The OE contains the STI infrastructure, which provides the interfaces defined by the STI APIs specification. The OE also contains the operating system and any related libraries.

The *HAL* is the library of software functions in the STI OE that provides a platform-vendor-specific view of the specialized hardware by abstracting the underlying physical hardware interfaces. The HAL allows specialized hardware to be integrated with the GPM so that the STI OE can access functions implemented on the specialized hardware of the STI platform.

The *Persistent Memory Storage* element holds both the permanent (e.g. read-only memory) and reprogrammable storage for the GPP element. This is likely to be implemented using a technology such as electrically erasable, programmable read-only memory (EEPROM) or flash memory, depending on system requirements. The Persistent Memory also provides the storage for the SPM FPGA elements (i.e. configurable hardware design). The GPM may be responsible for programming and scrubbing the SPM FPGAs and, if so, would have access to the appropriate "code" for the FPGAs.

The *Work Area Memory* element is provided as operational, scratch memory for the GPP element. This memory element is implemented in concert with the GPP element and may contain both data and code, as appropriate for the execution of the radio application running in the GPM.

Finally, the GPM contains a *System Control* element to control and moderate the GPM system bus. This element provides the necessary control for the System Bus, including the various memory and SPM elements interfaced by the System Bus. In addition, the System Control element provides a validated interface to the RFM hardware via the GPM RFM Control Interface. As the software running on the GPP element commands the RFM elements into certain states, those commands are interpreted by the System Control element and validated in a manner that will prevent damaging configurations of the RFM; for example, tying the transmit amplifier directly to the receive amplifier, bypassing the diplexer element. This level of validation in the GPM-to-RFM interfaces is intended to prevent physical damage to the radio arising from a software bug. The System Control element may also be implemented by an FPGA, but if so, it should have appropriate safeguards to ensure that the FPGA cannot be modified inadvertently during flight (e.g. such as using a "permanently programmed" device or by otherwise disabling the reprogramming capabilities).

GPM Functions

The GPM will provide the overall configuration and control of the STI architecture and may include any or all of the following functions:

- Management and Control
  - Module discovery
  - Configuration control
  - Command, control, and status

    - Fault recovery
    - Encryption
- STI infrastructure, radio configuration and control.
    - Radio control
    - System management
    - Application upload management
    - Device control
    - Message center
- External network interface processing
- Internal data routing
- Waveform data link layer
    - Media Access Control (MAC) and Logical Link Control (LLC) layer
    - Physical layer processing
- Onboard data switching

GPM Interfaces

- TT&C
- Ground Test
- General-purpose input output (GPIO), supporting but not limited to:
    - Interrupt source/sink.
    - Application data transfer.
- Control/configuration interface, supporting but not limited to:
    - RFM & SPM
    - Antenna
    - Power amplifier
- System Bus interface

Summary of GPM Requirements

► An STI platform shall have a GPM that contains and executes the STI OE and the control portions of the STI applications and services software.

► A module's diagnostic information shall be available via the STI APIs.

8.2.2 Signal-Processing Module

Figure 5, SPM Architecture Details, illustrates the SPM module. An SPM is optional for an STI platform. The SPM may implement the signal processing used to transform received digital signals into data packets and/or the

conversion of data packets into digital signals to transmit. The complexity of this module is based on the applications and data rates selected for a mission. The SPM modules contain components and capabilities to manipulate and manage digital signals that need higher processing capabilities than that supplied by the GPM. The configurable hardware design architecture describes a common interface for the application on the SPM, as described in section 9.1.

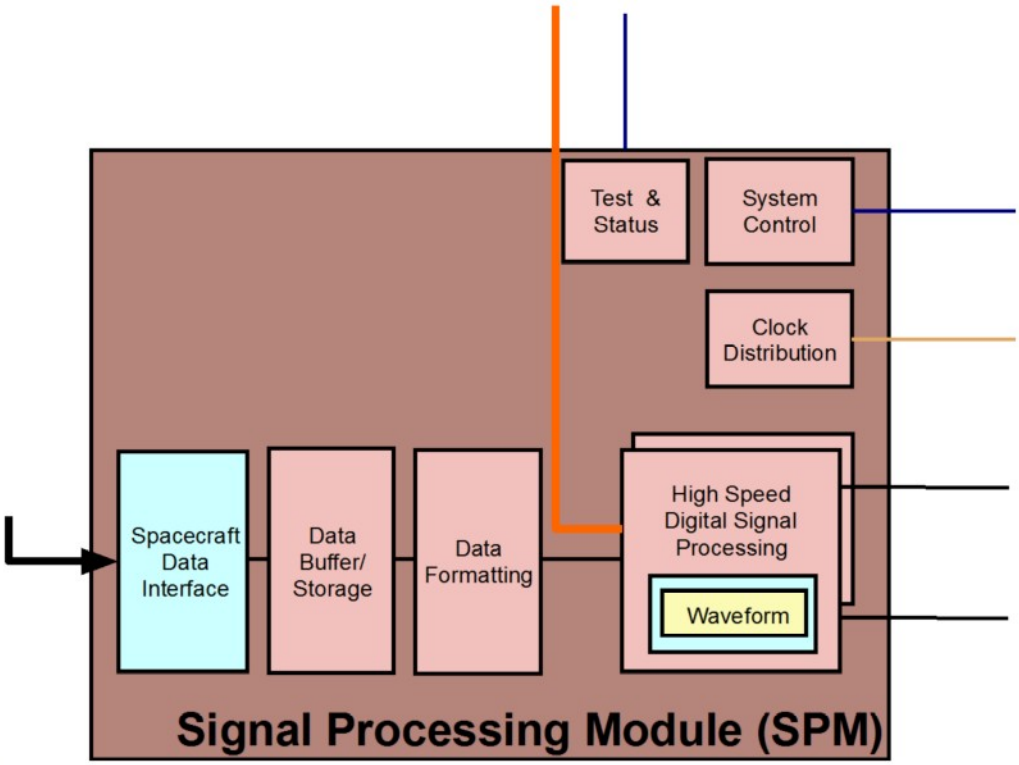

Figure 5: SPM Architecture Details

SPM Components

The SPM will initially be implemented primarily with FPGAs, DSPs, reconfigurable processors, ASICs, and other integrated circuits. However, technologies will change over time, so the specific implementation is left to the STI platform provider. It is also anticipated that STI platforms may use dedicated physical hardware slices (e.g., separate circuit boards) to implement specialized applications and technologies. For example, a dedicated global positioning system (GPS) receiver slice can complement the existence of reconfigurable SPM slices in the same radio. The dedicated slice offloads demand on the less specific SPM. If an STI platform contains an SPM slice, the slice should meet the module interface specifications for control and configuration and have an interface with the GPM via the GPM system bus and the SPM-to-GPM test interface. These two interfaces work in concert to provide a control and reprogramming path to the SPM from the GPM and the application running on the GPM.

SPM Functions

The SPM implements the digital signal processing functions that convert symbols to bits and vice versa. These functions are typically implemented on FPGAs, DSPs, or ASICs. It is recommended that reconfigurable and reprogrammable devices be used because this allows for new applications to be implemented on the SDR in the future without a hardware modification. However, mission-specific requirements may dictate that the application be implemented on a non-reprogrammable hardware device.

In addition to the digital signal processing functions, a data formatting function is typically provided to convert blocks of data stored in the data storage element into bit streams appropriate for encoding into symbols and vice

versa. The STI architecture does not require that these are discrete entities; in some cases, it may be possible to implement the data formatting function in the same device as the digital signal processing function.

A data storage element may be used to provide a buffer between the data interface and the bit stream coders/decoders. This data storage function can be implemented in either volatile or nonvolatile memory, depending on the operational requirements. An SPM may implement any or all of the following digital communication functions depending upon the mission waveforms:

- Digital up conversion—interpolation, filtering, and "local oscillator" multiplication of baseband samples to obtain an IF or RF output sample stream appropriate for digital-to-analog conversion. This is typically the last transmit function implemented in the SPM, and the output samples are sent to the RFM.
- Digital down conversion—multiplication with "local oscillator," downsampling, and filtering IF or RF samples to obtain a baseband output sample stream. This is typically the first receive function implemented in the SPM, with input samples coming from the analog-to-digital conversion in the RFM.
- Digital filtering—averaging, low-pass, high-pass, band-pass, polyphase, and other filters used for pulse shaping, matched filter, etc. This may overlap with some of the functionality in the up and down conversion.
- Carrier recovery and tracking—retrieval of the waveform carrier within the receive sample stream. Typical SPM functions for carrier recovery include shifting the recovered carrier frequency to compensate for local oscillator variations and Doppler shifts in the link.
- Synchronization (data, symbol, etc.)—alignment of received samples with symbol and data boundaries. There may be some integration with the digital down conversion and carrier recovery and tracking functions.
- Forward error correction coding—encoding transmit data so that receive data errors may be corrected to some level, enhancing the waveform performance.
- Digital automatic gain control (AGC)—scaling of the receive samples to optimize downstream operations.
- Symbol mapping (modulation)—translating transmit data bits to modulation symbol samples.
- Data detection (demodulation)—translating receive symbol samples to data bits.
- Spreading and despreading—a form of encoding data to obtain certain energy dispersion in the frequency domain.
- Scrambling and descrambling—a form of encoding data to ensure a certain level of randomness in the digital data stream, usually for synchronization of the receiver.
- Encryption and decryption—a form of encoding data for security purposes.
- Data Input/Output (I/O) (high-speed direct from or to source or sink)—interface for transmit and/or receive data to come in or out of the module. This may involve buffering and some protocol handling.

SPM Interfaces

The SPM's functions and external interfaces are shown in figure 5. Interfaces shown include those common to all module types as well as those specific for the SPM. These SPM-specific interfaces may not all be required for some radios. Note that the implementation of these interfaces may combine two or more on one physical transport. For example, the Data Interface and Control and Configuration Interfaces may both use the same physical Serial Rapid I/O connection.

- Data I/O to or from RFM—This is the digital sample stream going to the RFM's DACs for transmission, and the digital samples from the RFM's ADCs. However, if the DACs and ADCs are preferred to be a part of the SPM, then this interface is replaced with analog baseband or IF signals.

- Waveform control and feedback to RFM—This interface will be waveform dependent. It may be used, for example, to send feedback to an AGC or control frequency hopping.
- Data interface external to the radio—High-data-rate waveforms may need a direct interface to the SPM if the GPM is not designed to handle the data.
- System bus—Data to or from GPM—This interface exchanges the packetized data for transmission and reception.
- Control and configuration from GPM—Waveform loads and reconfigurable parameters are managed through this interface.
- Test and status to GPM—Tests are initiated through this interface by the GPM, and results are returned. This is a more basic interface (electrically and protocol-wise) than the Control and Configuration interface.
- Radiometric tracking.

The HID is to contain the characteristics of each reconfigurable device. Reconfigurable capacity is usually measured by the number of FPGA gates, slices, logic elements, or bytes. This information can be used by future STI application developers to determine the waveforms that can be implemented on a given platform.

8.2.3 Radio Frequency Module

The RFM handles the conversion to and from the carrier frequency, providing the SPM and/or the GPM with digital baseband or IF signals, and the transmission and reception equipment with RF to support the SPM and GPM functions. Its components typically include DACs, ADCs, RF switches, up converters, down converters, diplexers, filters, LNAs, power amplifiers, etc. Current and near-term RF technologies cannot be expected to allow multiband operation using a single channel RFM, and thus multiband radios will need to use multiple RFM slices. The RFM provides a band of frequency tunability on each slice. This tunability can be software controlled through the provided interfaces.

The RF module handles the interfaces that control the final stage of transmission or first stage of reception of the wireless signals, including antennas, optical telescopes, steerable antennas, external power amplifiers, diplexers, triplexers, RF switches, etc. These external radio equipment components would otherwise be integrated with the RFM except for the physical size and location constraints for transmission and reception. The interfaces are primarily the associated control interfaces for these components. The RFM HID encompasses the control and interface mechanism to the external components. The focus of the RF HID is to provide a standardized interface to the control of each of these devices, to synchronize the operation of the radio with any of these devices.

The other primary capability of the RFM is the conditioning and distribution of the frequency reference as defined by the Frequency Reference Interface. This provides a common reference for the RFM and SPM modules to enable the tracking and navigation functionality typically provided by SDRs. Figure 6, RFM Architecture Details, illustrates the RFM module.

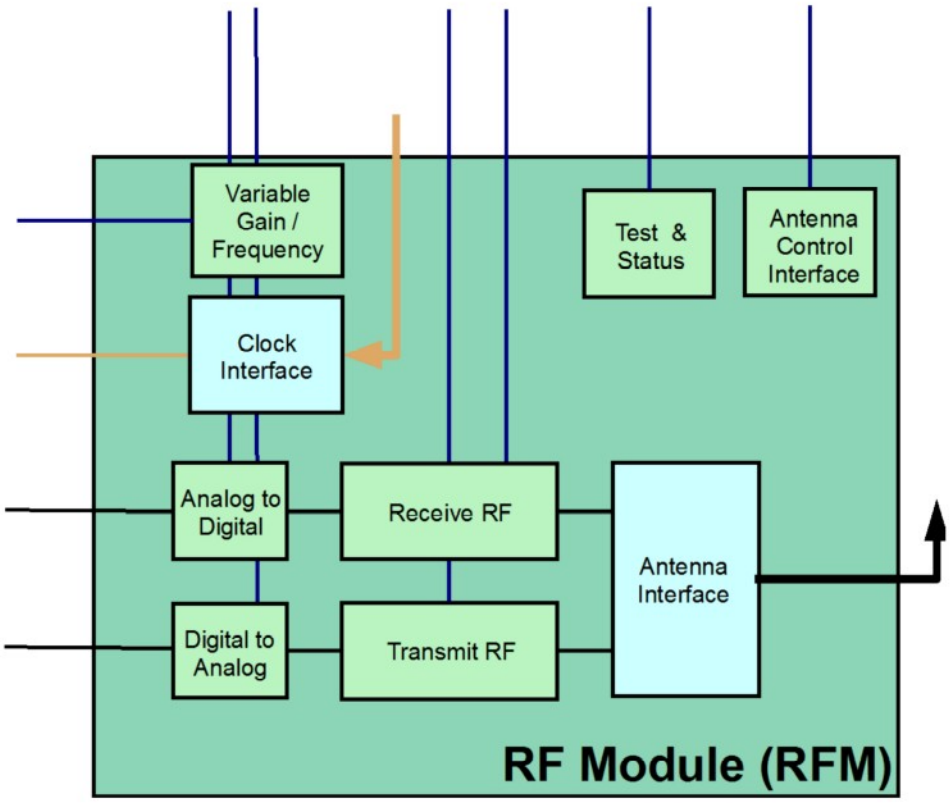

Figure 6: RFM Architecture Details

RFM Functions

The RFM transforms the antenna signal to or from a signal usable to the radio. The RFM functions are likely to include the following:

- Frequency conversion and gain control
- Analog filtering
- Analog-to-digital and digital-to-analog conversion.
- Radiometric tracking

RFM Components

The RFM can be implemented with a variety of integrated circuits. The control of these circuits can be implemented with a variety of different component technologies, including ASICs, discrete electronics, programmable logic devices including FPGAs and DSPs, or even microprocessors. The choice of technologies is left up to the developer of the particular implementation. It is expected that the control of the devices will become more sophisticated over time and that the level of control will increase, resulting in more complex control circuitry and logic devices being used.

RFM Interface

The RFM implements the following interfaces:

- External RF interface(s) to the radio.

- Read and write access to interface registers to monitor and perform control, status, and failure and fault-recovery functions (e.g., via RS-422 or SpaceWire).
    - Control: power level tunability, frequency tunability, antenna parameter tunability, etc.
    - Status: report status of components and system operation.
    - Failure and fault-recovery functions: detect component or system failure and determine appropriate action.
    - Diagnostic test functions
- I/O for exchanging digitized waveform signal data.

Summary of RFM Requirements

► The STI platform provider shall describe, in the HID document, the behavior and performance of the RF modular component(s).

The behavior and performance of the RF modular components should be sufficiently described such that future waveform developments may take advantage of the RF capability and/or account for its performance. Information in the HID may include such items as center frequency, IF and RF frequency(s), bandwidth(s), IF and RF input/output level(s), dynamic range, sensitivity, overall noise figure, AGC, frequency accuracy and stability, and frequency-tuning resolution.

8.2.4 Security Module

The goal of the security module is to address the security services required from an SDR. Currently this is a notional concept, as there are no specific requirements for this module, but a future revision of the STI standard may add requirements or specific details. This approach supports the evolutionary nature of the STI architecture; it is expected that this module will become more well-defined as feedback is received and common interfaces are identified.

If implemented, the architecture should support selectable data-protection services for entities requiring them, providing for both confidentiality and authentication. Missions may select security options provided by the infrastructure or may develop their own.

The authentication of commands sent to SDRs is supported, including changing the configuration or uploading new programs for either the infrastructure or new applications. The security section of the architecture will include support for key management, encryption standards, and mitigating threats other than the information and communication security threats currently identified.

8.2.5 Networking Module

The STI architecture has been structured such that networks can be implemented in an SDR; that is, an SDR can be a node in a network. The SDR may be connected to another node using the appropriate logical and physical interfaces that may be wired or wireless. The STI architecture can accommodate network protocols as services that can be made available to applications and devices. STI supports the ability to upload new software and dynamic hardware images. Therefore, advancements and replacement of existing protocols can be accomplished without affecting a spacecraft's mission resources.

8.2.6 Optical Module

The STI architecture also supports the use of optical communications in SDRs. The optical module, if present, would logically replace the Radio Frequency Module (RFM) that is typically used for RF communication.

STI interfacing to optical communication equipment follows the same techniques shown in integration with high-data-rate hardware. The OM would be controlled through the STI HAL interface that allows configuration and control of the digital components in the module, which abstracts the optical functionality.

8.3 Hardware Interface Description

The STI platform provider is to provide an HID document, which describes the physical interfaces, functionality, and performance of the entire platform and each platform module. The HID specifies the electrical interfaces, connector requirements, and all physical requirements for the delivered radio. The HID abstracts and describes the functionality and performance of each module. In this manner, STI application developers can know the features and limitations of the platform for their applications. The information in the HID provides the knowledge for OMG and others to integrate and test the hardware interfaces. The information in the HID may allow future module replacement or additions without the design of a completely new platform. For example, a Security Module could be added that was not originally planned, or a follow-on mission could use a different frequency band and only an RFM change would be needed. Include all waveform interfaces and any other interfaces that could be important to a waveform developer or a hardware integrator.

In addition to the GPM, SPM, and RFM HID requirements stated within each module section, the following interface descriptions and requirements are also specified for an STI platform.

Summary of HID Requirements

► The STI platform provider shall describe, in the HID document, the state of all hardware devices in the system after completion of power-up process

► The STI platform provider shall describe, in the HID document, the behavior and capability of each major module or component available for use by a waveform, service, or other application (e.g., FPGA, GPP, DSP, or memory), noting any operational limitations.

► The STI platform provider shall describe, in the HID document, the various capabilities, capacities, and any limitations of each reconfigurable component.

The description of the behavior and capability of modules or components available to STI application developers or reconfigurable components may include device type, processing capability, clock speeds, memory size(s), types(s), and speed(s), noting any constraints, as well as any limitation on the number of configurable hardware design reloads, as applicable, partial reload ability, built-in functionality, and any corresponding restriction on the number of gates.

► The STI platform provider shall describe, in the HID document, the interfaces that are provided to and from each modular component of the STI platform.

The specific modular components or hardware slices of an STI platform will vary among different implementations. The STI platform provider or STI integrator is expected to describe each modular component and their respective physical and logical interfaces as described in this section. Table 1, Module Interface Characterization, provides typical interface characteristics that should be included when identifying external interfaces or internal interfaces between modules for STI.

Parameter	**Description and Comments**
Name	Interface name (data, control, operating power, RF, security, etc.).
Interface type	Point to point, point-multipoint, multipoint, serial, bus, other.
Implementation level	Component, module, board, chassis, remote node.
Reference documents and standards	Applicable documents for interface standards or description of custom interfaces.
Notes and constraints	Variances from standards, physical and logical functional limitations.

Transfer speed	Clock speed, throughput speed.
Signal definition	Description of functionality and intended use.
Physical Implementation	
Technology	For example, GPP, DSP, FPGA, ASIC, and description.
Connectors	Model number, pin out (including unused pins).
Data plane	Width, speed, timing, data encoding, protocols.
Control plane	Control signals, control messages or commanding, interrupts, message protocol.
Functional Implementation	
Models	Data plane model, control plane model, test bench model.
Power	Voltages, currents, noise, conducted immunity, susceptibility.
API	Custom or standard, particular to OS environment.
Software	Device drivers, development environment, and tool chain.
Logical Implementation	
Addressing	Method, schemes.
Channels	Open, close.
Connection Type	Forward, terminate, test.

Table 1: Module Interface Characterization

8.3.1 Control and Data Interface

The control and data communications buses and links between modules within the radio are to be described by the STI platform provider to the level of detail necessary to facilitate integration of another vendor's module. If modules communicate using the IEEE 1394, A High Performance Serial Bus, interface, for example, this will be specified in the HID with appropriate connector and pinout information. Any nonstandard protocols used should also be specified. In some cases, this may be handled by the software HAL. Module interfaces will be completely described, including any unused pins.

Summary of Requirements for Control and Data Interface

► The STI platform provider shall describe, in the HID document, the control, telemetry, and data mechanisms of each modular component (i.e., how to program or control each modular component of the platform, and how to use or access each device or software component, noting any proprietary and nonstandard aspects).

Besides the interface descriptions already provided for each modular component, developers should provide specific information necessary for future STI application developers to know how to interact with the command and control aspects of the platform. The description of the control, telemetry, and data mechanism of each modular component should facilitate the porting of the application software to the platform.

8.3.2 Operating Power Interface

The operating power interface description for the radio has two parts:

1. the platform as a supplier to the various modules; and
2. the power consumption of the different modules, if multiple modules are provided.

Table 2, Example Operating Power Interface, shows an example listing of a platform operating power interface. There are four distinct sets of power requirements for the platform shown. For each module delivered with the radio, as well as those built by other vendors, the HID is to specify the needed voltages, currents, and connections. Voltages are to be specified with a maximum and minimum tolerance, and associated currents are to be specified with nominal and maximum values. Connectors for operating power are to be specified, including pinouts. If power is routed through a multipurpose connector such as a backplane connector, then the pins actually used are to be documented. Power is a limited commodity for most missions, and understanding the STI platform power needs is critical.

Parameter	**Values**			
Voltage Rail	-15V	+2.5V	+5V	+15V
Maximum current/chassis (platform)	2A	1.7A	3A	2A
Maximum current/slot (module)	1A	1A	1A	1A
Backplane supply pins	17,19	59,61	44,46,48	21,23
Backplane return pins	18,20	60,62	43,45,47	22,24
Voltage Ripple	100 mVpp	1 mVpp	5 mVpp	100 mVpp
Notes	Slot 1 & 2 only			Slot 1 & 2 only

Table 2: Example Operating Power Interface

Summary of Requirements for Operating Power Interface

► The STI platform provider shall describe, in the HID document, the behavior and performance of any power supply or power converter modular component(s).

8.3.3 Thermal Interface and Power Consumption

The power consumption and resulting heat generation of a reprogrammable FPGA will vary according to the amount of logic used, the switching rate of the waveform logic, and the clock frequency(s). The power consumption may not be constant for each possible waveform that can be loaded on the platform. The STI platform provider should document the maximum allowable power available and thermal dissipation of the FPGA(s) on the basis of the maximum allowable thermal constraints of FPGA(s) of the platform. For human spaceflight environments, touch temperature requirements may limit dissipation further; therefore, these reductions are to be factored into the given dissipation limits.

Summary of Requirements for Thermal and Power System

► The STI platform provider shall describe, in the HID document, the thermal and power limits of the hardware at the smallest modular level to which power is controlled.

9 Application Architecture

As shown in figure 7, Waveform Component Distribution, an example STI platform consists of one or more GPMs with GPPs, and optionally one or more SPMs containing DSPs, FPGAs, and ASICs. Application components loaded and executed on these modules provide the signal-processing algorithms necessary to generate or receive RF signals. To aid portability and reusability, the applications are to use the appropriate infrastructure APIs to access platform services. Using "direct to hardware" access instead would increase the effort to port the application to a platform with different hardware. The STI infrastructure provides the APIs and services necessary to load, verify, execute, change parameters, terminate, or unload an application. The STI infrastructure implements device components that utilize the HAL or vendor-specific API to abstract communications with the specialized hardware, whereas the HID identifies the hardware interfaces and how modules are physically integrated on a platform.

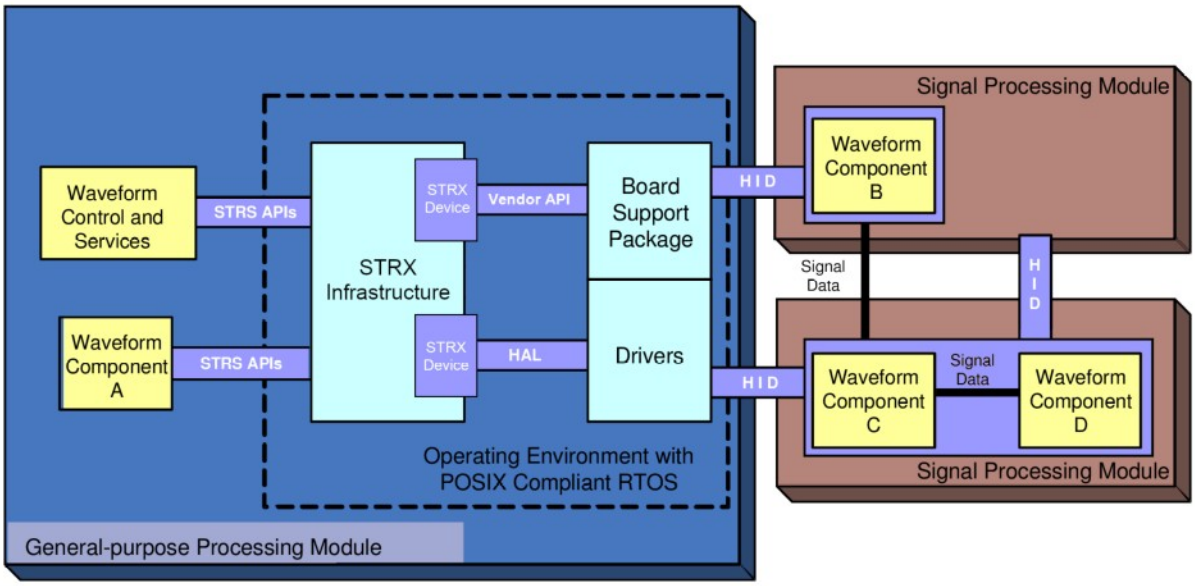

Figure 7: Waveform Component Distribution

The STI infrastructure shall implement device components to serve as a hardware abstraction layer for devices accessed by STI applications. These devices may in turn use the underlying platform HAL APIs, such as a device driver implemented to a standardized software interface. Alternatively, the device may use a custom vendor-specific API to communicate with application components on the platform specialized hardware via the physical interface defined by the STI platform provider.

9.1 Configurable Hardware Design

A configurable hardware design is one where data is used to configure a portion of the hardware without physical modification of the hardware. Configurable hardware designs are realized using a hardware device such as an FPGA or other type of programmable logic device (PLD). This section addresses the use of configurable hardware design from design and development through testing and verification and operations. It addresses aspects of model based design techniques and design for space environment applications.

Proper testing of configurable hardware design is critical to the development of reliable and reusable code. Development tools that enable early development and testing should be used so that problems can be identified and resolved early in the SDR life cycle. Many real-world signal degradations and SEUs can be simulated to identify potential issues with the waveform and waveform functions early in development, even before hardware is available.

Applications implemented in configurable hardware should be modular with clear interfaces to enable individual application component simulations and incremental testing.

The configurable hardware design architecture supports the modeling of STI applications implemented in configurable hardware at the system, subsystem, and functional levels. Model-based design techniques aid in the development of modular application functions. Application development models done in a platform (or target) independent manner aid in application testing, reuse, and portability. A platform-independent model (PIM) design can be used to target different platforms. PIM design flows might include high level models combined with manual code writing. On resource-constrained platforms, optimized code would be written. On nonresource-constrained platforms, PIMs may be used to auto generate code. These design flows can be employed to significantly reduce the porting effort.

Application portability and reusability should be considered in all facets of the design process from concept to implementation to testing. The coding technique of the application is also essential to reduce the application porting effort. Having defined syntax standards for HDLs (e.g., Verilog or VHDL) makes them appear to be easily portable across devices and software synthesizers, but this is an incorrect assumption. There are many things that can make hardware description languages hard to port. For example, the use of device-specific fixed hardware logic on the FPGA will decrease the portability. The use of specialized hardware may be necessary to meet the timing constraints of the application; however, the STI application developer should document any application function that uses the specialized hardware so that the effort to port the application function(s) can be determined. Non-boolean-type logic such as clock generation can also reduce portability. One method to decrease the porting effort would be to create a module that does the clock generation from which the rest of the application functions receive the necessary clock(s).

Development of configurable hardware design for STI radios should include provisions for mitigating space environmental effects such as SEUs. Near-term application of static random access memory (SRAM)-based FPGAs may require triple-mode redundancy (TMR), configuration memory scrubbing, and other mitigation techniques, depending on the intended mission environment and desired reliability. Commercial design tools are becoming available to aid in this process and some newer FPGAs have versions available with embedded TMR.

A key feature of SDRs is that they can be reconfigured after deployment. The ability to load new applications and services will benefit missions in several ways, including using one SDR (instead of several separate radios) to handle different applications for various phases of a mission, some planned and some unplanned. An STI platform should receive STI application software and configurable hardware design updates after deployment.

9.2 Specialized Hardware Interfaces

Standardizing and documenting the interface from the waveform applications on the GPP to the portion of the waveform in the specialized processing hardware, such as FPGAs, is intended to provide commonality among different STI platforms and to aid portability of application functional components implemented in configurable hardware design. Figure 8, Notional High Level Software and Configurable Hardware Design, depicts the high-level interface relationship between GPM, SPM, and RFM modules in an STI radio.

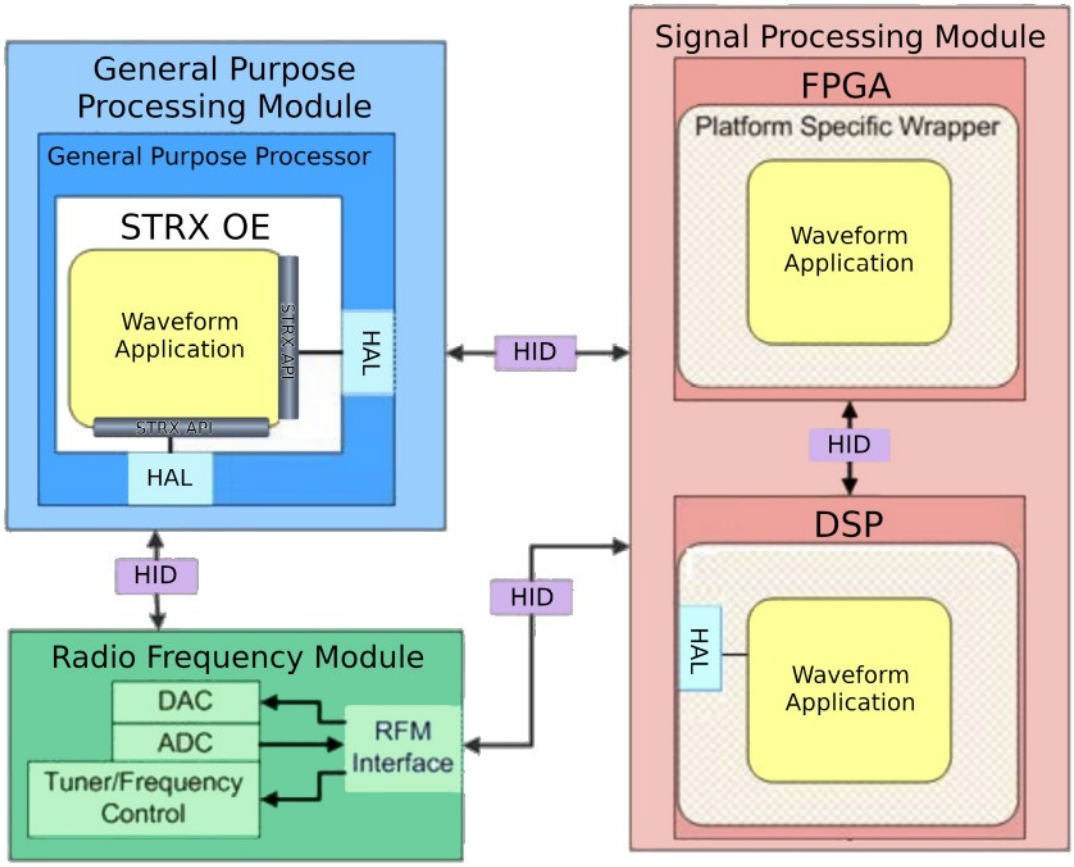

Figure 8: Notional High Level Software and Configurable Hardware Design

The STI architecture provides a common mechanism for the software to instantiate, configure, and execute the software and configurable hardware design applications on various platforms using different hardware devices. Reconfiguration may include changing the parameters of installed applications and uploading new applications after deployment.

The application accepts configuration and control commands from the GPM and uses STI APIs that interface to the device drivers associated with the SPM and RFM modules. The device drivers communicate via the HAL on the GPM that abstracts the physical interface specification described in the HID in transferring command and data information between the modules.

For FPGAs, the interface to the application is through a platform-specific wrapper. The platform-specific wrapper accepts command and data information from the GPM and provides them to the application. The platform-specific wrapper also abstracts details of the platform from the STI application developer, such as pinout information. The platform-specific wrapper should also provide clock generation, signal registering, and synchronization functions, and any other non-waveform-specific functions that the platform requires.

Documentation of the platform-specific wrapper is necessary so that STI application developers can interface applications to the platform. This documentation should include detailed timing constraints, such as signal hold times, minimum pulse widths, and duty cycles. The signal timing constraints refer to the protocol of a particular interface describing events happening on a particular clock cycle. For clock generation, one should document what clock domains are in the design, how each clock is generated, and the resources that are involved. Signal synchronization describes any additional logic needed when clock domains are changed across the interface. The

signal registering methods refer to any configurable hardware design interfaces between modules and if the input and output were registered, latched, or neither.

Summary of Requirements for Specialized Interfaces

► If the STI application has a component resident outside the GPM (e.g., in configurable hardware design), then the component shall be controllable from the STI OE.

► The STI SPM developer shall provide a platform specific wrapper for each user programmable FPGA, which performs the following functions:

1. Provides an interface for command and data from the GPM to the waveform application.
2. Provides the platform-specific pinout for the STI application developer. This may be a complete abstraction of the actual FPGA pinouts with only waveform application signal names provided.

► The STI SPM developer shall provide documentation on the configurable hardware design interfaces of the platform-specific wrapper for each user-programmable FPGA, which describes the following:

1. Signal names and descriptions.
2. Signal polarity, format, and data type.
3. Signal direction.
4. Signal-timing constraints.
5. Clock generation and synchronization methods.
6. Signal-registering methods.
7. Identification of development tool set used.
8. Any included non-interface functionality.

10 Software Architecture

The STI architecture is predicated on the need to provide a consistent and extensible development environment on which to construct SDR applications. The breadth of this goal implies that the specification be based on the following:

1. Core interfaces that allow flexibility in the development of application software; and
2. Hardware and software interface documentation that enable technology infusion.

10.1 Software Layer Model

The software architecture model shows the relationship between the software layers expected in an STI-compliant radio. The model illustrates the different software elements used in the software execution and defines the software interface layers between applications and the OE and the interface between the OE and the hardware platform. Figure 9, Software Execution Model, represents the software architecture execution model. The software model achieves the following objectives:

a) Abstracts the application from the underlying OE software to promote portability and reusability of the application.

b) Within the abstraction layer, minimizes custom routines by using commercial software standard interfaces such as POSIX®.

c) Depicts the STI software components as layers to specify their relationship to each other and their separation from each other which enables developers to implement the layers differently according to their needs while still complying with the architecture.

d) Introduces a lower-level abstraction layer between the OE and the platform hardware. Note that although software abstraction for general processors is typically accomplished with board support packages and device drivers, the abstraction of hardware languages or configurable hardware design is less defined. The model represents the software and configurable hardware design abstraction in this layer.

e) Indicates the relationship between the OE software and the different hardware processing elements (e.g., processor and specialized hardware).

The OE adheres to the interface descriptions provided in figure 9. This specification provides two primary interface definitions, as follows: (1) The STI APIs; and (2) The STI HAL specification, each with a control and data plane specification for interchanging configuration and run-time data. The STI APIs provide the interfaces that allow applications to be instantiated and use platform services. These APIs also enable communication between application components. The HAL specification describes the physical and logical interfaces for inter-module and intra-module integration.

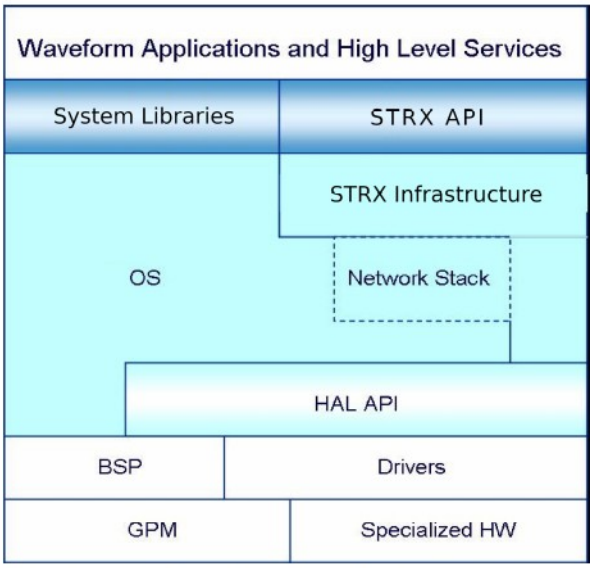

Figure 9: Software Execution Model

The STI software architecture presents a consistent set of APIs to allow waveform applications, services, and communication equipment to interoperate in meeting an application specification. Figure 10, Layered Structure in UML, represents a view of the platform OE that depicts the boundaries between the STI infrastructure provided by the STI platform provider and the components that can be developed by third-party vendors (e.g., waveform applications and services).

A key enabler of application portability and reusability is the removal of application dependencies on the infrastructure that take advantage of explicit knowledge of the infrastructure implementation. When waveforms and services conform to the API specification, they are easier to port to other STI platform implementations.

Figure 10 extends the view of the software architecture from the diagram introduced in figure 9 to include additional detail of the infrastructure, operating system, and hardware platform using Unified Modeling Language (UML) symbols. This approach clarifies the interfaces between components, adding additional detail.

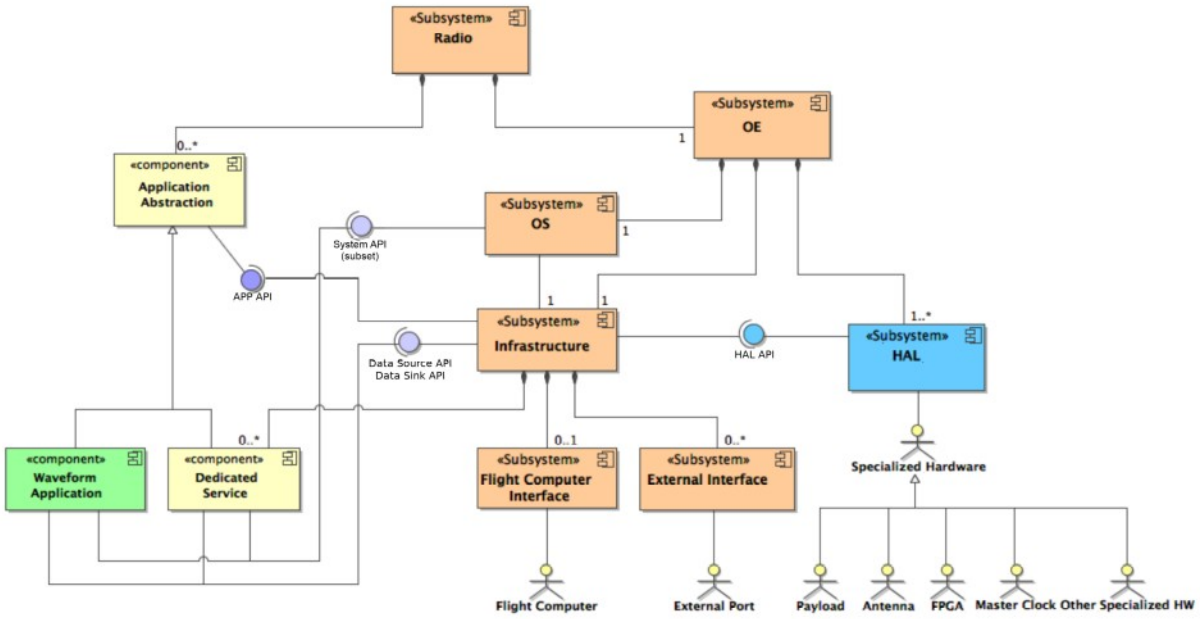

Figure 10: Layered Structure in UML

Figure 11, Operating Environment, describes the elements of the detailed OE depicted in figure 9. In the case that the OS or platform does not support the full set of dependencies, the missing functionality is to be implemented in the STI infrastructure using a compatibility layer. Figure 11 also illustrates the inclusion of a compatibility layer in the infrastructure. For example, when using non-POSIX® OS, the compatibility layer would implement any POSIX® functions required but not implemented by the OS.

In figure 11 the arrows identify interface dependencies and isolations. The waveform applications will not directly call the driver API but use the provided STI APIs, thus providing the "abstraction layer" that helps isolate the application from the platform.

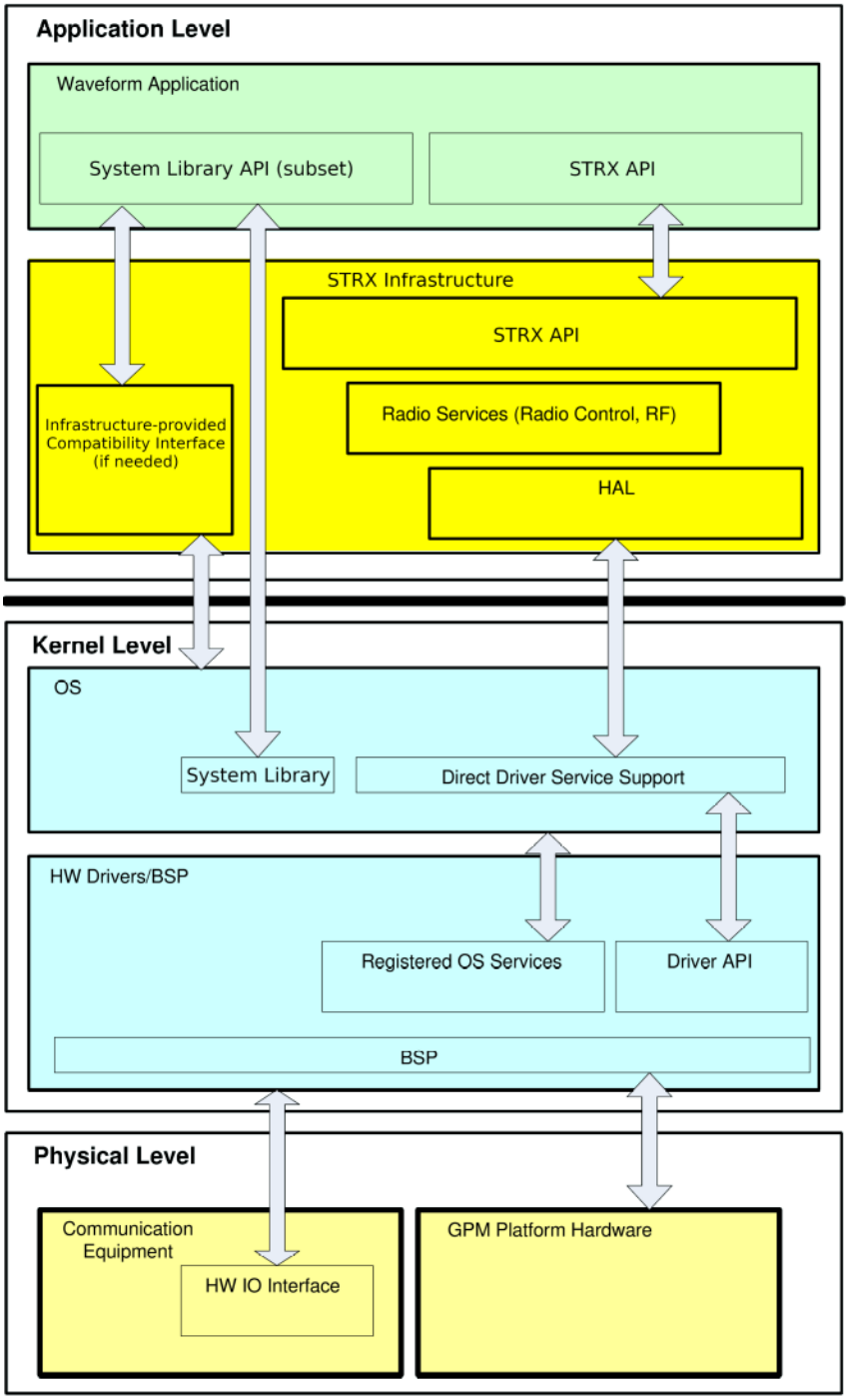

Figure 11: Operating Environment

In table 3, Software Component Descriptions, the different layers of the software model shown in figure 11 are further described.

Layer	Description
Waveform Application and services	Waveform application and services provide the radio GPP functionality using the STI infrastructure.
STI infrastructure	The STI infrastructure implements the behavior and functionality identified by the STI APIs as well as other required radio functionality
STI API	The STI APIs provides consistent interfaces for the STI infrastructure to control applications and services, and for the applications and services to access STI infrastructure services.
APP API	The APP API is the interface implemented by waveforms and services whose functions are used by the STI infrastructure.
Infrastructure-provided compatibility interface	This optional interface (see figure 12, Standards-Compliant vs Standards-Conformant OS) provides compatible services to the waveform application on platforms which do not implement these services natively.
Radio control services	These services are responsible for handling the radio commands and telemetry for the STI. Applications use the STI interface to communicate telemetry and receive commands from flight computer
HAL	The HAL provides the device control interfaces that are responsible for all access to the hardware devices in the STI radio. The HAL API is the interface to the software drivers and BSP that communicates with the hardware.
System Library API	The specific subset of system library functions utilized by the STI waveform application. For POSIX®-based environments, this is the minimum Application Environment Profile required by the waveforms.
OS	This is the operating system that supports the POSIX® API and other OS services. The POSIX® Abstraction Layer will provide applications with a consistent AEP interface that is mapped into the chosen OS functions
System Library	This is the implementation of the system library provided by the operating system or programming language environment.
Direct service support	This layer identifies the ability for the STI infrastructure to have a direct interface to the hardware drivers on the platform.
HW drivers/BSP	The hardware drivers provide the platform independence to the software and infrastructure by abstracting the physical hardware interfaces into a consistent device control API.
Registered OS services	These are services that are integrated with the chosen OS to provide services such as MAC-layer interface to physical Ethernet hardware.
Driver API	OS-supplied APIs are abstracted from applications via the device control API
BSP	The BSP is the software that implements the device drivers and parts of the kernel for a specific piece of hardware. It provides the hardware abstraction of the GPM module for the POSIX®-compliant OS. A BSP contains source files, binary files, or both. A BSP contains an original equipment manufacturer (OEM) adaptation layer (OAL), which includes a boot loader for initializing the hardware and loading the OS image. Essentially, the OAL is all of the software that is hardware specific. The OAL is actually compiled and linked into the embedded OS.
HW I/O interfaces	Device drivers have been created for these physical interfaces

GPM	This is the general-purpose processing module on which the STI infrastructure executes.
Specialized hardware	This is the physical layer of the hardware modules existing on the STI platform

Table 3: Software Component Descriptions

Figure 12 illustrates the difference between a standards-conformant OS and a nonconformant OS. On the left side, the prescribed set of application interfaces is provided entirely by the OS. On the right side, the OS is not directly conformant but is partially compliant. The application profile is shown in two parts: one part shows the compliant APIs that are directly included in the OS, and the other part shows the portion of the profile that is provided though some form of abstraction or compatibility layer. For support of waveforms implemented in C/C++, the STI OE must include a POSIX® PSE51-conformant OS or POSIX® abstraction layer for missing APIs.

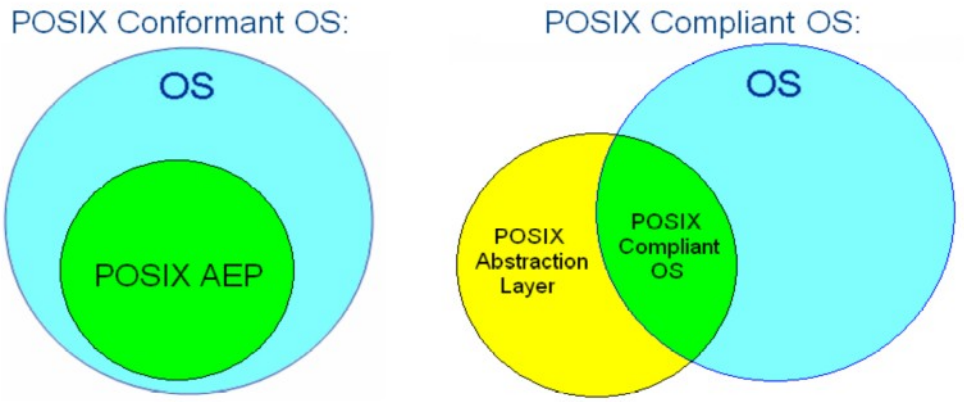

Figure 12: Standards-Compliant vs Standards-Conformant OS

10.2 Infrastructure

The STI infrastructure is part of the OE and provides the functionality for the interfaces defined by the STI APIs specification. The infrastructure exposes a standard set of method names to the applications to facilitate portability. Although the STI infrastructure may use any combination of OS, BSP functions, or other infrastructure methods to implement a radio function, which may vary on different platforms, the STI APIs will be the same to allow portability. The STI APIs are the well-defined set of interfaces used by STI applications to access specific radio functions or used by the infrastructure to control the applications.

The infrastructure is composed of multiple subsystems that provide the functionality to operate the radio. The components shown in figure 13, Infrastructure, represent the high-level subsystems and services needed to control STI applications within the STI platform. These services are provided by the platform infrastructure and support applications as they execute within the STI platform. The infrastructure functions will include fault management techniques, which are necessary to increase radio robustness and support mission-dependent requirements.

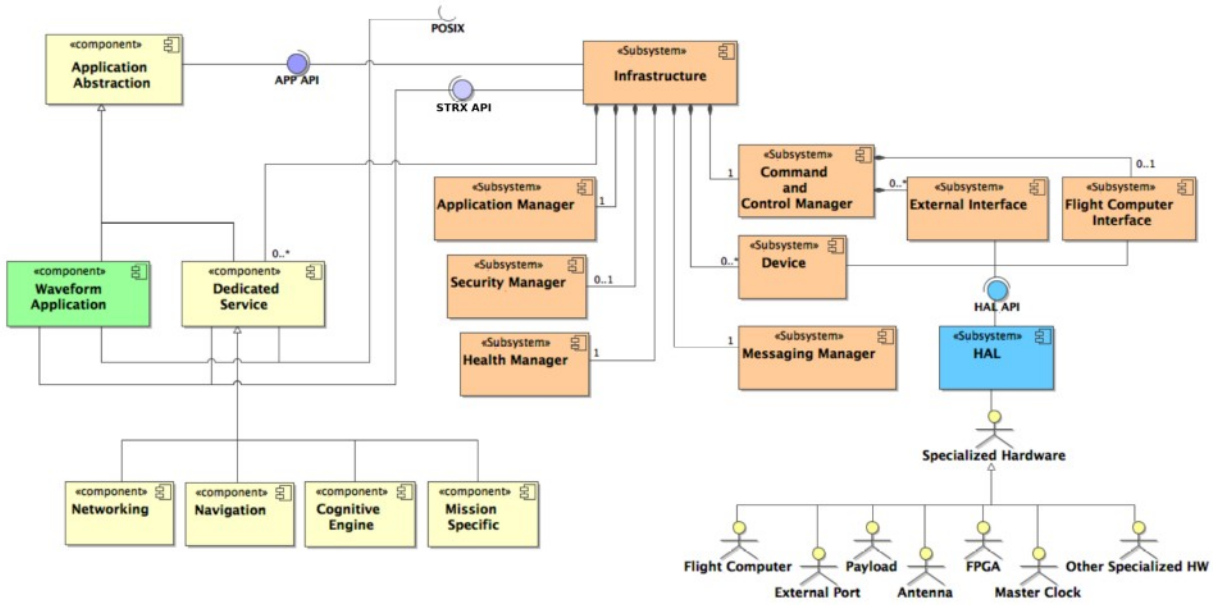

Figure 13: Infrastructure

10.3 API Overview

The STI APIs provide an open software specification so that the application engineers can develop STI applications. The goal is to have a standard API available to cover all application program requirements so that the application programs can be reused on other hardware systems with minimal porting effort and cost for the application implemented in software and/or configurable hardware design with increased reliability. Size, weight, and power constraints may limit the functionality of the radio by imposing a tradeoff among the following:

- The size of the API implementation,
- The size of other internal operations, and
- The size of the waveforms and services.

The size of the selected GPP should be sufficient to contain the OS, the STI infrastructure, and the appropriate portion of the waveforms and services to implement the required mission functionality, along with sufficient margin to support software upgrades. The STI APIs are defined to support internal radio commands. Any external interface commands, described in section 11, use the internal commands defined by the STI APIs to accomplish normal radio operations.

The API layer specification decouples the intellectual property rights of platform, application, and module developers. This allows development and interoperability of different radio aspects while protecting the investment of the developers.

The APIs in the following sections are grouped by type to simplify the description of the APIs while providing the detail for each requirement in tabular form. The table contains the name, description, parameters, return type, any additional information that is pertinent to the usage of that function. The examples shown in the table for each requirement are written from the point of view of the STI application developer.

Handle names and identifiers (i.e. `HandleID` values) have global scope within the operating environment. That is, a given identifier refers to the same application, device, file, queue, timer, or service across all applications.

A key aspect of a software architecture is the definition of the APIs that are used to facilitate software configuration and control of the target platform. The philosophy on which the STI architecture is based avoids the conflict between open architecture and proprietary implementations by specifying a minimum set of APIs that are used to execute waveform applications and to deliver data and control messages to installed hardware components. The following APIs exhibit similar functionality to a resource interface in the Object Management Group (OMG)/software radio (SWRADIO) or Software Communications Architecture (SCA 2.2.2) specifications.

10.3.1 Interface Structure

Figure 14, Application and Device Structure, shows a high-level overview of the STI software interface and object definitions.

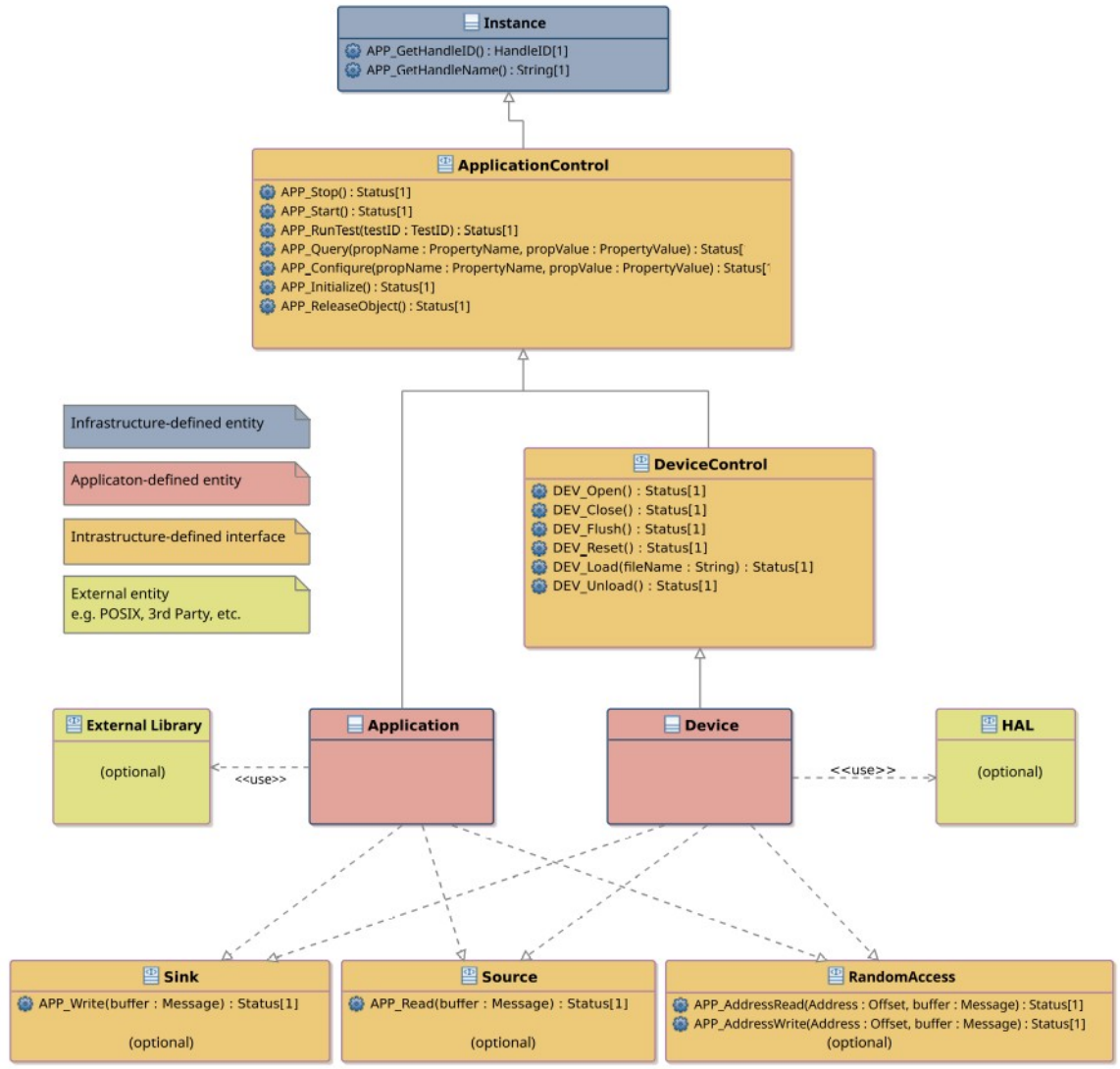

Figure 14: Application and Device Structure

As shown in figure 14, all applications and devices within the environment are derived from the `Instance` type, which is provided by the OE and serves as a common basis point for every entity. This base type has only has a

minimal set of infrastructure-defined methods. All operations are defined through several control interface definitions.

The operations include:

- A means for the application or device to obtain the corresponding name and ID.
- A means to configure or query the entity state from other applications, using name/value pairs.
- A means to execute tests on the application or device
- A means to dynamically start or stop a device or service from other applications
- A means to dynamically allocate/initialize system resources when needed and release resources when no longer necessary.
- A means to "read" or pull data from this entity to other applications
- A means to "write" or push data to this entity from other applications

An STI application implementation (e.g., waveform) would typically implement the `ApplicationControl` interface, which includes all operations relevant for applications except those related to data transfer. An STI device would implement the `DeviceControl` interface, which provides operations specific to devices as well as all operations defined by `ApplicationControl`. Any application or device may selectively choose to implement any of the data transfer interfaces as necessary, including `Source`, `Sink`, and `RandomAccess`.

Note that from the STI perspective, "Applications" and "Devices" are very similar concepts, differing only in that a device implements the operations specified in the `DeviceControl` interface, whereas an application typically does not implement these operations. Otherwise, the two software modules are identical. Throughout the remainder of this section, the term "Application" is used, but the same features and requirements generally apply to devices as well.

Summary of Infrastructure Software Requirements

► The STI infrastructure provider shall document the supported language interface(s) that are provided by the infrastructure, specifying any relevant standards or language revisions.

► The STI infrastructure shall use the STI Application-provided Application Control Interfaces to control STI applications.

10.3.2 Implementation

An STI operating environment may support applications written in any language, so long as it provides the infrastructure API in an appropriate form for the language in use. The software interfaces in this specification utilize the OMG Interface Description Language (IDL) syntax, and IDL language mappings provide a method to consistently translate the semantics of a given interface to many different programming languages.

To ensure naming consistency across differing OE implementations, a specific header file/module/namespace must be implemented such that the same function names are present and available on all STI implementations. Each programming language environment has differences in the paradigms used for this purpose.

The general STI architecture can also be implemented in programming languages using the translations prescribed by the IDL specification. Additional directives on how the IDL translations apply to the STI applications and infrastructure is available in Appendix 11, Language Translations. This section is intended to clarify certain aspects of the interface translation for commonly used programming languages, but other language translations beyond what is specified here are also possible. The appendix may be extended in a future revision of this specification to contain additional language mappings.

Nearly all modern high-level programming languages support some notion of "packages" or "modules" to separate functionality into logical entities. Whenever possible, all STI functionality should be encapsulated in a single package or module called "STI". Note that some languages, such as Java, dictate additional package naming recommendations. Any such language-specific package name recommendations should also be adhered to. In C and C++, the interfaces are available through multiple header files, as indicated in Annex A: Language Translations.

Language	Module Namespace	Example usage in application code
C	N/A (prefix)	`#include "STI.h" …`
C++	`STI`	`#include "STI.hh" …`
Java	`org.omg.STI`	`import org.omg.STI.*`
Python	`STI`	`import STI`
Perl	`OMG::STI`	`use OMG::STI`
Ruby	`STI`	`require 'STI'`
Lua	`STI`	`STI = require "STI"`

Table 4: Platform-Specific Naming Conventions

After utilizing the language-specific import statement, all components of the STI API can be referenced using the paradigm of the respective language's package/module facility. For instance, in languages such as Python, Ruby and Lua, the module contents are designated by the module name and a period (.) separator, for example the term "`STI.Initialize`" would refer to the `Initialize` function within the `STI` module.

Summary of Requirements for Software Modules

► Applications shall use the respective programming language's designated facilities, such as a package, module, or header file(s), to refer to all STI infrastructure-provided entities as prescribed in Annex A: Language Translations.

The specific syntactical requirements for application code depend on the specific language in use, whether it is C, C++ or some other language. However, the same general interface structure described earlier can be applied to many different languages.

All object-oriented languages such as C++, Java, and Python generally support the same fundamental concepts of inheritance and interfaces. For these languages, the interface translation is fairly straightforward, and the application should use the language's native inheritance mechanisms. For other languages such as C which are not natively object oriented, the approach differs slightly, but many of the same concepts can still be employed even if not directly supported by the language. Therefore, a different set of requirements will apply to applications implemented in C versus other object-oriented languages.

All STI applications and devices must encapsulate their state in an object or structure of some type, referred to as the "base object". Even for "singleton" objects of which there can only be one, STI dictates that there must still be a base object associated with it, even if this object does not contain any extra information.

Figure 14 also shows several different optional interfaces that an application or device may implement, depending on its specific design needs. In object oriented languages, the set of interfaces is indicated in the object definition, using the language's inheritance mechanisms. In these languages, a "connection" between the implementation and interface is automatically made through the language's type system. In non-object oriented languages, such as C, there must be a separate mechanism to explicitly create the connection between a given implementation to the interface it implements. For STI, a naming convention is employed to facilitate this connection.

Summary of Requirements instance object definition:

► Application object definitions use the programming language's inheritance mechanisms to specify the set of STI interfaces that are implemented by the application (for object oriented languages only).

► The application base object must be convertible to an `Instance` object as defined by the STI infrastructure.

In object oriented languages, the conversion to an Instance object is achieved by simply inheriting from the proper base class. In non-object-oriented languages, the application developer must implement this conversion, and it is not specified how the conversion takes place. For a singleton object, this can be a simple global. In C, this could be performed using a pointer conversion of some sort. Alternatively, this could be implemented using a lookup table or dictionary.

10.4 Data Types and Constants

The following data types are defined by the infrastructure. These types serve as the basis for the STI interfaces and API calls described in the subsequent sections.

10.4.1 Data Types

The STI infrastructure defines the following basic data types. For these data types, the specification allows some flexibility in how they are implemented by the infrastructure. This table only indicates only the general behavioral semantics of the type, such as an integer, string, or enumeration. For instance, all types with integer semantics should be compatible with the standard integer assignment and relational operators per the language in use.

For enumerated types, the possible values and definitions are shown in section 10.4.2, Constants.

Type Name	Semantics	Usage/Description
`Access`	Enumeration	Indicates desired access to a file. The specific possible values are described in Table 6.
`BufferSize`	Integer	Used to represent a buffer size in bytes. The range of the number is to be long enough to contain the maximum number of bytes to reserve or to transfer with a read or write.
`CalendarKind`	Enumeration	Identifies a specific method of time representation, such as TAI or UTC. The specific possible values are described in Table 7, CalendarKind Constants.
`CalendarTime`	Abstract Structure or Class	An abstract object that identifies a specific time for a particular `CalendarKind`. All possible `CalendarTime` values must be representable as a pointer or reference to this type.
`FileSize`	Integer	Used to represent a size in bytes. The range of the number is to be long enough to contain the largest possible number of bytes in GPP storage, as well as uniquely identifiable invalid values to indicate error conditions.
`HandleID`	Integer	Used to represent an STI application, device, file, or queue. The infrastructure defines the set of valid values for this type.
`Message`	Abstract Structure or Class	The base type of all data exchange (`Read`, `Write`) buffers. All STI data exchange messages must be must be representable as a pointer or reference to this type.
`Instance`	Structure or Class (base type)	The base type of all application and device context objects. All STI components must have a corresponding object of this type stored by the infrastructure, although the object itself is not exposed to

		other applications.
`Nanoseconds`	Integer	Used to indicate the number of nanoseconds (fractional part) within a `TimeWarp` object. Must be capable of representing at least the range of [0, 999999999], and may be implemented using an "unsigned" value type, if available.
`Offset`	Integer	Used to indicate an offset from the beginning of a file or device address space. Must be large enough to represent the last position in the largest file or device in the system. May be implemented using an "unsigned" value type, if available.
`PropertyName`	Integer, Enumeration or String	Used to identify properties by name. May be implemented as a numeric enumeration in languages which support this, or as a string value in other environments.
`PropertyValue`	Abstract Structure or Class	The base type of all property values used with the property set interface (`Configure`, `Query`). All STI property values must be must be representable as a pointer or reference to this type.
`QueueMaxMessages`	Integer	Used to represent the maximum number of messages allowed in a FIFO queue.
`Result`	Integer	Used to represent a status value, returned by many STI API calls. Specific predefined values represent error conditions, which are distinct from the set of valid results. See constants defined in Table 9, Result Constants.
`Seconds`	Integer	Used to indicate the number of seconds (whole number part) of a `TimeWarp` object. Negative values represent time intervals in the past, and positive values indicate time intervals in the future.
`TimeWarp`	Integer or Aggregate value (non-abstract)	The representation of an arbitrary time interval. Logically, this is a single, large value of fixed-point precision. The value should be at least 64 bits in size. If the largest native integer size is less than 64 bits on a given architecture, this may be defined as a structure to achieve the necessary range and precision. Units are implementation defined, but must be convertible to seconds and nanoseconds.
`TimeRate`	Integer	Used to indicate the adjustment factor of clock devices during adaptive sync and drift compensation. Positive values represent increased clock frequency/tick rates, negative values represent decreased frequency/tick rates, and a value of zero represents the nominal or "free-run" clock frequency. Units are implementation defined.

Table 5: Infrastructure-provided Data Types

10.4.2 Constants

The STI infrastructure defines the following constants. In strongly-typed languages, the constant shall evaluate to a value of the specific data type as indicated.

Declaration	enum Access { READ, WRITE, APPEND, BOTH };
Description	Indicates desired access to a file.
Usage	► `READ`: Indicates file “read” permission ► `WRITE`: Indicates file “write” permission, i.e. writing to beginning of file ► `APPEND`: Indicates file “append” permission, i.e. writing to end of file ► `BOTH`: Combination of `READ + WRITE` permissions.
Provided By	Infrastructure
Notes	Used exclusively by the `FileOpen()` API call

Table 6: Access Constants

Declaration	const CalendarKind TAI = {…}; const CalendarKind UTC = {…}; const CalendarKind GPS = {…}; const CalendarKind MJD = {…}; const CalendarKind LOCAL = {…};
Description	Identifies several well-defined time and date representations
Usage	► `TAI`: Corresponds to the International Atomic Time, a monotonically increasing time scale based on the average of numerous Earth-based atomic clocks ► `UTC`: Corresponds to the Coordinated Universal Time, which is offset from TAI by a number of leap seconds that is occasionally updated through international consensus ► `GPS`: Corresponds to the GPS time scale, a count of weeks and seconds since the GPS epoch. ► `MJD`: Corresponds to Modified Julian Date, which is a floating point representation of Earth days since the MJD epoch. ► `LOCAL`: Corresponds to the default local time representation. This is implementation-defined.
Provided By	Infrastructure
Notes	Platforms do not need to implement every defined calendar system. For those that are implemented, they should be implemented in a manner consistent with the name and specification indicated. Implementations may also define custom `CalendarKind` values for application-specific needs. Use of the `LOCAL` time and date representation in applications is discouraged, due to the inherent ambiguity. This is intended only for a user interface or display purpose. For more information on the specific time structures associated with these time and date representations, see section 10.6.8.

Table 7: CalendarKind Constants

Declaration	const HandleID HANDLEID_INVALID = {…}; const HandleID WARNING_QUEUE = {…}; const HandleID ERROR_QUEUE = {…}; const HandleID FATAL_QUEUE = {…}; const HandleID TELEMETRY_QUEUE = {…};
Description	A set of pre-defined values of the `HandleID` type
Usage	► `HANDLEID_INVALID`: A reserved value that will never alias a valid handle ID ► `WARNING_QUEUE`: The default queue to use in conjunction with for the Log() API for context information related to WARNING responses ► `ERROR_QUEUE`: The default queue to use in conjunction with for the Log() API for context information related to ERROR responses ► `FATAL_QUEUE`: The default queue to use in conjunction with for the Log() API for context information related to FATAL responses ► `TELEMETRY_QUEUE`: The default queue for general system telemetry data. The purpose and usage of this queue handle is implementation-defined.
Provided By	Infrastructure
Notes	The `HANDLEID_INVALID` constant is intended for use as an initializer, to avoid ambiguity in locally-instantiated `HandleID` values. For instance, this can be used within an initializer list in a C++ class constructor, before the member is set to a real handle ID, to avoid potential undefined behavior if the destructor is invoked before the value is set to an actual handle ID. Note: If checking return values, applications should never check for specifically for the `HANDLEID_INVALID` value, but rather use the `ValidateHandleID()` API call.

Table 8: HandleID Constants

Declaration	const Result OK = {…}; const Result WARNING = {…}; const Result ERROR = {…}; const Result FATAL = {…}; const Result UNIMPLEMENTED = {…};
Description	A set of pre-defined values of the `Result` type
Usage	► `OK`: Indicates the operation was successful ► `WARNING`: Indicates the operation was not successful, but little or no corrective action is required. The component is still operational; this may be a transient error. ► `ERROR`: Indicates the operation was not successful, and some corrective action may be required. The component is still operational. ► `FATAL`: Indicates the operation was not successful, and significant corrective action is required. The component is not able to function. ► `UNIMPLEMENTED`: Indicates that the operation it is not implemented by the component or by the infrastructure.
Provided By	Infrastructure

Notes	Values other than `OK` may also indicate success. Applications should never check for this value specifically, but rather use `IsOK()` to determine if an operation succeeded. An `ERROR` indicates component is operational, but the request may not be applicable to the component or may not be valid per the current component state. The caller should take action to correct the underlying issue before attempting the call again. The `UNIMPLEMENTED` value is intended to differentiate between a request that was successfully sent to the target but failed to execute, versus a request that was not sent to the target because it does not implement an optional interface. This may be treated similarly to an `ERROR` response.

Table 9: Result Constants

Declaration	const string OE_HANDLE_NAME = "…"; const string DEFAULT_CLOCK_NAME = "…";
Description	A set of pre-defined handle names
Usage	`OE_HANDLE_NAME`: A name identifying the operating environment `DEFAULT_CLOCK_NAME`: A name identifying the default system clock device
Provided By	Infrastructure
Notes	These names may be passed to `HandleRequest()` to find the corresponding handle ID, which can then be used to interact with the target component.

Table 10: Handle Name Constants

Declaration	const PropertyName COMPONENT_PROVIDER = {…}; const PropertyName COMPONENT_VERSION = {…}; const PropertyName COMPONENT_STATE = {…};
Description	A set of pre-defined property names
Usage	► `COMPONENT_PROVIDER`: A name associated with the provider of the component. ► `COMPONENT_VERSION`: A name associated with the version of a component. ► `COMPONENT_STATE`: A name associated with the state of a component.
Provided By	Infrastructure
Notes	All applications, as well as the operating environment, must implement these property names. Devices may also implement these property names but it is not required; for any devices provided by the platform, the values would generally match that of the OE. The values associated with these property names should be free-form strings The `PROVIDER` value is usually a company name or university, followed by a subsidiary, division, or department name. The `VERSION` value is implementation-specific, and may be of the format `MAJOR.MINOR.REVISION` and may also include additional identification information, such as a baseline version control revision ID or tag/branch if relevant. The `STATE` value is implementation-specific, and the meaning should be indicated by the application developer.

Table 11: Property Name Constants

Declaration	const BufferSize MAX_PROPERTY_NAME_SIZE = {…}; const BufferSize MAX_PROPERTY_VALUE_SIZE = {…}; const BufferSize MAX_PATH_NAME_SIZE = {…}; const BufferSize MAX_HANDLE_NAME_SIZE = {…}; const BufferSize MAX_LOG_MESSAGE_SIZE = {…}; const MaxQueueMessages MAX_QUEUE_MESSAGES = {…};
Description	Establishes a set of known maximum size limits for various items
Usage	► `MAX_PROPERTY_NAME_SIZE`: The maximum size, in bytes, of any `PropertyName` object ► `MAX_PROPERTY_VALUE_SIZE`: The maximum size, in bytes, of any `PropertyValue` object ► `MAX_PATH_NAME_SIZE`: The maximum length, in characters, of a file name ► `MAX_HANDLE_NAME_SIZE`: The maximum length, in characters, of a handle name ► `MAX_LOG_MESSAGE_SIZE`: The maximum length, in characters, of strings accepted by the `Log()` API ► `MAX_QUEUE_MESSAGES`: The maximum number of messages that can be stored in a queue.
Provided By	Infrastructure
Notes	These constant definitions are mainly intended for use in languages such as C/C++ where application developers are responsible for buffer allocation. In other languages, buffer allocation may occur automatically and as such these size limits may not be relevant. In C/C++ environments these constants must evaluate at compile time, such that they may be used as an array dimension. Note that for string length sizes, the value reflects the maximum number of actual characters in the string, and does *not* take into account the terminating NUL character. The value should always be increased by 1 if the constant is used to as the dimension of a `char[]` array.

Table 12: Size Limit Constants

Declaration	const TimeWarp TIME_INTERVAL_ZERO = {…}; const TimeWarp TIME_INTERVAL_UNLIMITED = {…};
Description	Constant values suitable for usage with functions accepting a `TimeWarp` value
Usage	► `TIME_INTERVAL_ZERO`: Represents the value of zero ► `TIME_INTERVAL_UNLIMITED`: A value indicating no limit to the respective time interval or step size.
Provided By	Infrastructure
Notes	The `TIME_INTERVAL_UNLIMITED` constant is intended be used with functions such as `TimeSynch()`. When this value is passed as the `stepMax` argument, it indicates that the infrastructure may directly step the clock to any value.

Table 13: TimeWarp Constants

10.5 Application and Device Control Interface

The application and device interface, illustrated in figure 14, is the mechanism through which local applications receive requests from the STI infrastructure.

All operations described in this section operate on a single context object, which is a data structure stored in local memory that contains the state of the application instance. The specific semantics of this context object depend on the language in use. In C, this context object is passed explicitly as a pointer argument to each call, which can then be cast or converted to the correct structure type. In C++ or Java, these operations are implemented as class member functions, and as such the context object is passed implicitly through the `this` reference. Other object oriented languages have a similar paradigm to reference the context object, such as the "self" object in Python.

As a general convention, interfaces that apply to all components (applications, devices, etc.) have operations named with an `APP` prefix, and interfaces that apply only to devices have operations named with a `DEV` prefix. Further details on each of these operations are provided in the following sections.

Note that the operations listed in this section are *not* invoked directly by other applications or components in the system. The infrastructure is responsible for managing the life cycle of all context objects, and these objects are not directly exposed to other components in the system. All operation requests from other components must go through the STI infrastructure, which may in turn invoke a context switch or middleware as needed, to provide the correct context for the subsequent operation. For every interface operation described in this section, there is a corresponding infrastructure-provided API call that operates on an abstract handle value rather than a context object. These handle-based API calls, as described in section 10.6 , are intended to be invoked from other entities.

10.5.1 Infrastructure-Provided Component Identifier Interface

The interface operations described in this section must be provided by the infrastructure, and may be invoked by an application or device to obtain information from the infrastructure. The interface provides a consistent means for an application or device to obtain identification information about itself.

Declaration	interface Instance { HandleID APP_GetHandleID(); ... };
Description	Obtain the handle ID for the application, stored by the OE.
Return	The actual handle ID of the calling application, or an invalid handle ID on failure
Implemented By	Infrastructure
Invoked By	Application
Notes	This call should never fail when invoked from a normal, fully-constructed application or device context. If invoked from an application or device context that is not fully constructed, an invalid ID may be returned. Specifically, this condition may occur while the constructor or destructor are currently executing (see section 10.5.2.1). In the event that the infrastructure cannot obtain the correct handle ID, the infrastructure shall return the value `HANDLEID_INVALID`; it must not return a valid (but incorrect) handle ID.

Table 14: APP_GetHandleID() Definition

Declaration	interface Instance { string APP_GetHandleName(); ... };

Description	Obtain the name for the application, stored by the OE.
Return	The name of the calling application, or a NULL/undefined value on failure
Implemented By	Infrastructure
Invoked By	Application
Notes	This call should never fail when invoked from a normal, fully-constructed application or device context. If invoked from an application or device context that is not fully constructed, this call may fail. Specifically, this condition may occur while the constructor or destructor are currently executing (see section 10.5.2.1). In the event that the infrastructure cannot obtain the correct handle name, the infrastructure should return an invalid value appropriate for the respective programming language, such as NULL in C/C++ or the corresponding undefined value representation in other languages. It should not return a valid string in this case.

Table 15: APP_GetHandleName() Definition

10.5.2 Application-Provided Application Control Interfaces

The operations detailed in this section must be provided by the application developer.

10.5.2.1 Constructor and Destructor

For all applications, constructor and destructor functions must be provided by the application developer. These functions will create and destroy an instance of the respective application's state structure, as an object of the `Instance` base type.

For applications or services that are instantiated multiple times within a single environment, the constructor will be invoked by the infrastructure for each instance. After construction, the `Instance` reference identifies the specific context object to work with for all subsequent calls interface operations. In C++ terminology, it equates to the `this` pointer.

The notion of a statically-allocated "singleton" object is allowed, but the application must still supply a stub function for use as a constructor and destructor. In this case, the constructor may directly return the statically-allocated instance, and the destructor may be empty.

Note that these methods implement the "factory" pattern in object oriented design. As such, they are not instance methods, but rather static methods when translated to object oriented environments.

Declaration	`Instance APP_Instance(` `  in HandleID id,` `  in string name` `);` `...`
Description	Construct an instance of the application, identified by the `id` and `name` indicated in the parameters.
Return	On success, return a reference to the constructed instance. On failure, return an invalid reference (i.e. NULL in C/C++, or the respective undefined value in other languages)
Implemented By	Application
Invoked By	Infrastructure

Notes	The `id` and `name` values passed to this constructor become valid only *after* the constructor has completed successfully and returned a valid object reference/pointer. As such, other infrastructure calls should not be invoked from the constructor using these values. Use of the values during the construction of the object itself is not defined, as the infrastructure may still consider it an invalid ID or name. For statically allocated objects, a pointer to the pre-allocated structure may be returned, without performing any additional allocation. In all cases, the object returned must be of the Instance type, either directly or as a derivative type. In object-oriented languages, the instance object must inherit from the correct base object or class. In C, this can be done by ensuring the first member of the returned structure object is an `Instance` object as defined by the infrastructure.

Table 16: APP_Instance() Definition

Declaration	```void APP_Destroy(in Instance inst); ...```
Description	Delete an instance of the application, identified by the `inst` parameter.
Return	None
Implemented By	Application
Invoked By	Infrastructure
Notes	This function must be defined, but may be empty or a “no-op” for statically-allocated entities. After this call completes, the object referred to by the `inst` parameter is considered invalid, and the infrastructure ensures that any internally-stored references to the instance have been deleted.

Table 17: APP_Destroy() Definition

10.5.2.2 Life Cycle Interface

The Life Cycle interface is intended to provide additional control over the application start up/initialization and shutdown processes. In many cases, an application will require some allocation steps which are dependent on configuration, such as storage buffer sizes, and these configuration items may not be known at the time the constructor is invoked. This interface allows the initialization of the application to be separated from the instantiation of the application. The required application properties can then configured after instantiation but before the initialization takes place.

Declaration	```interface ApplicationControl { Result APP_Initialize(); ... };```
Description	Initialize the application. Obtain any underlying system resources as required for operation, and set all internal variables to a known initial state.
Return	Status code which the caller should validate using `IsOK()` On failure, returns one of the defined `Result` error constants. On success, return the status code `OK`.

Implemented By	Application
Invoked By	Infrastructure
Notes	If initialization is unsuccessful for any reason, the implementation must ensure that any external system resources obtained before the failure are returned to their original state. There is no provision to permit "partial" initialization sequences to occur. If not successful, the implementation should log details of the failure to the log facility.

Table 18: APP_Initialize() Definition

Declaration	interface ApplicationControl { Result APP_ReleaseObject(); ... };
Description	Release any system resources that were obtained during the initialization operation.
Return	Status code which the caller should validate using `IsOK()` On failure, returns one of the defined `Result` error constants. On success, return the status code `OK`.
Implemented By	Application
Invoked By	Infrastructure
Notes	This operation should be the inverse of the `APP_Initialize()` operation, returning the application or device to the same state as it was prior to initialization. After this operation, the infrastructure must either destroy the instance or initialize it again.

Table 19: APP_ReleaseObject() Definition

10.5.2.3 Property Set Interface

The Property Set interface consists of two operations, configure and query, which operate on name/value pairs. The implementation should perform all necessary validation of the input parameters, including whether the property name specified is valid, and whether it is permissible to set or retrieve the value in the current application state. The notion of a "read-only" property is also allowed, where any attempt to configure such properties returns the `ERROR` status code.

Declaration	interface ApplicationControl { Result APP_Query(in PropertyName propName, out PropertyValue propValue); ... };
Description	Obtain or "get" the value for one property in the component
Parameters	► `propName`: The name or identifier of the property to get ► `propValue`: A buffer to store the property value
Return	Status code which the caller should validate using `IsOK()` On failure, returns one of the defined `Result` error constants. The status code `OK` indicates that the property value has been retrieved in its entirety.

Implemented By	Application
Invoked By	Infrastructure
Notes	If an error is returned by an implementation, a corresponding message indicating details of the failure should be written to the log facility for diagnostic purposes. Return status codes other than the defined status constants are permissible for backward compatibility, but must validate using the `IsOK()` function. Use of additional codes is not recommended for new software; for maximum portability, custom status codes or "partial success" return codes should be avoided. For C/C++ implementations, the abstract `propValue` parameter is translated to two parameters, a base object pointer and size.

Table 20: APP_Query() Definition

Declaration	interface ApplicationControl { Result APP_Configure(in PropertyName propName, in PropertyValue propValue); ... };
Description	Configure or "set" the value for one property in the component
Parameters	► `propName`: The name of the property to set ► `propValue`: The value to set the property to
Return	Status code which the caller should validate using `IsOK()` On failure, returns one of the defined `Result` error constants. The status code `OK` indicates that the property value has been configured.
Implemented By	Application
Invoked By	Infrastructure
Notes	If an error is returned by an implementation, a corresponding message indicating details of the failure should be written to the log facility for diagnostic purposes. Status codes (other than the defined status constants) are permissible for backward compatibility, but must validate using the `IsOK()` function. This is not recommended for new software; for maximum portability, custom status codes or "partial success" return codes should be avoided. For C/C++ implementations, the abstract `propValue` parameter is translated to two parameters, a base object pointer and size.

Table 21: APP_Configure() Definition

10.5.2.4 Test Interface

The test interface provides a means to invoke any built in testing routines. Test routines are identified by a test ID, which is a numeric value that the application must define.

The application developer is responsible for documenting the test ID's which are implemented, including the purpose and any restrictions or dependencies associated with the test. For example, tests targeted toward finding manufacturing or assembly defects may only be executable as a "ground test" when the system is connected to a

designated test facility. Other tests may be permissible during run-time or flight operations, but may interfere with normal radio communication.

Tests may be implemented either synchronously or asynchronously (i.e. as a background operation). For synchronous tests, the status returned indicates the complete test result, with passing indicated by returning a successful status code. For asynchronous tests, the status returned indicates only if the test has been initiated. The application implementation should utilize the PropertySet interface and specify property names/values to communicate the progress and results of the test.

Declaration	interface ApplicationControl { Result APP_RunTest(in TestID testID); ... };
Description	Invokes the test of the target application as indicated by the test ID.
Return	Status code which the caller should validate using `IsOK()` On error, returns one of the defined `Result` error constants. A successful result (status code `OK`) indicates that the test is successful or that the test is running in the background.
Implemented By	Application
Invoked By	Infrastructure
Notes	Tests which are not appropriate for a given system state, such as invoking a ground-specific test while in a flight operation mode, should generate an error status return and record the issue in the system log.

Table 22: APP_RunTest() Definition

10.5.2.5 Controllable Component Interface

The ControllableComponent interface is intended for applications or devices to enter or exit their normal operation mode after initialization. Typically, this should not involve any additional allocation or resource acquisition, but it should only activate or deactivate the previously allocated resources.

For example, in an application designed to estimate incoming signal power, the `Initialize` operation (described in section 10.5.2.2, Life Cycle Interface) would allocate any buffer storage and set up the resources necessary to "tap" the incoming signal samples, but would not actually start or activate the power estimation algorithm. The `Start` operation described here would begin the process of taking snapshots of the incoming data and executing the power estimation algorithm. Similarly, the `Stop` operation would stop the active process, but it would not tear down or release any buffers or other system resources, which is the domain of the LifeCycle interface.

This interface is also applicable to devices which have the notion of a "standby" state; after initialization, the device would become ready but not active. The `Start` and `Stop` operations would put the device into its active or standby state, respectively.

Declaration	interface ApplicationControl { Result APP_Start(); ... };
Description	Begin normal target component (application or device) processing.

Return	Status code which the caller should validate using `IsOK()` On error, returns one of the defined `Result` error constants. On success, return the status code `OK`.
Implemented By	Application
Invoked By	Infrastructure
Notes	If the application is not in the appropriate internal state, then nothing is done and an error is returned. If an error is returned by an implementation, a corresponding message to indicate details of the failure should be written to the log facility for diagnostic purposes.

Table 23: APP_Start() Definition

Declaration	interface ApplicationControl { Result APP_Stop(); ... };
Description	End normal target component (application or device) processing.
Return	Status code which the caller should validate using `IsOK()` On error, returns one of the defined `Result` error constants. On success, return status code `OK`.
Implemented By	Application
Invoked By	Infrastructure
Notes	If the application is not in the appropriate internal state, then nothing is done and an error is returned. If an error is returned by an implementation, a corresponding message to indicate details of the failure should be written to the log facility for diagnostic purposes.

Table 24: APP_Stop() Definition

10.5.3 Device-Provided Device Control Interface

An STI Device is a proxy for the data and/or control path to the actual hardware. An STI Device is a "bridge" used to decouple an abstraction from its implementation so that the two can vary independently. All operations detailed in this section must be provided by the device developer or platform provider. Like the application control interface, all operations described in this section are invoked by the STI infrastructure based on requests from other entities within the environment. The operations listed below are *not* invoked directly by other applications.

The STI Device may be implemented using any available platform-specific hardware access layer to communicate with and control the specialized hardware. While portability is not a specific goal for devices, if the hardware access layer is also standardized and/or adheres to commonly implemented patterns, then the STI device itself can also potentially be re-used in other environments with minimal modifications.

For example, many UNIX and UNIX-like RTOS operating systems implement a very similar pattern to configure and access a serial device, using a pseudo-file in the `/dev` filesystem combined with a defined set of `ioctl()` operations and "termios" C library calls. As such, an STI device abstraction for UNIX-style serial ports and other serially-connected devices could be shared among any operating environment using this style of operating system and device model. In contrast, an operating systems such as Microsoft Windows® utilizes a driver architecture specific to itself, and as such any STI device abstractions written using this driver model are not likely to be portable

to any other operating system. However, in either case, an STI-compliant application that accesses serial devices using the STI device abstraction would be portable to either environment.

The basic operations listed in this section correspond to the DeviceControl interface as illustrated in figure 14.

Declaration	interface DeviceControl { Result DEV_Open(); ... };
Description	Open the device for command and control
Return	On error, returns one of the defined `Result` error constants. On success, return status code `OK`.
Implemented By	Device or Platform Provider
Invoked By	Infrastructure
Notes	The implementation should obtain whatever operating system or HAL resources are necessary to initiate communication or data transfer with the hardware device. Depending on the underlying device and operating system driver infrastructure, use of a hardware device may be limited to one process at a time, so a successful call to this function may prevent other processes in the system from using the device. Likewise, if another process is using the device, or the device is otherwise not able to accept control requests, this operation may fail or block until the device becomes available. If no specific operating system resources are required for communication with the device, this implementation may be a no-op. In this case, this operation should return `OK` to maintain compatibility.

Table 25: DEV_Open() Definition

Declaration	interface DeviceControl { Result DEV_Load(in string fileName); ... };
Description	Load a binary application image or configuration file to the device
Parameters	► `fileName`: name of the image or configuration file to load to the device
Return	On error, returns one of the defined `Result` error constants. On success, return status code `OK`.
Implemented By	Device or Platform Provider
Invoked By	Infrastructure
Notes	If the device is an FPGA, this operation would load a specific hardware design image to the device. If the device represents a microcontroller or DSP, this should load a firmware or application image to the device.

Table 26: DEV_Load() Definition

Declaration	interface DeviceControl { Result DEV_Reset(); ... };
Description	Initialize a device to a known state.
Return	On error, returns one of the defined `Result` error constants. On success, return status code `OK`.
Implemented By	Device or Platform Provider
Invoked By	Infrastructure
Notes	This operation should bring a device into a known clean state, if possible. This operation may utilize a hardware reset function if available, or it may reconfigure all internal registers to a known initial value. This function should not "unload" programming information from an FPGA device. If a hardware reset function is used and this clears the programming information, the implementation should ensure that previously-loaded image is restored before returning.

Table 27: DEV_Reset() Definition

Declaration	interface DeviceControl { Result DEV_Flush(); ... };
Description	Clear any pending input/output buffers associated with the device
Return	On error, returns one of the defined `Result` error constants. On success, return status code `OK`.
Implemented By	Device or Platform Provider
Invoked By	Infrastructure
Notes	This operation should ensure that any existing data that may be buffered within the hardware device or control software is cleared, such that subsequent read operations (for source devices) or write operations (for sink devices) only transfer new data. It is implementation-defined how existing data that has not yet been fully transferred is handled. On a sink device, the operation may wait until the data is transferred, or the data may be discarded, depending on what is more appropriate for the device and the system context. On a source device, any received but unread data should typically be discarded. The device developer or platform provider should document the behavior of this operation.

Table 28: DEV_Flush() Definition

Declaration	interface DeviceControl { Result DEV_Unload(); ... };
Description	Unload a binary image or configuration file to the device
Return	On error, returns one of the defined `Result` error constants. On success, return status code `OK`.

Implemented By	Device or Platform Provider
Invoked By	Infrastructure
Notes	This operation clears any programming information from the device. Ideally this should be the inverse of the `DEV_Load()` operation. If the device does not support this operation, this may be implemented as a "no-op".

Table 29: DEV_Unload() Definition

Declaration	`interface DeviceControl {` `  Result  DEV_Close();` `  ...` `};`
Description	Closes the device
Return	On error, returns one of the defined `Result` error constants. On success, return status code `OK`.
Implemented By	Device or Platform Provider
Invoked By	Infrastructure
Notes	This operation should be the inverse of the `DEV_Open()` operation. If the open operation was a no-op, this operation should also be empty and it should return `OK` for compatibility.

Table 30: DEV_Close() Definition

10.5.4 Data Transfer Interface

The interfaces described in this section allow bulk data transfer between the component and the infrastructure. Like all other operations, this interface exists only between the infrastructure and the respective target components. The infrastructure is responsible for transporting the data between entities in the system.

The interfaces described in this section are optional. Applications or devices choosing to implement this interface must indicate this in the application declaration. In object oriented languages, this is done by inheriting or implementing the Source and/or Sink interface. In non-object oriented languages, it must be indicated in an OE-specific manner.

10.5.4.1 Source Interface

The Source interface is intended for applications or devices that supply arbitrary data to other entities using a "pull" model. The specific nature of the data is not defined by this specification and must be documented by the application developer. It may represent a stream of raw data, such as ADC samples, or it may be processed data, such as a power profile or constellation of the received signal.

Declaration	interface Source { Result APP_Read(out Message buffer); ... };
Description	The buffer is filled with data from the component.
Parameters	► `buffer`: a storage area for data transferred from the target
Return	Status code which the caller should validate using `IsOK()` On error, returns one of the defined `Result` error constants. On success, the return value indicates the number of units of data (records or bytes) actually obtained from the application or device, which may be less than the complete buffer size.
Implemented By	Application
Invoked By	Infrastructure
Notes	The actual storage for the buffer is allocated by the caller or infrastructure prior to invoking this function. The application should fill the buffer to the maximum extent possible and return the amount of buffer actually filled. The application developer defines the specific format and units for the buffer. In languages with direct memory access (e.g. C), it may be an arbitrary memory buffer with the units specified in bytes. In other languages, the units should reflect logical records, such as a number of characters, samples or objects. The infrastructure makes no assumptions about the format of the message data, nor the presence or expectation of a terminating entity, such as a NUL character as required for C-style strings. If a terminating character is required, the caller must ensure that sufficient space is available in the buffer to store the termination character. If an error is returned by an implementation, a corresponding message to indicate details of the failure should be written to the log facility for diagnostic purposes. For C/C++ implementations, the abstract `buffer` parameter is translated to two parameters, a base object pointer and size.

Table 31: APP_Read() Definition

10.5.4.2 Sink Interface

The Sink interface is intended for applications or devices that accept arbitrary data from other entities using a “push” model. Like the Source interface, the specific nature of the data is not defined by this specification and must be documented by the application developer. It may represent a stream of raw data, such as ADC samples, or it may be higher-level data structures.

Declaration	interface Sink { Result APP_Write(in Message buffer); ... };
Description	The buffer data is sent to the target component.
Parameters	► `buffer`: an abstract data set that should be transferred to the target

Return	Status code which the caller should validate using `IsOK()` On error, returns one of the defined `Result` error constants. On success, the return value indicates the number of units of data (records or bytes) actually sent to the application or device, which may be less than the buffer size.
Implemented By	Application
Invoked By	Infrastructure
Notes	The actual storage for the buffer is allocated and filled by the caller or infrastructure prior to invoking this function. The application should transfer the data to the maximum extent possible and return the amount of buffer actually transferred to the device. The application developer defines the specific format and units for the buffer. In languages with direct memory access (e.g. C), it may be an arbitrary memory buffer with the units specified in bytes. In other languages, the units should reflect logical records, such as a number of characters, samples or objects. The infrastructure makes no assumptions about the format of the message data, nor the presence or expectation of a terminating entity, such as a NUL character as required for C-style strings. If a terminating character is required, the caller must ensure that it has been added to the buffer prior to invoking this operation. If an error is returned by an implementation, a corresponding message to indicate details of the failure should be written to the log facility for diagnostic purposes. For C/C++ implementations, the abstract `buffer` parameter is translated to two parameters, a base object pointer and size.

Table 32: APP_Write() Definition

10.5.4.3 Random Access Interface

This optional device interface provides a means to read or write data directly to a specific location within a file or device. The location specified indicates the offset from the beginning of the file, address space, or memory map of the file or device. For memory-mapped entities or devices attached to some other physical bus (e.g. I^2C) this should translate to the respective bus cycles to read or write from the given location on that bus.

The register set exposed via this interface may be emulated; the implementation is free to translate or modify the request as needed by the underlying devices or hardware infrastructure. The physical bus access, if any, may go through one or more levels of indirection, and the actual physical addresses accessed may be different than the address requested.

Declaration	``` interface RandomAccess { Result APP_AddressRead(in Offset offset, out Message buffer); ... }; ```
Description	The buffer is filled with data from the component at the specified location
Parameters	► `offset`: the location to read data from ► `buffer`: a storage area for data transferred from the target

Return	Status code which the caller should validate using `IsOK()` On error, returns one of the defined `Result` error constants. On success, the return value indicates the number of units of data (records or bytes) actually obtained from the application or device, which may be less than the complete buffer size.
Implemented By	Application
Invoked By	Infrastructure
Notes	The actual storage for the buffer is allocated by the caller or infrastructure prior to invoking this function. The application should fill the buffer to the maximum extent possible and return the amount of buffer actually filled. The application developer defines the specific format and units for the buffer. In languages with direct memory access (e.g. C), it may be an arbitrary memory buffer with the units specified in bytes. In other languages, the units should reflect logical records, such as a number of characters, samples or objects. The infrastructure makes no assumptions about the format of the message data, nor the presence or expectation of a terminating entity, such as a NUL character as required for C-style strings. If a terminating character is required, the caller must ensure that sufficient space is available in the buffer to store the termination character. If an error is returned by an implementation, a corresponding message to indicate details of the failure should be written to the log facility for diagnostic purposes. For C/C++ implementations, the abstract `buffer` parameter is translated to two parameters, a base object pointer and size.

Table 33: APP_AddressRead() Definition

Declaration	``` interface RandomAccess { Result APP_AddressWrite(in Offset offset, in Message buffer); ... }; ```
Description	The buffer data is written to the target component at the specified location
Parameters	► `offset`: the location to write the data ► `buffer`: an abstract data set that should be transferred to the target
Return	Status code which the caller should validate using `IsOK()` On error, returns one of the defined `Result` error constants. On success, the return value indicates the number of units of data (records or bytes) actually sent to the application or device, which may be less than the buffer size.
Implemented By	Application
Invoked By	Infrastructure

Notes	The actual storage for the buffer is allocated and filled by the caller or infrastructure prior to invoking this function. The application should transfer the data to the maximum extent possible and return the amount of buffer actually transferred to the device. The application developer defines the specific format and units for the buffer. In languages with direct memory access (e.g. C), it may be an arbitrary memory buffer with the units specified in bytes. In other languages, the units should reflect logical records, such as a number of characters, samples or objects. The infrastructure makes no assumptions about the format of the message data, nor the presence or expectation of a terminating entity, such as a NUL character as required for C-style strings. If a terminating character is required, the caller must ensure that it has been added to the buffer prior to invoking this operation. If an error is returned by an implementation, a corresponding message to indicate details of the failure should be written to the log facility for diagnostic purposes. For C/C++ implementations, the abstract `buffer` parameter is translated to two parameters, a base object pointer and size.

Table 34: APP_AddressWrite() Definition

10.6 STI API

The API calls in this section comprise the "public" interface into the STI infrastructure, and may be used by all components in the system to initiate actions in other components. Operations primarily utilize handle ID values, which are opaque/abstract values that uniquely reference a single component within the STI infrastructure. The specific format or structure of the handle ID value is implementation-defined, but the following criteria must apply:

- Handle ID values apply within a single run-time instance of an STI operating environment. They are not meaningful outside the operating environment, nor are they meaningful in a different instance of an STI operating environment. Note that a "reboot" of an environment is considered a different run-time instance; handle ID values are not required to be persistent across restarts, and may be assigned differently.
- Handle ID values refer to the same component for that respective component's lifetime; a component cannot ever change its handle ID unless that component is destroyed and re-created.
- All components within the same operating environment can refer to the same set of handle IDs, and a given handle ID referenced from one component refers to the same entity as the same handle ID referenced from a different component.
- Two Handle ID values may be tested for equality using the programming language's normal equality check operator (e.g. `if (Handle1 == Handle2) …`), but all other inquiries or tests must be performed via the infrastructure.

Portable applications and devices must treat handle ID values as opaque objects, without any assumptions regarding the validity of specific values or the data type(s) capable of storing the value. Only the infrastructure-supplied `HandleID` type may be used to store a handle ID value.

It is recommended that the infrastructure implement handle IDs as an integer or a type derived from an integer, for speed and simplicity of operation, although this is not required. As such, a handle ID value should not be compared to any other integers.

10.6.1 General Utility API

The utility functions described in this section allow an application to make inquiries about the state of the infrastructure or a previous operation, and generally do not perform any operation of their own. These functions may be used at any time by any application.

10.6.1.1 Response Handling and Analysis

The function calls described in this section allow analysis of the return value of a previous call. Many STI API calls return one of four data types:

- A status code (`Result`)
- A handle ID (`HandleID`)
- A size (`FileSize`)
- A string (language-dependent)

In most circumstances, calls returning a `Result` type should use the defined value `OK` to indicate a successful result. However, there are some API calls, mainly those that use variably-sized data buffers for reading or writing, for which partial success is permissible. In these cases, the function returns an actual size or count value rather than a fixed value upon success. For this reason, portable applications must not directly check for the specific return value `OK` to determine success of any STI call. Instead, applications must use a second operation to check if a given status code is represents success or failure.

Similarly, operations that return a `HandleID` or `FileSize` type may also fail, where failure is indicated by an invalid value. A secondary check operation must be employed to determine whether the returned value is valid or not.

Finally, for functions that directly return the name of components as a string, the language in use defines the semantics of invalid responses. In C, where strings are direct pointers to memory, this is the special pointer value "NULL". Other languages have differing representations of an "undefined value" such as `None` (Python) or `nil` (Lua), but the semantics vary from language to language. In these cases, portable applications must check the return value using the string semantics for the language in use, before passing the value to another operation.

Declaration	boolean IsOK(in Result status);
Description	Determine if a `Result` value represents a successful response
Parameters	► `status`: A return value from a previous call
Return	If the status code represents a successful result, evaluates as TRUE. If the status code represents a failure, evaluates as FALSE.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	Converts a status code from any previous API call into a boolean value that can be used in conjunction with the programming language conditional statements. For efficiency reasons, this may be implemented as a macro or inline function in languages which support this concept.

Table 35: IsOK() Definition

Declaration	Result ValidateHandleID(in HandleID id);
Description	Determine if a `HandleID` value is valid

Parameters	► `id`: A return value from a previous call
Return	If the handle ID value is valid, returns the status value `OK`. If the handle ID is not valid, returns one of the defined error constants.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	This must be used to check the result of any function returning a `HandleID` value. The result of this function should be passed to `IsOK()` for use in any conditional test.

Table 36: ValidateHandleID() Definition

Declaration	Result ValidateSize(in FileSize size);
Description	Determine if a `FileSize` value is valid
Parameters	► `size`: A return value from a previous call
Return	If the size value is valid, returns the status value `OK`. If the size is not valid, returns one of the defined error constants.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	This must be used to check the result of any function returning a `FileSize` value. The result of this function should be passed to `IsOK()` for use in any conditional test.

Table 37: ValidateSize() Definition

Declaration	HandleID GetErrorQueue(in Result status);
Description	Obtain the error queue associated with the given status value
Parameters	► `status`: An error status code from a previous call
Return	Returns a handle ID value identifying the queue to which any associated log message should be written.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	This call is intended for use in conjunction with the `Log()` function for preserving error-related context information. The platform may direct different types of errors to different log queues to aid with diagnostics. For any given error response, this locates the proper queue for logging of any related information. In general, this should only be used for error status codes (i.e. those for which `IsOK()` returns FALSE). However, in all cases, the return value from this function must be passable directly to the `Log()` routine, without further validation, for any status code.

Table 38: GetErrorQueue() Definition

10.6.1.2 Name to Handle ID Mappings

All components operating within an environment must have an associated name and handle ID value. The name is more user-friendly, and as such is generally more useful for user interaction, whereas the numeric ID value is generally simpler and more efficient for software use. The functions described in this section provide a means to convert between these two forms of identification.

Declaration	`string GetHandleName(` `  in HandleID fromID,` `  in HandleID toID` `);`
Description	Obtain the handle name associated with the given handle ID
Parameters	► `fromID`: The handle ID of the current component making the request. ► `toID`: The handle ID of the component for which the name is to be obtained
Return	On success, returns a string representing the handle name. On error, returns an undefined or invalid value.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	In the event that the infrastructure cannot obtain the handle name, the infrastructure shall return an invalid string value for the respective language mapping. In C/C++ implementations an invalid string is represented as a NULL pointer.

Table 39: GetHandleName() Definition

Declaration	`HandleID HandleRequest(` `  in HandleID fromID,` `  in string toName` `);`
Description	Obtain the handle ID associated with the given handle name
Parameters	► `fromID`: The handle ID of the current component making the request. ► `toName`: The handle name of the component for which the ID should be obtained
Return	On success, returns a Handle ID value identifying the component. On error, an invalid handle ID value is returned. The returned value should always be validated by the caller using the `ValidateHandleID()` API call to determine success or failure.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	

Table 40: HandleRequest() Definition

10.6.2 Application Control API

The operations in this section are used for controlling applications or devices from other components in the system. Each operation generally corresponds to a corresponding operation in the application control interface documented in section 10.4.

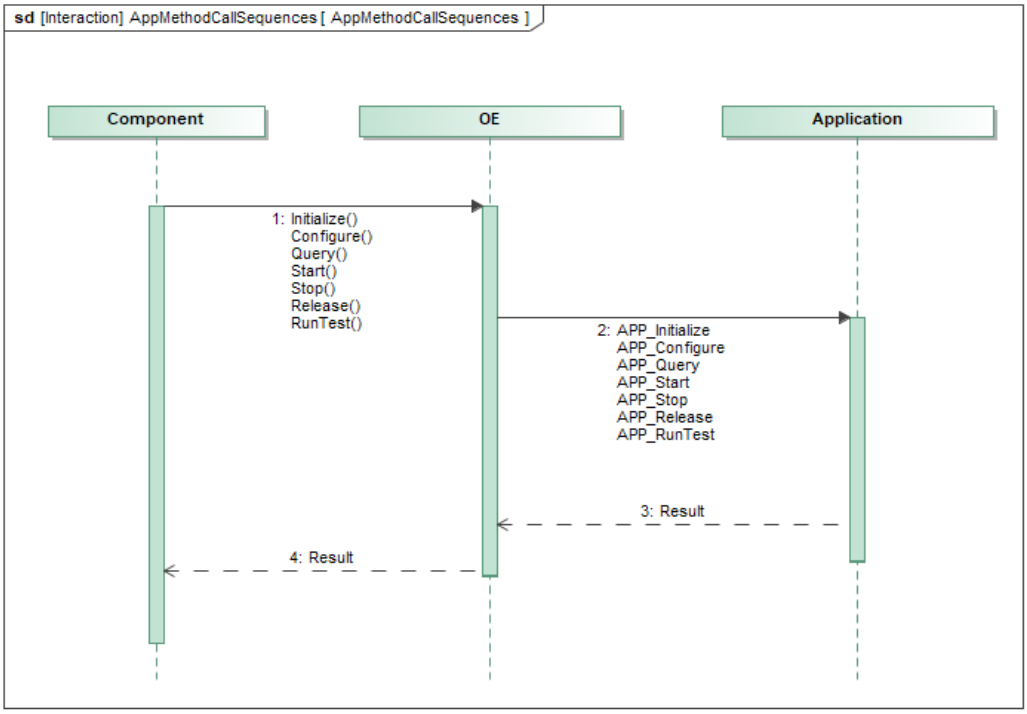

Figure 15: Sequence Diagram for Application Control

Figure 15 illustrates the general pattern of operations between the infrastructure API calls and the corresponding interface in the target application. The left side is the request originator, or the "from" entity in terms of the API descriptions, and is identified as handle 1. The right side is the request target, or the "to" entity in terms of the API descriptions, and is identified as handle 2. The originator uses the API calls described in this section, which in turn trigger the infrastructure to invoke the corresponding call on the target side. Upon completion, the return value follows the inverse path, through the infrastructure, and back to the originating component.

10.6.2.1 Setup and Teardown

The following API calls support the dynamic creation and deletion of components within the environment. See the corresponding application interface description in section 10.5.2.1 for more information.

Declaration	HandleID InstantiateApp(in HandleID fromID, in string handleName, in string configuration);
Description	Instantiate an application or service.
Parameters	► `fromID`: The handle ID of the current component making the request. ► `handleName`: The name of the new component to be instantiated. ► `configuration`: Configuration data to be associated with the new instance. If NULL or undefined, the OE should use defaults if appropriate/possible.

Return	On success, returns a Handle ID value identifying the newly-created instance. On error, an invalid handle ID value is returned. The returned handle value should always be validated by the caller using the `ValidateHandleID()` API call to determine success or failure.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	The handle name specified for the application, service, or device is to be unique within the scope of the current STI environment. The OE may also impose additional operations to be performed during instantiation, such as the loading of dynamic link libraries or shared objects, as documented by the platform provider. It is up to the OE to determine whether any additional resources are to be loaded to accomplish the instantiation. The configuration parameter also must be defined by the platform provider. This is a free-form string, and intended as a generic means to pass additional instructions to the infrastructure as part of the instantiation process. This string may directly contain a set of encoded configuration data (e.g. XML) , or it may refer to a filename on the system storage device containing additional information about the instance.

Table 41: InstantiateApp() Definition

The interaction between the originating component, the operating environment, and the target application for an InstantiateApp call is illustrated in Figure 16.

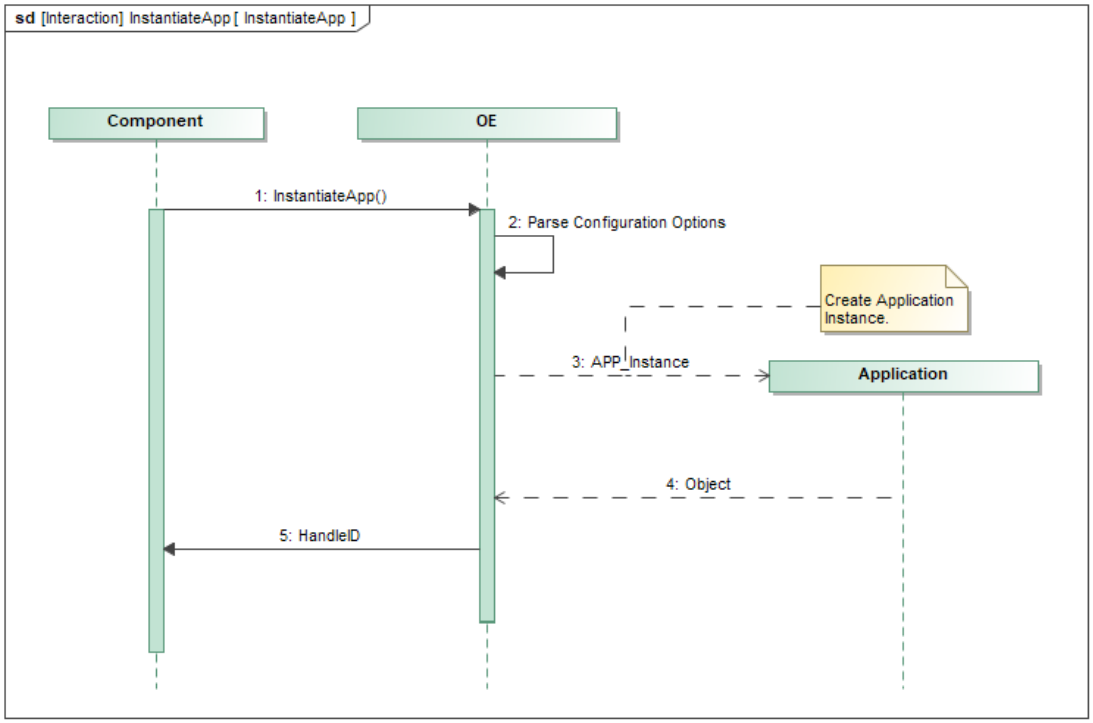

Figure 16: Sequence Diagram for InstantiateApp

Declaration	Result AbortApp(in HandleID fromID, in HandleID toID);
Description	Abort an application or service.
Parameters	► `fromID`: The handle ID of the current component making the request. ► `toID`: The handle ID of the target component that should respond to the request
Return	Status code which the caller should validate using `IsOK()` On error, returns one of the defined `Result` error constants. On success, returns `OK`.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	The target component will be removed from the environment, and any system resources associated with it should be released.

Table 42: AbortApp() Definition

The interaction between the originating component, the operating environment, and the target application for an AbortApp call is illustrated in Figure 17.

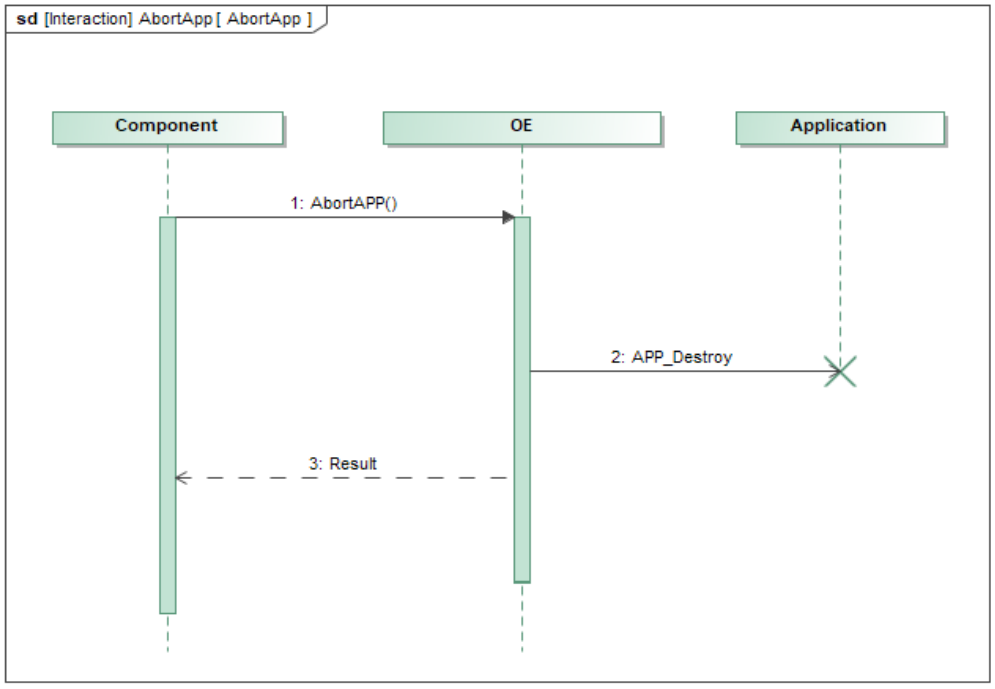

Figure 17: Sequence Diagram for AbortApp

10.6.2.2 Life Cycle Control

The following API calls correspond to the LifeCycle interface on the target component. See the corresponding application interface description in section 10.5.2.2 for more information.

Declaration	Result Initialize(in HandleID fromID, in HandleID toID);
Description	Initialize the target component.
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the component that should respond to the request
Return	On error, returns one of the defined error constants. On success, returns `OK`.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	This sets the component to a known initial state. The specific definition of this state is application-defined. This triggers the `APP_Initialize()` operation on the target interface.

Table 43: Initialize() Definition

Declaration	Result Release(in HandleID fromID, in HandleID toID);
Description	Releases any system resources held by the application or component
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the component that should respond to the request
Return	On error, returns one of the defined error constants. On success, returns `OK`.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	This triggers the `APP_Release()` operation on the target interface.

Table 44: Release() Definition

10.6.2.3 Property Set Control

The following API calls correspond to the PropertySet interface on the target component. See the corresponding application interface description in section 10.5.2.3 for more information.

Declaration	Result Configure(in HandleID fromID, in HandleID toID, in PropertyName propName, in PropertyValue propValue);

Description	Configures or sets a single property in the target component
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the component that should respond to the request ► `propName`: The name or identifier of the property to set ► `propValue`: A buffer containing the value to set the property to
Return	On error, returns one of the defined error constants. On success, returns `OK`.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	The caller manages the memory associated with the value buffer. This triggers the `APP_Configure()` operation in the target interface.

Table 45: Configure() Definition

Declaration	Result Query(in HandleID fromID, in HandleID toID, in PropertyName propName, out PropertyValue propValue);
Description	Obtains or gets a single property from the target component
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the component that should respond to the request ► `propName`: The name or identifier of the property to get ► `propValue`: A buffer into which the current value should be stored
Return	On error, returns one of the defined error constants. On success, returns `OK`.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	The caller manages the memory associated with the value buffer. This triggers the `APP_Query()` operation in the target interface.

Table 46: Query() Definition

10.6.2.4 Test Control

The following API calls correspond to the TestableObject interface on the target component. See the corresponding application interface description in section 10.5.2.4 for more information.

Declaration	Result RunTest(in HandleID fromID, in HandleID toID, in TestID testID);

Description	Obtain the handle ID associated with the given handle name
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the component that should respond to the request ► `testID`: The ID of the test to be performed
Return	On error, returns one of the defined error constants. On success, returns `OK`.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	The specific values and meaning of the `testID` parameter are application-specific. This triggers the `APP_RunTest()` operation in the target interface.

Table 47: RunTest() Definition

10.6.2.5 Operational Control

The following API calls correspond to the ControllableComponent interface on the target component. See the corresponding application interface description in section 10.5.2.5 for more information.

Declaration	Result Start(in HandleID fromID, in HandleID toID);
Description	Begin normal application or device processing
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the component that should respond to the request
Return	On error, returns one of the defined error constants. On success, returns `OK`.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	This triggers the `APP_Start()` operation in the target interface.

Table 48: Start() Definition

Declaration	Result Initialize(in HandleID fromID, in HandleID toID);
Description	End normal application or device processing
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the component that should respond to the request
Return	On error, returns one of the defined error constants. On success, returns `OK`.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device

Notes	This triggers the `APP_Stop()` operation in the target interface.

Table 49: Stop() Definition

10.6.3 Device Control API

The following API calls allow applications to interact with STI devices. These operations provide a means to establish a path of communication to the device, and correlate to the DeviceControl interface on the target component. See the corresponding application interface description in section 10.5.3 for more information.

Declaration	Result DeviceOpen(in HandleID fromID, in HandleID toID);
Description	Open the device
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the component that should respond to the request
Return	On error, returns one of the defined error constants. On success, returns `OK`.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	This triggers the `DEV_Open()` operation in the target interface. This must be the first call issued before invoking any other device control operations.

Table 50: DeviceOpen() Definition

Declaration	Result DeviceLoad(in HandleID fromID, in HandleID toID, in string fileName);
Description	Load an application, hardware design, or configuration file into the device
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the component that should respond to the request ► `fileName`: The name of the file to load
Return	On error, returns one of the defined error constants. On success, returns `OK`.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	This triggers the `DEV_Load()` operation in the target interface.

Table 51: DeviceLoad() Definition

Declaration	Result DeviceReset(in HandleID fromID, in HandleID toID);

Description	Resets the device into a known state
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the component that should respond to the request
Return	On error, returns one of the defined error constants. On success, returns `OK`.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	The specific state after reset is device-defined. This triggers the `DEV_Reset()` operation in the target interface.

Table 52: DeviceReset() Definition

Declaration	Result DeviceFlush(in HandleID fromID, in HandleID toID);
Description	Clears any pending input/output data buffers in the device
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the component that should respond to the request
Return	On error, returns one of the defined error constants. On success, returns `OK`.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	This triggers the `DEV_Flush()` operation in the target interface.

Table 53: DeviceFlush() Definition

Declaration	Result DeviceUnload(in HandleID fromID, in HandleID toID);
Description	Unload any previously-loaded application, hardware design image, or configuration file
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the component that should respond to the request
Return	On error, returns one of the defined error constants. On success, returns `OK`.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	This triggers the `DEV_Unload()` operation in the target interface.

Table 54: DeviceUnload() Definition

Declaration	Result DeviceClose(in HandleID fromID, in HandleID toID);
Description	Closes the device
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the component that should respond to the request
Return	On error, returns one of the defined error constants. On success, returns `OK`.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	This triggers the `DEV_Close()` operation in the target interface. The device must not be used by the application after this call unless opened again.

Table 55: DeviceClose() Definition

10.6.4 Data Transfer API

The following API calls correspond to the data transfer (Source, Sink, RandomAccess) interfaces on the target component. These functions are also used to transfer data to or from files or message queues.

10.6.4.1 Data Source

The data source operation described in this section is applicable to any application or device that implements the "Source" interface. See the corresponding application interface description in section 10.5.4.1 for more information.

Declaration	Result Read(in HandleID fromID, in HandleID toID, out Message buffer);
Description	Read or "pull" arbitrary data from another application or device
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the component that should respond to the request ► `buffer`: A buffer to hold the transferred data
Return	On error, returns one of the defined error constants. On success, returns a status value indicating the actual number of records or bytes of data that was transferred into the supplied buffer. The caller must check the return status using `IsOK()`
Implemented By	Infrastructure
Invoked By	Application, Service, or Device

Notes	The storage for the buffer must be managed by the caller. The target application defines the specific specific binary format for the data. In languages with direct memory access (e.g. C/C++), the buffer is an arbitrary memory location with the units specified in bytes. In other languages, the units should reflect logical records, such as a number of characters, samples or objects. The infrastructure makes no assumptions about the format of the message data, nor the presence or expectation of a terminating entity, such as a NUL character as required for C-style strings. If a terminating character is required, the caller must ensure that sufficient space is available in the buffer to store the termination character.

Table 56: Read() Definition

10.6.4.2 Data Sink

The data sink operation described in this section is applicable to any application or device that implements the "Sink" interface. See the corresponding application interface description in section 10.5.4.2 for more information.

Declaration	Result Write(in HandleID fromID, in HandleID toID, in Message buffer);
Description	Write or "push" arbitrary data to another application or device
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the component that should respond to the request ► `buffer`: A buffer containing the data to be transferred
Return	On error, returns one of the defined error constants. On success, returns a status value indicating the actual number of records or bytes of data that was transferred from the supplied buffer. The caller must check the return status using `IsOK()`
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	The storage for the buffer must be managed by the caller. The target application defines the specific specific binary format for the data. In languages with direct memory access (e.g. C/C++), the buffer is an arbitrary memory location with the units specified in bytes. In other languages, the units should reflect logical records, such as a number of characters, samples or objects. The infrastructure makes no assumptions about the format of the message data, nor the presence or expectation of a terminating entity, such as a NUL character as required for C-style strings. If a terminating character is required, the caller must ensure that sufficient space is available in the buffer to store the termination character.

Table 57: Write() Definition

10.6.4.3 Random Access

These operations provide a means to directly access specific locations within a device or file, and correlate to the RandomAccess interface on the target component. See the corresponding application interface description in section 10.5.4.3 for more information.

Declaration	`Result AddressRead(` `  in HandleID fromID,` `  in HandleID toID,` `  in Offset offset,` `  out Message buffer` `);`
Description	Read data from a specific offset or address within a device or file
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the component that should respond to the request ► `offset`: The location to read data from ► `buffer`: A buffer to hold the transferred data
Return	On error, returns one of the defined error constants. On success, returns a status value indicating the actual number of records or bytes of data that was transferred into the supplied buffer. The caller must check the return status using `IsOK()`
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	The storage for the buffer must be managed by the caller. The target application defines the specific specific binary format for the data. In languages with direct memory access (e.g. C/C++), the buffer is an arbitrary memory location with the units specified in bytes. In other languages, the units should reflect logical records, such as a number of characters, samples or objects. The infrastructure makes no assumptions about the format of the message data, nor the presence or expectation of a terminating entity, such as a NUL character as required for C-style strings. If a terminating character is required, the caller must ensure that sufficient space is available in the buffer to store the termination character.

Table 58: AddressRead() Definition

Declaration	`Result AddressWrite(` `  in HandleID fromID,` `  in HandleID toID,` `  in Message buffer` `);`
Description	Write data to a specific offset or address within a device or file
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the component that should respond to the request ► `offset`: The location to write data to ► `buffer`: A buffer containing the data to be transferred

Return	On error, returns one of the defined error constants. On success, returns a status value indicating the actual number of records or bytes of data that was transferred from the supplied buffer. The caller must check the return status using `IsOK()`
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	The storage for the buffer must be managed by the caller. The target application defines the specific specific binary format for the data. In languages with direct memory access (e.g. C/C++), the buffer is an arbitrary memory location with the units specified in bytes. In other languages, the units should reflect logical records, such as a number of characters, samples or objects. The infrastructure makes no assumptions about the format of the message data, nor the presence or expectation of a terminating entity, such as a NUL character as required for C-style strings. If a terminating character is required, the caller must ensure that sufficient space is available in the buffer to store the termination character.

Table 59: AddressWrite() Definition

10.6.5 Log API

The Log API provides a means to record contextual information regarding errors or other conditions present in applications. The log data is maintained by the infrastructure, and may be sent to the operating system log facility if one exists. The platform provider must indicate the specific manner with which log data may be retrieved and examined by the operator, such as a file location or system log viewer.

Declaration	```Result Log(in HandleID fromID, in HandleID toID, in string logMsg);```
Description	Sends an information message to the specified log facility
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the log queue to which the message should be sent ► `logMsg`: A message to send to the log facility
Return	On error, returns one of the defined error constants. On success, returns `OK`.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	When logging context information related to errors, the `GetErrorQueue()` function may be used to determine the proper value to use for the `toID` parameter. In other cases, the predefined error queue constants may be used, as listed in Table 8, HandleID Constants. Behavior is not specified if the `toID` parameter is does not refer to a component capable of accepting log messages (i.e. one of the defined log facilities).

Table 60: Log() Definition

10.6.6 File API

The API calls described in this section allow an STI application or device to open, close, and manipulate files, in an abstract sense, within the operating environment. Note that the file system implemented by the STI infrastructure may or may not correspond to an actual file system in the underlying operating system. The file system may be virtualized, and the presence of these API functions does not imply a requirement that the operating system must actually implement a conventional file system.

The basic requirements of the file system abstraction are:

- All applications and devices access the same file system (real or virtual). A file created by one application or device, may be subsequently opened by a different application or device, using the same file name.
- The files are persistent for at least the lifetime of the current infrastructure. A virtual file system backed in RAM or other volatile storage may be cleared when the infrastructure is restarted or the host system is rebooted. File systems should have longer persistence (i.e. across reboots) when backed by a non-volatile storage device.

The platform developer must indicate the level of persistence offered by the file system abstraction.

The methods defined in this section pertain to file system manipulation and provide a means to open or close file handles. For actual data transfer operations, file handles shall respond to the data transfer methods as defined in section 10.6.4, Data Transfer API.

10.6.6.1 File Handle Operations

Like other components, files in STI operating environment are identified using a handle ID, and as such file handles share many of the same semantics with other applications and devices. The difference lies in that file handles are obtained using the specific API methods described here, rather than the previously described methods used for applications or devices. The operations in this section manipulate file handles within the environment.

Declaration	HandleID FileOpen(in HandleID fromID, in string fileName, in Access fileAccess);
Description	Opens a file within the infrastructure file system
Parameters	► `fromID`: The handle ID of the current component making the request ► `fileName`: The name of the file to be opened ► `fileAccess`: Whether the file is to be opened for reading, writing, appending, or both (reading and writing).
Return	On success, returns a Handle ID value identifying the open file. On error, an invalid handle ID value is returned. The returned handle value should always be validated by the caller using the `ValidateHandleID()` API call to determine success or failure.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	After successfully opening a file, data transfer can be performed using the read and write functions described in section 10.6.4. For the file access types, see Table 6, Access Constants.

Table 61: FileOpen() Definition

Declaration	Result FileClose(in HandleID fromID, in HandleID toID);
Description	Closes a file handle
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the file that should be closed
Return	On success, returns `OK`. On error, returns one of the defined error constants.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	The `toID` parameter should reflect a file handle that was previously obtained using `FileOpen()`. Behavior is undefined if this function is called with other types of handle IDs.

Table 62: FileClose() Definition

10.6.6.2 File System Operations

The operations in this section manipulate or query the file system itself, rather than on file handles within the file system.

Declaration	FileSize FileGetSize(in HandleID fromID, in string fileName);
Description	Get the size of the specified file
Parameters	► `fromID`: The handle ID of the current component making the request ► `fileName`: The name of the file to obtain the size of
Return	On success, returns the size of the file. On error, returns an invalid size.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	The return value should be validated by the caller using the `ValidateSize()` operation as described in section 10.6.1.1.

Table 63: FileGetSize() Definition

Declaration	Result FileRemove(in HandleID fromID, in string fileName);
Description	Removes a specified file from the system
Parameters	► `fromID`: The handle ID of the current component making the request ► `fileName`: The name of the file to remove

Return	On success, returns `OK`. On error, returns one of the defined error constants.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	Behavior of this function is implementation-defined if the specified file is currently open within the infrastructure. Some systems may support this by "unlinking" the file name but deferring the actual removal (and recovery of space) until the file is closed. On other systems, the function may return an error if the file is currently open. Portable applications should ensure that a file has been closed prior to removal.

Table 64: FileRemove() Definition

Declaration	Result FileRename(in HandleID fromID, in string oldName, in string newName);
Description	Renames a specified file in the file system
Parameters	► `fromID`: The handle ID of the current component making the request ► `oldName`: The existing/current name of the file ► `newName`: The desired name of the file
Return	On success, returns `OK`. On error, returns one of the defined error constants.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	Behavior of this function is implementation-defined if the specified file is currently open within the infrastructure. Some systems may support renaming an open file, but on other systems the function may return an error. Portable applications should ensure that a file has been closed prior to rename.

Table 65: FileRename() Definition

Declaration	FileSize FileGetFreeSpace(in HandleID fromID, in string fileSystem);
Description	Get the total free space available for file storage on the indicated file system
Parameters	► `fromID`: The handle ID of the current component making the request ► `fileSystem`: Identifies the file system to check
Return	On success, returns the amount of free space. On error, returns an invalid size.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device

Notes	The specific format and options for the fileSystem parameter must be defined by the platform provider. An invalid (undefined/NULL) or empty string value should always be interpreted to refer to the "default" or root storage device, if available. A non-empty string may refer to a physical device name, drive identifier, or a mount point, depending on the system.

Table 66: FileGetFreeSpace() Definition

10.6.7 Messaging API

The STI applications use the Messaging API to establish facilities to send messages between components using a common handle ID. The ability for applications, services, devices, or files to communicate with other STI applications, services, devices, or files is crucial for the separation of radio functionality among independent components. When using the message passing API, the final destination of a message is not necessarily known to the producer of the message.

For example, the receive and transmit telecommunication functionalities can be separated between two applications. Another example is when commands or log messages come from several independent sources and have to be merged appropriately. Some examples of independent components that may need to interact with others could be for navigation, GPS, file upload, file download, and computations.

There are two models for passing messages: queues (first in, first out, or FIFO) and publish/subscribe (PubSub). In a queue, messages are written to a queue by one entity and read from the queue by another entity. In a PubSub model, messages written to the message passing facility by one application are delivered to all subscribers of that publisher.

To write to or read from a FIFO queue, the `Read()` and `Write()` operations are used, respectively, as described in section 10.6.4, Data Transfer API. In this model , the originating entity pushes data to the queue, where it is temporarily stored. The receiving entity pulls data from the queue at a later time, at which time it is removed from the queue. By definition, FIFO queues only provide sequential data, they do not support random access.

In the publish/subscribe (PubSub) messaging model, the data is pushed to all subscribers using a one-to-many distribution. All applications subscribing to receive data using this model must implement the "Sink" interface as described in section 10.5.4.2. Note that any handle ID capable of acting as a data sink may be subscribed to a PubSub message distribution, including files and FIFO queues. By registering an open file handle ID, one can effectively create a "tap" to log all published data. Likewise, by registering a FIFO queue, the two messaging models may be combined, allowing broadcast data to be buffered and then "pulled" by the receiver at a later time.

10.6.7.1 FIFO Queue Model

The API calls described in this section implement the "first-in, first-out" (FIFO) queue model.

Declaration	HandleID MessageQueueCreate(in HandleID fromID, in string queueName, in QueueMaxMessages nmax, in BufferSize nb);
Description	Create a FIFO message queue
Parameters	► `fromID`: The handle ID of the current component making the request ► `queueName`: The name of the queue to create ► `nmax`: The maximum number of messages (depth) of the FIFO queue ► `nb`: The maximum size of each entry in the queue

Return	On success, returns a Handle ID value identifying the FIFO queue. On error, an invalid handle ID value is returned. The returned handle value should always be validated by the caller using the `ValidateHandleID()` API call to determine success or failure.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	The queue name must be unique in within the current environment. Once a queue depth reaches its maximum (`nmax`), applications will be unable to write new data into the queue. Data does not "expire" from a FIFO queue; any data successfully written to the input side of a queue is removed only by reading the data from the output side of the queue, or by deleting the entire queue. If the `nb` parameter is omitted or specified as 0, the interpretation is implementation-defined. Specifically, this may be used for languages that employ automatic memory management and do not expose the size of objects in memory to applications.

Table 67: MessageQueueCreate() Definition

Declaration	Result MessageQueueDelete(in HandleID fromID, in HandleID toID);
Description	Delete a FIFO queue
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the queue that should be deleted
Return	On success, returns `OK`. On error, returns one of the defined error constants.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	Any written but unread data messages in the queue are discarded.

Table 68: MessageQueueDelete() Definition

10.6.7.2 Publish/Subscribe Model

The API calls described in this section implement the publish/subscribe messaging model.

Declaration	HandleID PubSubCreate(in HandleID fromID, in string pubSubName);
Description	Create a PubSub entity
Parameters	► `fromID`: The handle ID of the current component making the request ► `pubSubName`: The name of the PubSub entity to be created

Return	On success, returns a Handle ID value identifying the PubSub entity. On error, an invalid handle ID value is returned. The returned handle value should always be validated by the caller using the `ValidateHandleID()` API call to determine success or failure.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	The name must be unique in within the current environment. Unlike FIFO queues, PubSub entities do not store messages; any messages pushed to a PubSub entity are immediately distributed to all currently registered subscribers at the time the message is pushed.

Table 69: PubSubCreate() Definition

Declaration	Result PubSubDelete(in HandleID fromID, in HandleID toID);
Description	Delete a PubSub entity
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the PubSub entity to be deleted
Return	On success, returns `OK`. On error, returns one of the defined error constants.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	Any registered subscribers will be automatically unregistered upon deletion.

Table 70: PubSubDelete() Definition

Declaration	Result Register(in HandleID fromID, in HandleID toID, in HandleID recipientID);
Description	Add a handle to the recipient list of the PubSub entity
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the PubSub entity ► `recipientID`: The handle ID of another application, device, file, or queue that should receive all messages written to the PubSub entity
Return	On success, returns `OK`. On error, returns one of the defined error constants.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device

Notes	A single recipient cannot be registered multiple times. If a recipient is already registered within the PubSub entity, this function returns a success code without making any change.

Table 71: Register() Definition

Declaration	Result Unregister(in HandleID fromID, in HandleID toID, in HandleID recipientID);
Description	Remove a handle from the recipient list of the PubSub entity
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the PubSub entity ► `recipientID`: The handle ID of the other application, device, file, or queue that should no longer receive messages written to the PubSub entity
Return	On success, returns `OK`. On error, returns one of the defined error constants.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	

Table 72: Unregister() Definition

10.6.8 Time API

The STI Infrastructure Time methods are used to access the hardware and software timers. Methods are also defined to support synchronization of oscillators or other timing sources to a reference signal.

Many time operations utilize an object type called `TimeWarp`, which represents an abstract time interval. Nominally, the `TimeWarp` object is expected to be some form of timer tick counter, with the specific resolution/units and epoch being implementation-defined. A `TimeWarp` object may represent time in standardized units, such as milliseconds or microseconds, or it may be based on the CPU clock or timer interrupt frequency. Although some API methods are defined to a nanosecond time resolution, that does not imply that the actual timer resolution is nanoseconds or that the underlying `TimeWarp` object contains its data in nanoseconds.

The following must be true:

- The resolution or units of `TimeWarp` objects is a fixed constant defined by the infrastructure, and does not change for the lifetime of the infrastructure. For instance, if a clock is sampled at times A, B, and C, and the time interval between B-A and C-B is equal, then the corresponding difference between the successive `TimeWarp` values must also be equal.
- All clock devices within an infrastructure must share the same definition of `TimeWarp`, with respect to range and resolution, even if the clock devices do not share the same epoch.
- `TimeWarp` objects must be capable of differentiating between positive intervals (time in the future) and negative intervals (time in the past).

Depending on the application, time intervals may be of a long duration (years or decades) and/or high resolution (microseconds or nanoseconds). To support a wide range of time while also maintaining a high resolution, it may

not be possible to represent a `TimeWarp` value as a single value on a particular CPU. For instance, if a timer has a resolution of 1 microsecond and is represented using a 32-bit signed integer, which is the largest native integer type on some microcontrollers, then the measurable time intervals would be limited to only (2^{31}-1) microseconds, or approximately 35.7 minutes. Therefore, `TimeWarp` may be implemented as a structure or other extended-range numeric type in order to achieve the necessary range and resolution requirements.

10.6.8.1 Time Conversion and Arithmetic

The `TimeWarp` object is defined by the infrastructure as a value that represents a specific interval in time. The specific structure of this object is implementation-defined. For example, the underlying `TimeWarp` object could count ticks from some epoch, such as the infrastructure boot time, and then `GetSeconds` and `GetNanoseconds` compute the seconds and nanoseconds, respectively, based on the tick rate.

The following methods provide a means to work with `TimeWarp` objects, and to convert or translate these objects into other representations. As the specific implementation of the `TimeWarp` object may vary, applications cannot assume that normal arithmetic or logical operations are possible (i.e. addition or subtraction, equality testing, etc.). Therefore, the infrastructure must define these operations.

In order to make these operations as efficient as possible, all operations defined in this section may be implemented as macros or inline functions on platforms that offer this feature. There is also no need for error checking and no possibility of failure on these operations, as any input value is valid.

Declaration	Nanoseconds GetNanoseconds(in TimeWarp twObj);
Description	Get the number of nanoseconds (fractional quantity) from the `TimeWarp` object.
Parameters	► `twObj`: The value from which the nanoseconds portion of the time is extracted
Return	Returns the number of nanoseconds
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	The nanoseconds value is always non-negative, and reflects the difference between the actual interval time and the number of whole seconds in the interval as reported by `GetSeconds()`

Table 73: GetNanoseconds() Definition

Declaration	Seconds GetSeconds(in TimeWarp twObj);
Description	Get the number of seconds (whole number quantity) from the `TimeWarp` object.
Parameters	► `twObj`: The value from which the seconds portion of the time is extracted
Return	Returns the number of seconds
Implemented By	Infrastructure
Invoked By	Application, Service, or Device

Notes	The seconds value may be negative, which indicates an interval back in time. For fractional intervals, the seconds value reflects the largest integral value not greater than the interval length in seconds, similar to the POSIX `floor()` operation applied to a floating-point value. For example, given a `TimeWarp` interval corresponding to -1.1s, the `GetSeconds()` function shall return -2, and the `GetNanoseconds()` function shall return 900,000,000.

Table 74: GetSeconds() Definition

Declaration	TimeWarp GetTimeWarp(in Seconds isec, in Nanoseconds nsec);
Description	Get the `TimeWarp` object value corresponding to the seconds and nanoseconds
Parameters	► `isec`: The number of seconds in the time interval (whole number portion) ► `nsec`: The number of nanoseconds in the time interval (fractional portion)
Return	Returns the corresponding time value as a `TimeWarp` value
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	The `nsec` parameter value should be between 0 and 999,999,999, inclusive. If the `nsec` value is not within this range, the infrastructure should adjust the `isec` and `nsec` values by decrementing/incrementing `nsec` by 1,000,000,000 and incrementing/decrementing `isec` by 1, respectively, until the `nsec` value is within this range.

Table 75: GetTimeWarp() Definition

Declaration	TimeWarp TimeAdd(in TimeWarp t1, in TimeWarp t2);
Description	Compute the sum of two `TimeWarp` values
Parameters	► `t1, t2`: Any previously-obtained time values
Return	The sum `(t1 + t2)` expressed as a `TimeWarp` value
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	

Table 76: TimeAdd() Definition

Declaration	TimeWarp TimeSubtract(in TimeWarp t1, in TimeWarp t2);

Description	Compute the difference between two `TimeWarp` values
Parameters	► `t1, t2`: Any previously-obtained time values
Return	The difference `(t1 - t2)` expressed as a `TimeWarp` value
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	This operation may be implemented as a macro or inline function for efficiency, on languages that offer this feature. This operation may be used by software to compute the elapsed time between two successive calls to `GetTime()`. The result can be converted back to engineering units via the `GetSeconds()` and `GetNanoseconds()` operations

Table 77: TimeSubtract() Definition

10.6.8.2 Basic Clock Get/Set Operations

The API calls described in this section implement the basic clock operations such as getting the time, setting the time, or suspending/delaying operation until the clock reaches a specific value.

Declaration	Result GetTime(in HandleID fromID, in HandleID toID, out TimeWarp currentTime);
Description	Obtains the current value of the clock
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the clock device that should respond to the request ► `currentTime`: A buffer to store the current time, as an interval since the epoch
Return	On success, returns `OK`. On error, returns one of the defined error constants.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	The output value returned represents a direct measurement of elapsed time since its respective epoch according to the clock's time scale, and is not adjusted for nor dependent upon any locale-specific time representations (i.e. time zone, daylight savings time, etc.) or effects of relativity.

Table 78: GetTime() Definition

Declaration	Result SetTime(in HandleID fromID, in HandleID toID, in TimeWarp deltaTime);
Description	Sets the current value of the clock

Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the clock device that should respond to the request ► `deltaTime`: The step size, relative to the current clock value
Return	On success, returns `OK`. On error, returns one of the defined error constants.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	This function will "step" the base clock. Since the offset is applied against the base clock measurement, it affects **all** calendar representations of the clock accordingly. It may be used to synchronize a clock based on information obtained after start up. Not all clock devices are required to support this operation. If a clock device is read-only and not settable from an application, this function should return `UNIMPLEMENTED`. Note that this is not intended for implementing the concept of a "time zone" or "local time" (i.e. the time as commonly expressed in a given geopoliticial region). If the platform implements the concept of local time, then the specific local time offset or conversion rules should be configured using the PropertySet API as described in section 10.6.2.3. The specific property name and value format for time zone configuration is platform-defined. On some systems, it may be directly configured as a number (i.e. minutes ahead of GMT) or it may be configured as a string reflecting a predefined rule (i.e. "US/Eastern") if the system is capable of automatic daylight savings time adjustments.

Table 79: SetTime() Definition

Declaration	```Result GetCalendarTime(in HandleID fromID, in HandleID toID, in TimeWarp referenceTime, in CalendarKind calendarKind, out CalendarTime calendarTime);```
Description	Convert the base clock time value to a defined calendar representation
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the clock device that should respond to the request ► `referenceTime`: The time to convert, expressed as an interval since the clock epoch ► `calendarKind`: The calendar system to convert the reference time to ► `calendarTime`: A buffer to store the calendar representation of the reference time
Return	On success, returns `OK`. On error, returns one of the defined error constants.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device

Notes	This call is used by applications to convert an abstract `TimeWarp` value (of which the definition is platform-specific) into a value in one of the defined calendar systems, such that portable applications can interpret it in a consistent manner. If the system or clock does not support the requested `calendarKind`, the implementation should return `UNIMPLEMENTED`. If the `referenceTime` is zero, such as the result of a call to `GetTimeWarp(0,0)` then this shall return the respective calendar representation of the clock epoch.

Table 80: GetCalendarTime() Definition

Several predefined constants for the `CalendarKind` type are specified in section 10.4.2. A compliant platform does not necessarily need to implement all the calendar types listed, and may implement additional types not listed as application-specific extensions. To support the various different time representations, several structures are provided by the infrastructure. The time representations are illustrated in Figure 18, Calendar Time Value Representations.

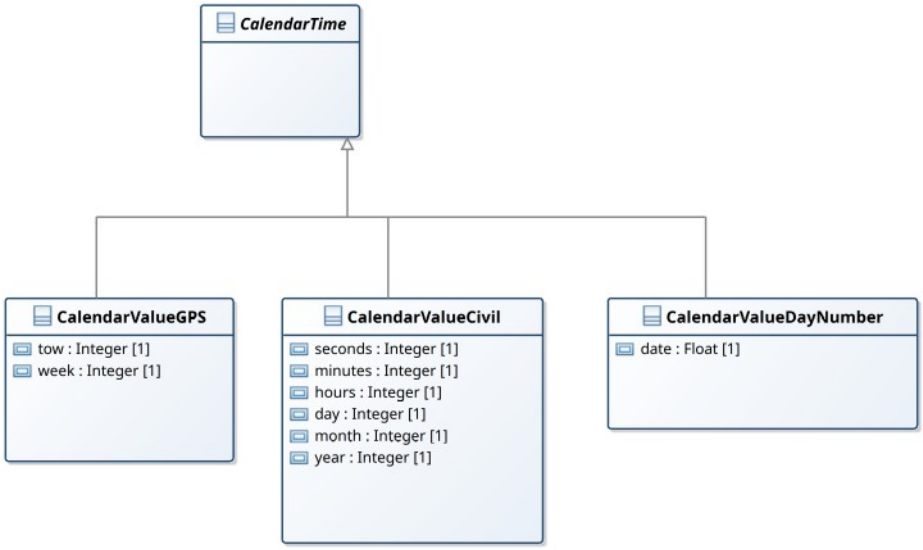

Figure 18: Calendar Time Value Representations

The `CalendarTime` type may be expressed as an IDL union of all possible time representations, as indicated below.

Declaration	struct CalendarValueCivil { long nanoseconds; octet seconds; octet minutes; octet hours; octet day; octet month; short year; };
Description	Definition of time representation type for the common era / Gregorian calendar

Member Details	► `nanoseconds`: The number of nanoseconds, range of [0-999999999] ► `seconds`: The seconds value, range of [0-60] ► `minutes`: The minutes value, range of [0-59] ► `hours`: The hours value, range of [0-23] ► `day`: The day number within the month, range of [0-30] ► `month`: The month number within the year, range of [0-11] ► `year`: The full year number, expressed as an integer (i.e. 2019)
Implemented By	Infrastructure
Notes	This format is applicable to UTC and, usually, the local time representations. For local time representations, the specific offset from UTC and daylight savings configuration should be configured or queried separately through the property set interface. The nanoseconds field is intended to support applications that require higher precision time values. This does not imply that the underlying clock has nanosecond precision. For clocks that do not support higher precision timing, this field should always be set as zero.

Table 81: CalendarValueCivil Structure Definition

Declaration	struct CalendarValueGPS { long tow; short week; };
Description	Definition of time representation expressed in weeks and seconds, similar to the style used in GPS navigation messages
Member Details	► `tow`: The time of week in milliseconds, range of [0-604799999] ► `week`: The number of weeks elapsed since the epoch
Implemented By	Infrastructure
Notes	This is not an exact representation of GPS time codes, but rather a method of expressing time in terms that facilitate easy conversion to/from actual GPS navigation code formats while also providing higher precision. Legacy GPS navigation signals express the week number as a 10 bit integer, which rolls over every 1024 weeks, with time of week expressed as a 19 bit integer with 1.5 second resolution. Other navigation signals have a different format, using 13 bit week number along with a 2-hour interval time of week and 18-second time of interval. This structure expresses the time of week value in units of milliseconds. Conversion from legacy GPS time of week values is accomplished via multiplication by 1500 (1.5 seconds), and conversion from 18-second time of interval codes is accomplished via multiplication by 18000. Likewise, a conversion to whole seconds can be achieved by dividing the `tow` by 1000, and the day of week can be determined by dividing by 86400000.

Table 82: CalendarValueGPS Structure Definition

Declaration	struct CalendarValueDayNumber { double date; };

Description	Definition of time representation expressed as a fractional day number
Member Details	► `date`: The day number expressed as a fractional / floating point value
Implemented By	Infrastructure
Notes	The whole number (integer portion) of the value expresses the number of Earth days since the epoch, and the fractional part expresses the time of day.

Table 83: CalendarValueDayNumber Structure Definition

Declaration	union CalendarTime switch(CalendarKind) { case MJD: CalendarValueDayNumber dayNumber; case GPS: CalendarValueWeekSeconds weekSeconds; case LOCAL: case TAI: case UTC: CalendarValueCivil civil; };
Description	Definition of `CalendarTime` type based on `CalendarKind` value
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	

Table 84: CalendarTime Union Definition

10.6.8.3 Clock Rate Adjustment and Drift Compensation

If clock devices require synchronization with external signals, a dedicated service should continuously monitor for drift and handle the adjustment as needed. Common synchronization sources include a "time at tone" signal from a ground station, a 1 pulse per second (PPS) input from a GPS receiver, or via the network time protocol (NTP). Differences between the synchronization source and the clock device can be compensated by either directly stepping the clock device using `SetTime()`, or, if the underlying device supports it, by low-level adjustment of the clock source tick rate such that the drift is gradually absorbed and corrected over time.

The `SetTime()` API sets the clock directly, and will step the timer forward or backward as indicated. However, a timer step may have undesirable consequences for some software, particularly control loops that rely on relative time differences between successive samples. This can sometimes be mitigated by making many small steps rather than one large step. However, even the smallest step still might cause unacceptable effects on a control loop that relies on precise relative timing measurements.

The adjustment functions are intended to address this by providing an alternative method to adjust for clock drift. In many clock device implementations, the underlying "tick" or reference signal is supplied using a hardware PLL/oscillator or clock divider of some type, driving a periodic timer tick interrupt to the CPU. Furthermore, if the source allows some level of control during operation, such as increasing or decreasing the oscillator rate by a certain ratio (e.g. parts per million) or by modifying the clock divider ratio by a small amount, then this can be used to provide for a more stable drift compensation method. By increasing or decreasing the underlying timekeeping tick rate, small differences between the clock device and the reference source can be compensated over time without ever "stepping" the clock.

Support for these adjustment routines is platform-dependent. If a platform does not support clock drift adjustment, an appropriate error code will be returned.

Declaration	Result SetTimeAdjust(in HandleID fromID, in HandleID toID, in TimeRate rateAdjustment);
Description	Adjusts the tick rate of the clock device
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the clock device that should respond to the request ► `rateAdjustment`: The amount of adjustment to apply
Return	On success, returns `OK`. On error, returns one of the defined error constants.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	The `rateAdjustment` parameter is a signed integer, where the value of 0 represents the nominal or free-run rate of the clock without any adjustment applied. If any adjustment had been previously applied, calling this function with a value of 0 will restore a clock to its default rate. A positive value will cause the clock frequency to increase from the nominal rate, and a negative value will cause the clock frequency to decrease from the nominal rate. The specific unit of rate increase/decrease is platform defined, although typically might reflect a number of parts per million or parts per billion depending on clock design. If the underlying device does not support rate adjustment, then this function will return the `UNIMPLEMENTED` status code. A typical use-case of this function would periodically compute the difference between the reference clock and the local clock device, which is then multiplied by a feedback ratio (proportional coefficient) to compute the adjustment value to pass into this function.

Table 85: SetTimeAdjust() Definition

Declaration	TimeRate GetTimeAdjust(in HandleID fromID, in HandleID toID);
Description	Obtain the current tick rate adjustment value of the clock device
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the clock device that should respond to the request
Return	Returns the current tick rate adjustment value
Implemented By	Infrastructure
Invoked By	Application, Service, or Device

Notes	A return value of 0 indicates the clock is operating at its nominal or free-run frequency. If the underlying device does not support rate adjustment, then this function always returns 0. A positive value indicates the clock frequency is above nominal, and a negative value indicates the clock frequency is below nominal. The specific units of the `TimeRate` value are platform defined, although typically might reflect a number of parts per million or parts per billion depending on clock design.

Table 86: GetTimeAdjust() Definition

Declaration	Result TimeSynch(in HandleID fromID, in HandleID toID, in HandleID referenceID, in TimeWarp stepMax);
Description	Synchronizes a clock device with another waveform or device in the system
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the clock device that should respond to the request ► `referenceID`: The handle ID of another device or waveform in the system that provides a synchronization source for the target clock device. ► `stepMax`: The maximum amount that the target clock should be modified.
Return	A status code that should be checked using `IsOK()`.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	This function is intended for use in systems where a local general-purpose clock/timer service may be selectively synchronized with other devices on the system. Support for this function is implementation-defined, and this function may return `UNIMPLEMENTED` if the clock device does not support synchronization with any other devices. The infrastructure provider must document set of devices or services suitable for use with the `referenceID` parameter. This reference device may be another infrastructure-provided clock/timer service, or it may be another form of timing reference, such as a software service implementing a protocol such as NTP or IEEE-1588, or a device capable of recovering timing signals from received bit streams. The `stepMax` parameter specifies the maximum amount that the target clock device may be modified in a single step change. The constant `TIME_INTERVAL_MAX` may be specified to indicate no limit to the step size, permitting the target device to be directly set to any value. If the synchronization is successful with a single call to `TimeSynch()`, such that no further action is required, the implementation shall return `OK`. If the synchronization is successful but requires multiple calls (e.g. due to constraints imposed by `stepMax`) the implementation shall return a positive integer value indicating the anticipated number of calls required. If synchronization is not possible under the given constraints the implementation shall return a suitable error response.

Table 87: TimeSynch() Definition

10.6.8.4 Delay Operations

The following functions provide a means for an algorithm to delay its own execution, or wait for a clock to reach a certain deadline.

Declaration	Result Sleep(in HandleID fromID, in HandleID toID, in TimeWarp interval);
Description	Delays the caller until the specified interval has elapsed, as measured by the clock device.
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the clock device that should respond to the request ► `interval`: The amount of time that the caller should be delayed, relative to the current clock value
Return	On success, returns `OK`. On error, returns one of the defined error constants.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device
Notes	The call may be interrupted under some circumstances, causing the infrastructure to return to the caller before the interval has elapsed. In these cases, the infrastructure should return the `WARNING` response. Note that the actual sleep time may be longer than requested due to the resolution of the clock device and operating system scheduling variances. Setting a clock using `SetTime()` while this operation is in progress has undefined effects on the delay operation.

Table 88: Sleep() Definition

Declaration	Result DelayUntil(in HandleID fromID, in HandleID toID, in TimeWarp endTime);
Description	Delays the caller until the clock reaches the indicated value
Parameters	► `fromID`: The handle ID of the current component making the request ► `toID`: The handle ID of the component that should respond to the request ► `endTime`: The time value at which the function should return, relative to the clock epoch
Return	On success, returns `OK`. On error, returns one of the defined error constants.
Implemented By	Infrastructure
Invoked By	Application, Service, or Device

Notes	The call may be interrupted under some circumstances, causing the infrastructure to return to the caller before the end time has been reached. In these cases, the infrastructure should return the `WARNING` response. Note that the actual sleep time may be longer than requested due to the resolution of the clock device and operating system scheduling variances. Setting a clock using `SetTime()` while this operation is in progress has undefined effects on the delay operation.

Table 89: DelayUntil() Definition

10.7 Non-STI Software Interfaces

JPH-NOTE: In flight software, mainly anything using the NASA Core flight system, coding standards actually disallow direct calls to POSIX API. Instead, interaction with the OS is done through an operating system abstraction layer (OSAL). Directly specifying POSIX as the operating system interface here goes against some NASA flight software coding standards.

Also, this does not translate well to languages other than C/C++. For Python one would leverage the included "os" or "sys" modules, not invoke POSIX directly. Similar for Java, Ruby, Lua, Perl, etc.

There should be a provision to allow applications to use an operating system interface other than POSIX, provided that whatever interface is used is externally documented and available on more than one platform. The application obviously will not be portable to an STI environment that does not provide the same library/interface.

Furthermore, STI applications might in turn depend on other libraries, unrelated to the operating system (e.g. FFTW, TensorFlow, SciPy, etc). A more generic approach would be to require than an application developer document the dependencies, including those of the operating system interface utilized.

For now, the POSIX requirement is included fairly unchanged from STRS, pending futher discussion.

STI applications and services may need to utilize libraries or services outside the scope of STI, such as the services provided by the operating system or additional software libraries. As such, an STI module can only be ported to an environment that also provides a compatible set of services or libraries, so it is critical to identify these dependencies.

Examples of software libraries include, but are not limited to:

- Operating system operations such as task/thread creation or synchronization
- Floating-Point mathematical operations
- Complex algorithms, such as machine learning

Most programming languages, including C/C++, also define a "standard library" in addition to the language syntax and semantics. This library is defined by the respective standards body, such as ISO/IEC for C and C++, as a set of interfaces that all compliant implementations must meet. For instance, in ISO/IEC 9899 (C), this standard library includes a minimum set of header files specifying a core set of function calls, including basic memory access, mathematical operations, and string manipulation (e.g. `memset()`, `strcmp()`, `sqrt()`, etc.).

► An STI application may use any operations defined in the standard library of the respective programming language.

► An STI application developer should avoid the use of library functions which are marked as deprecated, non cross-platform, or non thread-safe, where applicable. If no replacement or alternative exists, this dependency should be expressly noted in the application documentation.

Beyond the standard library, additional software libraries may be used for specific functions. These include, but are are not limited to:

- Accessing operating system or task scheduling resources (e.g. POSIX® or other operating system abstraction library)
- Additional mathematical computations beyond those provided by the standard library (e.g. BLAS, LAPACK, NumPy, etc.)
- Scientific or Machine Learning packages (e.g. SciPy, TensorFlow™, etc.)

10.7.1 Operating System Interface

STI applications implemented in C or C++ which do not leverage a specific 3rd party operating system abstraction library may use a subset of the POSIX® API as shown in figure 9, Software Execution Model. POSIX® refers to a family of IEEE standards 1003.n that describe the fundamental services and functions necessary to provide a UNIX®-like kernel interface to applications. POSIX® itself is not an OS but is instead specifies the programming interfaces available to the application programmer.

POSIX® specifies a set of OS interfaces and services. The specification is not bound to a single operating system, and has in fact been implemented on top of operating systems such as Digital Equipment Corporation's (DEC's) OpenVMS™ (Virtual Memory System) and Microsoft Windows®. However, the creation of POSIX® is closely coupled to the UNIX® OS and its evolution. The goal was to create a standard set of interfaces that all of the UNIX® flavors would support in order to facilitate software portability. Even though POSIX® technically refers to the family of specifications, it is more commonly used to refer specifically to IEEE 1003.1, Information Technology - Portable Operating System Interface (POSIX®), which is the core POSIX® specification.

Characteristics of POSIX® include the following:

- Application-oriented.
- Interface, not implementation.
- Source, not object, portability.
- The C-language/system interfaces written in terms of the ISO C standard.
- No superuser, no system administration.
- Minimal interface, minimally defined—core facilities of this specification have been kept as minimal as possible.
- Broadly implementable.
- Minimal changes to historical implementations.
- Minimal changes to existing application code.

The original POSIX® specification was based on a general purpose computing platform, but a series of amendments addressed the unique requirements of real-time computing. These amendments follow:

- IEEE 1003.1B-Realtime Extension.
- IEEE 1003.1C-Threads Extension.
- IEEE 1003.1D-Additional Realtime Extensions.
- IEEE 1003.1J-Advanced Realtime Extensions.
- IEEE 1003.1Q-Tracing.

These amendments were rolled into the base specification in version IEEE 1003.1-1996. IEEE 1003.13 provides a standards-based option for an STI AEP.

10.7.1.1 STI Application Environment Profile

The subset of the POSIX® API described below is used by STI applications to access platform services when no STI Infrastructure-provided API is available. The IEEE 1003.1 standard provides a means to implement a subset of the interfaces by using "Subprofiling Option Groups." These option groups specify "Units of Functionality" that can be removed from the base POSIX® specification.

IEEE 1003.13 created four AEPs that specified subsets of 1003.1 more suitable to embedded applications. These profiles follow:

- PSE51—Minimal Realtime Systems Profile.
- PSE52—Realtime Controller System Profile.
- PSE53—Dedicated Realtime System Profile.
- PSE54—Multi-Purpose Realtime System Profile.

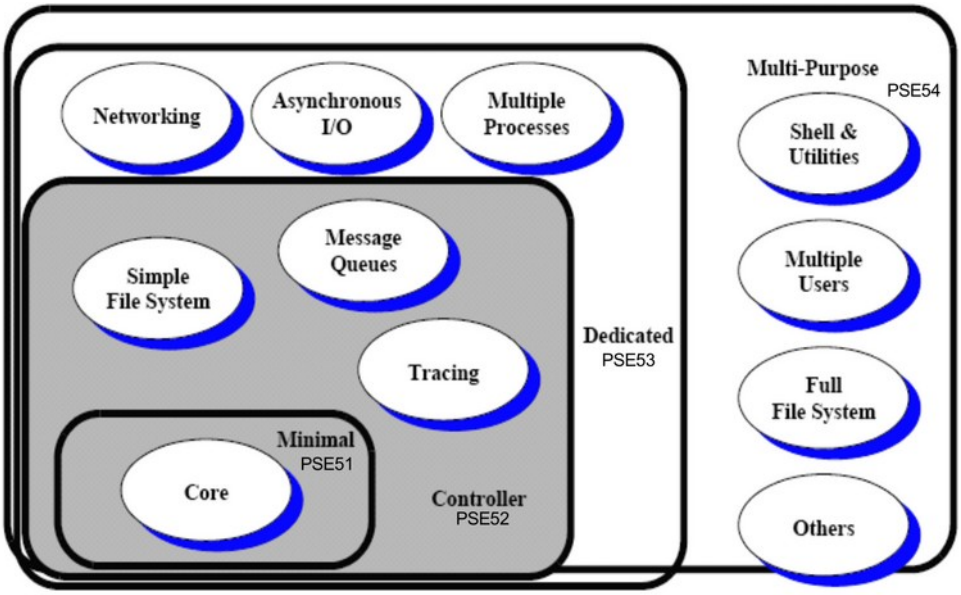

Figure 19: Profile Building Blocks

The profiles are each upwardly compatible and consist of the basic building blocks shown in figure 19, Profile Building Blocks. Each of these profiles has increasing capabilities, which increase requirements on resources. Profiles 51 and 52 runs on a single processor with no Memory Management Unit (MMU), and thus imply a single process containing one or more threads. Profile 52 adds a file system interface and asynchronous I/O. Profile 53 adds support for multiple processes, thus requiring an MMU. The last and largest profile 54 adds support for interactive users, and is almost a full POSIX® 1003.1 environment. The higher numbered profiles are supersets of the lower numbered profiles, such that PSE52 includes all the features of a PSE51.

Upward portability between profiles is supported by requiring certain APIs, such as memory locking, for profiles PSE51 and PSE52. Even though there is no MMU support on the PSE51 and PSE52 profiles, code written as if there is an MMU present will be portable among all four profiles by requiring such APIs to be defined in all four profiles. The signature of these APIs will be identical on all profiles, but the functionality will differ according to the capabilities. For example, calling a memory-locking API on a PSE51 platform with no MMU will always return

success. When this example application is ported to a PSE53 platform, the memory locking will work as intended without modification to the source code.

Currently, this specification supports platforms based on profiles PSE51 through PSE54, although PSE54 will only be used for development platforms and ground stations. Allowing multiple profiles allows the architecture to scale to different platforms. Applications developed for a specific profile are compatible with higher profiles; that is, a profile 52 application could be ported to profile PSE53 and PSE54 platform, but not vice versa. This upward scalability anticipates that smaller platforms will desire smaller profiles and will not have the resources to run larger applications that comply with the larger profiles.

Summary of Requirements for Operating System Interface

► The STI infrastructure provider shall document the set of interfaces provided by the infrastructure.

For POSIX® interfaces this should indicate the supported application profiles as described in standard IEEE 1003.13. For other operating systems or operating system abstraction layers, this should indicate the specific API or abstraction layer and associated version, where applicable.

► The STI application developer shall document the set of operating system interfaces required by the application

For POSIX® interfaces this should indicate the required application profiles as described in standard IEEE 1003.13. For other operating systems or operating system abstraction layers, this should indicate the specific API or abstraction layer and associated version, where applicable.

Regardless of the POSIX® profile implemented, STI applications should avoid use of any POSIX® function which is not thread safe, to preserve portability of application code to multi-threaded STI platforms. In addition, STI applications must not invoke any function which would cause the parent process to abort or exit (e.g. `exit()` or `abort()`) as these functions may disrupt the operation of other STI applications.

In areas where there is overlap between an STI API and a function provided by POSIX®, such as messages queues and file system access, applications must use the STI provided API.

Table 90 lists a set of common POSIX® functions and the alternative function to use in an STI application. Note that this list only contains a subset of the possible non thread-safe functions, and should not be considered exhaustive or complete. Refer to the POSIX® specification for a complete set of non thread-safe functions.

POSIX® Function(s)	**Suggested Alternate**
`asctime(),ctime(),localtime()`	STI `GetCalendarTime()`
`open(),close()`	STI `FileOpen(),FileClose()`
`mq_open()`	STI `MessageQueueCreate()`
`read(),write()`	STI `Read(),Write()`
`strtok()`	`strtok_r()`
`rand()`	`rand_r()`
`abort(), exit()`	STI `AbortApp()`
`ioctl(), mmap()`	STI `AddressRead(),AddressWrite()`
`system(), atexit()`	None; do not use

Table 90: Function Alternatives

11 External Command and Telemetry Interfaces

An STI radio cannot perform the necessary application and platform functions without an external system providing commands, accepting responses, and monitoring the radio's health and status. The STI radio implements an external interface to receive and act on the commands from the external system, translates the commands into the format expected by the application, and provides the information for monitoring the health and status of the radio. If the STI radio has the capability for new or modified OE, application software, or configurable hardware design, the external command and telemetry interfaces should be able to accept and store new files. The interface in the STI radio and in the external system, which is to provide the control, via a command sequence, to the STI radio and receive responses from an STI radio, is referred to as the STI command and telemetry interfaces. The external STI command and telemetry functionality illustrated in figure 20, Command and Telemetry Interfaces, typically resides on the spacecraft's flight computer, and/or it may reside on a ground station or another spacecraft.

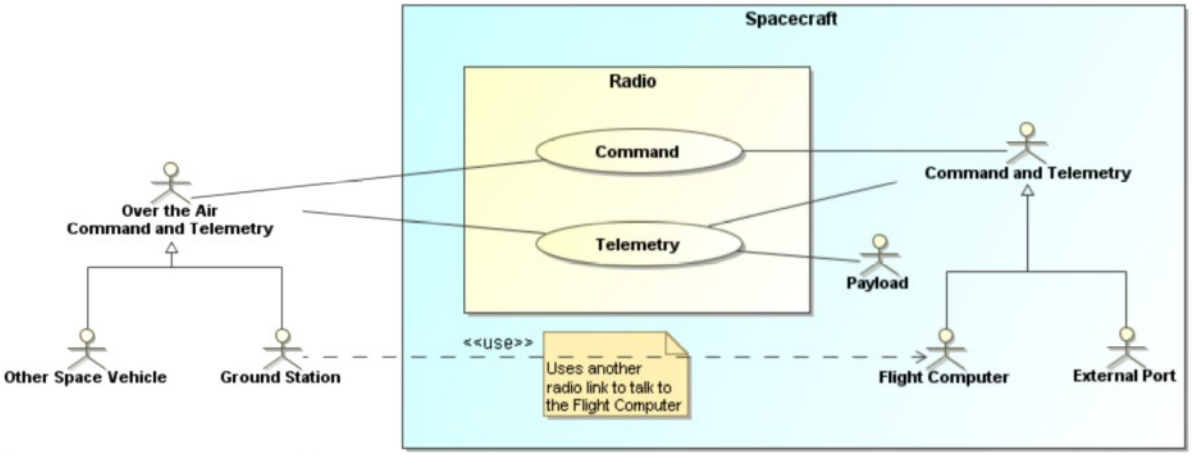

Figure 20: Command and Telemetry Interfaces

This shared capability implies that the STI radio is capable of performing the interface functions. Within the STI radio, if there are data stored on the radio that are to be transferred to an external system, the capability is to exist to send data using a mission-specific protocol to the receiver (flight computer, ground station, or other spacecraft) and capability in the receiver to process those data or write those data to a file or download service or to a storage area that is accessible from both. The reverse capability for STI radio control is also necessary: The external system is capable of sending commands using a mission-specific protocol and the STI radio is capable of validating, deciphering, and processing those commands. For example, data coming over the Flight Computer Interface are interpreted by the Command and Control Manager as shown in figure 20 and are processed by the STI infrastructure.

Within the STI radio, components of the command and telemetry interfaces are necessary to provide the interfaces between the STI OE and the STI command and telemetry functionality on the external system. The command and telemetry interfaces may include a standard type of mechanical, electrical, and functional spacecraft bus interface, such as MIL–STD–1553, Digital Time Division Command/Response Multiplex Data Bus; command and telemetry interpretation; and translation of the command set to the STI standard necessary for application control. The protocol, command set, and telemetry set for the STI command and telemetry interfaces are *not* standardized, and can be customized according to the needs of any particular deployment. However, a number of interface and behavioral requirements are required.

Requirements for External Command and Telemetry Interface

► An STI platform shall accept, validate, and respond to external commands.

► An STI platform shall execute external application control commands using the standardized STI APIs.

► An STI platform provider shall document any external commands describing their format, function, and any STI methods invoked.

► The STI infrastructure shall use the STI_Query method to service external system requests for information and to provide telemetry data about an STI application.

The telemetry set should contain some or all of the following parameters:

- **Electrical Conditions:** Voltage, current, and power consumption.
- **Environmental Conditions:** Temperature, Pressure.
- **Module Configuration**: Module type/location, Hardware revision.
- **Self-test Status:** RAM/ROM, file system, software revision, and individual module test status.
- **Operating Environment Status:** Infrastructure software revision, name/ID/state of components, available memory for data and files
- Other Application-specific parameters

The command set may contain some or all of the following actions:

- **Application Instantiation and Deletion:** Manually create or delete a waveform or device.
- **Property Set:** Query or Configure a specific component property via the STI PropertySet API.
- **File Operations**: Query, delete, or rename files via the STI File API.
- **Invoke Self-tests:** Interface to the STI TestableObject API.
- **Device Operations:** Manually load, flush, or reset a device via the STI Device API.

If the command interface lies on a network containing other devices, the infrastructure should implement some form of command authentication, to reduce the likelihood that commands are received in error or from an unauthorized source. Furthermore, the infrastructure may also implement encryption on the command and telemetry interfaces to ensure that the data is not disclosed to other entities in the system while in transit. Any such security procedures should be implemented at the network transport level, which is outside the scope of this specification.

The specific command or telemetry set available for use is always at the discretion of the system integrator. While the set described here is potentially useful for a development platform, flight operations may choose to use an entirely different set.

Annex A: Language Translations

This appendix describes some specific mappings to programming languages for STI interfaces. This section is intended to clarify certain aspects of the IDL mappings to ensure that different implementations will remain consistent in regard to these interface definitions.

Many of the interface definitions in this specification are provided as OMG Interface Definition Language (IDL) fragments. OMG also specifies a specific method for mapping these interfaces to source code in various common programming languages, and the STI implementation of these interfaces should adhere to these mappings where relevant.

Earlier versions of the OMG IDL specification were specifically designed for defining the interfaces within a CORBA environment. IDL has since been revised as a general purpose interface definition language, and has been released independently from CORBA since version 3.5. While a compliant implementation of STI may utilize a CORBA-like layer to exchange data between modules, there is no requirement for nor assumption of a CORBA environment within STI. As such, the function prototypes or interface definitions based on the IDL fragments in this specification must not directly include any CORBA references.

All IDL fragments in this document shall be interpreted as belonging to an IDL module called "STI", with interface and identifier names mapped accordingly.

A.1 C Language Mapping

The C programming language is standardized as ISO/IEC 9899, with a specific revision to the standard identified by a year number suffix (e.g. ISO/IEC 9899:1999). The STI architecture should be implementable in any current or future version of the C programming language.

Naming Conventions

Unlike other languages, the C language does not include the concept of a "namespace" or "module" to avoid identifier name collisions between global-scope symbols in separate libraries or code units. As such, it is common practice to add a prefix to all global identifier names supplied by a library or module as a means of differentiation.

All infrastructure-provided functions, constants, and types defined in this specification shall be denoted with an "`STI_`" prefix when mapped to identifiers in the C programming language. For example, the "`Instance`" type shall be named "`STI_Instance`", the "`OK`" result value constant shall be named "`STI_OK`", the "`Write`" method shall be named "`STI_Write`", and so forth.

All application-provided implementation written in the C language shall be denoted with a prefix defined by the application. For instance, if an application were named "Example", the application-provided application control methods may be called "`Example_APP_Instantiate`", "`Example_APP_Start`", and so forth.

Header Files

The following header files must be provided by the infrastructure, such that applications can utilize the `#include` preprocessor directive to utilize the respective resources:

Include File	Provides
`STI.h`	C language STI data types and abstract object definitions. This file provides declarations of all data types described in section 10.4.

Include File	Provides
STI_APIs.h	C language function prototype declarations for all infrastructure-provided API calls. This file provides declarations of all calls described in section 10.6.
STI_ApplicationControl.h	C language function prototype declarations associated with ApplicationControl interface, as described in section 10.5.2.
STI_DeviceControl.h	C language function prototype declarations associated with DeviceControl interface, as described in section 10.5.3.
STI_Source.h	C language function prototype declarations associated with the Sink interface, as described in section 10.5.4.1.
STI_Sink.h	C language function prototype declarations associated with the Source interface, as described in section 10.5.4.2.
STI_RandomAccess.h	C language function prototype declarations associated with the RandomAccess interface, as described in section 10.5.4.3.

Table 91: C Language Header Files

Interface Type Mappings

Table 5, Infrastructure-provided Data Types, in section 10.4.1 indicates the general semantics of each STI-defined type. These general semantics, in turn, determine the proper method to pass a value or object of that type through an IDL-defined interface or function definition.

The table below indicates the basic type mappings for the C language. This table also indicates whether an operand should be passed as value or as a pointer/reference, and how the pointer type should be qualified, if applicable. For operations which utilize an abstract base type containing application-defined data of arbitrary size (e.g. Message and PropertyValue types), the size of this data must also be specified. In these cases, a single object in the IDL fragment will translate to two arguments in the C function prototype. This also applies to strings, as the C language implements strings as a pointer to the `char` type, rather than as a distinct value type in itself.

Semantics	Usage	Pass As	C Data Type(s)	Applicable to
Integer, Enumeration, or aggregate value	in, return	Value	STI_<type>	Result, HandleID, TimeWarp, Access, etc.
	out, inout	Pointer to Value	STI_<type> *	
string	in, return	Pointer	const char *	Object Names
	out, inout	Pointer and Size	char *, size_t	
Abstract Object	in	Pointer and Size	const STI_<type> *, size_t	Message, PropertyValue
	out	Pointer and Size	STI_<type> *, size_t	
Base Type	any	Pointer	STI_Instance *	Context Objects

Table 92: C Language Data Type Mapping

Inheritance and Base Types

Although C is not an object oriented language by nature, the same basic concepts can still be manually implemented by the programmer through use of specific patterns and by utilizing type casting where necessary. The main

requirement is that structure definitions be defined appropriately such that a pointer to a base structure can be reliably converted to a derived structure and vice versa.

The first element of a C structure is guaranteed to be at the same memory address as the structure itself, as specified in ISO/IEC 9899 section 6.7.2.1, as follows:

> A pointer to a structure object, suitably converted, points to its initial member, and vice versa. There may be unnamed padding within a structure object, but not at its beginning.

Given this requirement, the concept of single inheritance may be implemented simply by ensuring that the "base type" of a given structure is declared as its first element. For STI, the base type of all context objects is the `Instance` type. The specific content of the `Instance` type is implementation-defined, but the infrastructure must provide this type such that it is suitable for use as a base type, as in this example:

```
typedef struct
{
    STI_Instance Base;
    int LocalValue;
} Example_Object;
```

Using this definition, a pointer to the base object (`STI_Instance*`) may be safely typecast by the application to the derived object (`Example_Object*`) and vice-versa. Note that while this approach generally works for simple cases, more complex applications may necessitate a different approach. The STI infrastructure only stipulates that interaction with the infrastructure takes place using an Instance object; more complex applications may in turn use this object to index into a larger state table or database.

Interface Operations

All operations defined in the STI application or device control interfaces in section 10.5 require a context object, which is the in-memory data structure comprising the device or application state. This is an application defined structure that may contain any arbitrary state information needed by the application. In object oriented languages this object is often referred to as the "self" or "this" object, and is usually implicitly supplied through the respective language internal mechanisms.

Since the C programming language does not provide these object oriented features, the context object must be explicitly included as an argument in the function prototype. This context object shall always be the first parameter to the function, followed by the remainder of the operands specified in the interface definition.

STI requires that all such context objects in the system are derivatives of the infrastructure-defined `Instance` type. Therefore, in the C programming language, all interaction between the infrastructure and the application will use a pointer to the "`STI_Instance`" type to identify the target of the operation. For example, the C prototype for the `APP_Instance()` and `APP_Start()` operations in the "Example" application would be:

```
STI_Instance*   Example_APP_Instance(STI_HandleID id, const char *name);
STI_Result      Example_APP_Start(STI_Instance *inst);
```

A.2 C++ Language Mapping

The C++ programming language is standardized as ISO/IEC 14882, with a specific revision to the standard identified by a year number suffix (e.g. ISO/IEC 14882:2003). The STI architecture should be implementable in any current or future version of the C++ programming language.

Mapping of the STI interfaces to C++ should follow the guidelines set forth in the OMG IDL C++ language mapping. However, in STI there is no assumption or dependence on CORBA types or interfaces. This section is intended to clarify how the C++ language mapping applies to STI.

Naming Conventions

All STI infrastructure-provided functions, constants, and types shall be defined within a C++ namespace called "STI". For example, the "`Instance`" type shall be named "`STI::Instance`", the "`OK`" result value constant shall be named "`STI::OK`", the "`Write`" method shall be named "`STI::Write`", and so forth.

Header Files

The following header files must be provided by the infrastructure, such that applications can utilize the `#include` preprocessor directive to utilize the respective resources:

Include File	Provides
`STI.hh`	Fundamental STI data types and abstract object definitions. This file provides declarations of all data types described in section 10.4.
`STI_APIs.hh`	Function prototype declarations for all infrastructure-provided API calls. This file provides declarations of all calls described in section 10.6.
`STI_ApplicationControl.hh`	ApplicationControl interface class definition, as described in section 10.5.2.
`STI_DeviceControl.hh`	DeviceControl interface class definition, as described in section 10.5.3.
`STI_Source.hh`	Sink interface class definition, as described in section 10.5.4.1.
`STI_Sink.hh`	Source interface class definition, as described in section 10.5.4.2.
`STI_RandomAccess.hh`	RandomAccess interface class definition, as described in section 10.5.4.3.

Table 93: C++ Language Header Files

Constructor and Destructor

STI defines the `APP_Instance()` and `APP_Destroy()` methods as a means to construct and destruct instances, rather than relying on language-specific paradigms to invoke a class constructor or destructor. These should be implemented as static methods in the C++ application class. This aligns with a "factory" design pattern that allows additional application control over the construction process. When the infrastructure invokes the factory function, the application should invoke the class constructor appropriately, and return the newly-constructed object.

Interface Classes

All other application and device control interfaces defined in section 10.5 shall each be mapped to a C++ abstract interface base class provided by the infrastructure. The class shall declare each of the operations as a pure virtual function, which in turn requires that any derivative class provide an implementation as a prerequisite to being instantiated.

For example, the following class definition would represent the `ControllableComponent` interface:

```
namespace STI
{

class ControllableComponent
```

```
{
    public:
        virtual Result APP_Start() = 0;
        virtual Result APP_Stop() = 0;
};

}
```

All application-provided methods shall be class member functions of an application-defined class inheriting from some or all of these abstract interface classes.

Table 5, Infrastructure-provided Data Types, in section 10.4.1 indicates the general semantics of each STI-defined type. These general semantics, in turn, determine the proper method to pass a value or object of that type through an IDL-defined interface or function definition.

Semantics	Usage	Pass As	C++ Data Type(s)	Applicable to
Integer, Enumeration, or aggregate value	in, return	Value	`STI::<type>`	Result, HandleID, TimeWarp, Access, etc.
	out, inout	Pointer to Value	`STI::<type> *`	
string (see note)	in, return	Pointer	`const char *`	Object Names
	out, inout	Pointer and Size	`char *, size_t`	
Abstract Object	in	Pointer and Size	`const STI::<type> *, size_t`	Message, PropertyValue
	out	Pointer and Size	`STI::<type> *, size_t`	
Base Type	any	Pointer	`STI::Instance *`	Context Objects

Table 94: C++ Language Data Type Mapping

Note: The "string" types in C++ shall utilize C-style string representations (pointer to `char`) rather than the `std::string` type. This is because the C++ string type typically relies on dynamic memory allocation, and usage of this type may also introduce additional compile-time and run-time dependencies on the C++ standard library. Using C-style strings also facilitates an infrastructure implementation supporting both C and C++.

A.3 Python Mapping

Python is an object oriented programming language developed by the Python Software Foundation. The language has seen significant adoption by the scientific and research communities, and is often utilized for prototyping software algorithms.

Python is an interpreted language and utilizes a dynamic type system with automatic memory management. As such, it may not be suitable for flight software environments where strict deterministic behavior is required. However, during the SDR development stages, the ability to integrate existing Python applications into an SDR may be highly useful and beneficial. This can be accomplished by mapping the STI interfaces to a Python language environment.

Naming Conventions

All STI infrastructure-provided functions, constants, and types shall be provided through a Python module called "STI". All infrastructure-provided types and methods shall be available through this module. For example, the "`Instance`" type shall be identified as "`STI.Instance`", the "`OK`" result value constant shall be named "`STI.OK`", the "`Write`" method shall be named "`STI.Write`", and so forth.

Application Classes

Applications utilizing the STI infrastructure shall utilize the standard Python module import mechanisms to access the STI infrastructure. All application base classes utilized with STI shall inherit from the "Instance" class provided through this module.

For example, an application would typically have an "import" statement at the beginning of the source file, followed by an application class definition.

```
import STI

class ExampleWaveform(STI.Instance):
    …
```

Constructor and Destructor

The Python application module must also provide an implementation of the APP_Instance and and APP_Destroy methods, implementing a "factory" design pattern that can be invoked by the infrastructure. These must be implemented as static methods in the application class.

Interface Operations

Unlike C++, the methods in a Python class are dynamic and do not need to be explicitly declared at compile time. Therefore, applications do not need to inherit from an interface class as in C++. Instead, implementation of any application-provided interface method defined in section 10.5 is simply a matter of defining a matching method within the application class.

For example, the following class definition would implement the `ControllableComponent` interface:

```
class ExampleWaveform(STI.Instance):

    def APP_Start(self):
        # Implementation-defined action…
        return STI.OK

    def APP_Stop(self):
        # Implementation-defined action…
        return STI.OK
```

All application-provided methods shall be class member functions of an application-defined class inheriting from some or all of these abstract interface classes.

Being a fully object oriented language with automatic memory management, Python represents all values in software code as an logical object of some type. Unlike C and C++, the actual memory storage and representation of these objects is hidden from the developer, and there is no direct equivalent of a "pointer" type. However, Python does provide some data types that can directly deal with memory reservation and access, and these can be used to exchange data directly with C/C++ software. Since all Python objects are fundamentally self describing, with a type and size known to the interpreter, the STI interfaces do not need to explicitly indicate size information when passing abstract buffer objects through the interface.

Python classifies certain object types as "immutable", which include strings, integers, and other fundamental value types. Once instantiated, these values can never be modified; instead, a new, distinct value object must be created, and the previous object can be destroyed. On the other hand, aggregate types such as classes, dictionaries, and lists are "mutable", meaning that the content can be modified after instantiation. Some fundamental objects have both mutable and immutable variants (e.g. byte/bytearray, frozenset/set, etc.). When translating from IDL, immutable types can only be used to implement "in" or "return" parameter values from an operation definition. Parameters designated as "out" or "inout" must only use mutable types.

Table 5, Infrastructure-provided Data Types, in section 10.4.1 indicates the general semantics of each STI-defined type. These general semantics, in turn, define the expected mutability of a value of the given type, and therefore its applicability to IDL-defined operations.

Semantics	Mutability	Python Data Type	Applicable to
Integer	immutable	`STI.<type>`	Result, HandleID, etc.
Enumeration	immutable	`str`	Access, CalendarKind
string	immutable	`str`	Object Names
Aggregate Value	mutable	`STI.TimeWarp`	TimeWarp
Abstract Object	mutable	Any object type implementing the Python “buffer protocol”, such as `bytearray`.	Message, PropertyValue
Base Type	mutable	`STI.Instance`	Context Objects

Table 95: Python Language Data Type Mapping

Note that Python does not implement enumerated data types as C/C++ do. Rather, enumerated values are implemented as free-form strings, and no separate type needs to be defined for these values. Due to the immutability of string values in Python, using free-form strings in this manner does not incur the same efficiency issues that it would in C/C++.

Annex B: Conformance Table

The tables in this appendix summarize all requirements for the hardware and software architecture of an STI compliant system.

JPH-NOTE: This section is incomplete. Maintained in accompanying spreadsheet for now, to track changes. Once reviewed the content will be moved here.

ID	Requirement	Applicable To	Reference
STI-1	An STI platform shall have a known state after completion of the power-up process.		

