NOT MEASUREMENT SENSITIVE

NASA TECHNICAL HANDBOOK

Office of the NASA Chief Engineer

NASA-HDBK-4009A
w/CHANGE 1:
ADMINISTRATIVE/
EDITORIAL CHANGE
2018-05-11

Approved: 2018-03-14
Superseding NASA-HDBK-4009
(Baseline)

SPACE TELECOMMUNICATIONS RADIO SYSTEM (STRS) ARCHITECTURE STANDARD RATIONALE

DOCUMENT HISTORY LOG

Status	Document Revision	Change Number	Approval Date	Description
Baseline			2014-06-05	Initial Release
Revision	A		2018-03-14	Significant changes were made to this NASA Technical Handbook. It is recommended that it be reviewed in its entirety before implementation. Key changes were: Changed configuration file requirements to recommendations and possible project requirements. Moved example of possible configuration file from the NASA Technical Standard to an appendix in this Handbook. Removed all precondition and postcondition to a known state from the Standard and moved them to the Handbook as suggestions. Added many new sections to explain how to use the CCSDS format to read and write data and how to add asynchronous read and write capability, as well as to answer the questions asked by users. Deleted 5 requirements outright, made 7 into project suggestions, added 27 requirements to standardize STRS Devices, separated 2 types of queueing, augment time adjustment, etc. Moved the examples for each requirement from the Standard to the Handbook.
Change		1	2018-05-11	Administrative/Editorial Changes—Table 66, 4th paragraph, changed OEClockAppName and OEClock Kind; changed 6.17,1st paragraph; 6.23, deleted 1st paragraph; added 6.24 and renumbered remaining sections; deleted 6.28; added Note in 7.9; changed Notes in 7.10 and 7.17; changed Example in 7.29, 7.30, 7.32 (for C), 7.33, 7.40, 7.50, 7.53, 7.69, 7.85, 7.115, 7.117, 7.131, 7.132, 7.133; changed See Also to Verification Method in 7.54, 7.55, 7.56, 7.57, 7.82, and 7.100; deleted See Also in 7.48, 7.51, 7.61, 7.62, 7.63, 7.64, 7.65, 7.68, 7.69, 7.71, 7.72,

Status	Document Revision	Change Number	Approval Date	Description
Change		1	2018-05-11	Continued
				7.74, 7.75, 7.76, 7.77, 7.80, 7.81, 7.88, 7.115, 7.116, 7.126, 7.127, 7.129, 7.130, and 7.134; deleted Example in 7.54, 7.55, 7.56, 7.57, 7.77, and 7.82; changed See Also in 7.83, 7.84, 7.87, and 7.118; changed Traced-from in 7.108; added Verification Method in 7.134; deleted See Also, added Verification Method, and changed Example in 7.134 and 7.135; changed PUBSUB list, REGISTER list, RPN, and VALUE list in A.2.

FOREWORD

This NASA Technical Handbook is published by the National Aeronautics and Space Administration (NASA) as a guidance document to provide engineering information; lessons learned; possible options to address technical issues; classification of similar items, materials, or processes; interpretative direction and techniques; and any other type of guidance information that may help the Government or its contractors in the design, construction, selection, management, support, or operation of systems, products, processes, or services.

This NASA Technical Handbook is approved for use by NASA Headquarters and NASA Centers and Facilities. It may also apply to the Jet Propulsion Laboratory (a Federally Funded Research and Development Center (FFRDC)), other contractors, recipients of grants and cooperative agreements, and parties to other agreements only to the extent specified or referenced in applicable contracts, grants, or agreements.

This NASA Technical Handbook establishes the key rationale, explanatory material, and additional information to support NASA-STD-4009A, Space Telecommunications Radio System (STRS) Architecture Standard. This architecture is a standard for reconfigurable communication transceiver developments among NASA missions.

NASA-STD-4009A strives to provide commonality among NASA radio developments to take full advantage of emerging software-defined radio (SDR) technologies from mission to mission. This architecture serves as an overall framework for the design, development, operation, and upgrade of these software-based radios.

Requests for information should be submitted via “Feedback” at https://standards.nasa.gov. Requests for changes to this NASA Technical Handbook should be submitted via MSFC Form 4657, Change Request for a NASA Engineering Standard.

Original signed by	03/14/2018
Ralph R. Roe, Jr.	Approval Date
NASA Chief Engineer	

TABLE OF CONTENTS

SECTION		PAGE
DOCUMENT HISTORY LOG		**2**
FOREWORD		**4**
TABLE OF CONTENTS		**5**
LIST OF APPENDICES		**10**
LIST OF FIGURES		**11**
LIST OF TABLES		**11**
1.	**SCOPE**	**12**
1.1	Purpose	12
1.2	Applicability	12
2.	**APPLICABLE DOCUMENTS**	**12**
2.1	General	12
2.2	Government Documents	13
2.3	Non-Government Documents	13
2.4	Order of Precedence	14
3.	**ACRONYMS, ABBREVIATIONS, AND DEFINITIONS**	**14**
3.1	Acronyms and Abbreviations	14
3.2	Definitions	16
4.	**HIGH-LEVEL RATIONALE**	**16**
4.1	Operational Requirements	17
4.2	Operating Environment (OE) Requirements	17
4.3	Documentation Requirements	18
4.4	Source Code Requirements	18
4.5	Configuration File or Script File Recommendation	19
4.6	Roles and Responsibilities	23
5.	**HOW TO USE STRS APIs**	**25**
5.1	How to Associate an FPGA with an STRS Application	26
5.2	How to Load an FPGA	26
5.3	How to Set Attributes	27
5.4	How to Get Attributes	27
5.5	How to Push Packets	27
5.6	How to Pull Packets	27
5.7	How to Process Errors	27
5.8	How to Make Multiple Instances of an Application	28
5.9	How to Map Memory Locations	28
5.10	When to Use STRS_Log and STRS_Write	29

TABLE OF CONTENTS (Continued)

SECTION		PAGE
5.11	Difference Between Run Test and Ground Test	29
5.12	When to Use Start/Stop, Load/Unload, and Open/Close	30
5.13	How to Read and Write Data	31
5.14	How to Add Asynchronous Publisher/Subscriber Functionality	32
6.	**QUESTIONS AND ANSWERS**	**32**
6.1	Fault State and Use of the ERROR, WARNING, and FATAL Queues	32
6.2	Pub/Sub Messaging and Queues Need Clarification	33
6.3	What is an STRS Device?	35
6.4	How to Configure and Control SDR Hardware	37
6.5	STRS Infrastructure Methods Do Not Belong to Any Class	38
6.6	Explain Clocks and Timers	39
6.7	FPGA Partial Reconfiguration	40
6.8	Compliance Testing	40
6.9	Configuration Files Examples	41
6.10	C Language Naming Duplication	43
6.11	Sequence Diagrams Depicting STRS API Calls	45
6.12	Why are APP_Instance and APP_Initialize Separate?	48
6.13	Why Start with SCA?	49
6.14	Security for STRS	50
6.15	What is Configurable Hardware Design?	51
6.16	Why is there STRS_InstantiateAPP and no STRS_Instance method?	51
6.17	Uniqueness of Handle Names and IDs	52
6.18	Are there any exception-safety rules?	52
6.19	What does it mean that an STRS Device or STRS Service may be part of the OE?	52
6.20	How does the application know how to put the data (address, command, data) into the buffer in the specialized hardware?	53
6.21	Can STRS applications run in multiple address spaces?	53
6.22	Does an STRS application require a main entry point?	54
6.23	How is STRS_TimeSynch used to adjust time?	55
6.24	How is Clock Rate Adjustment Used?	55
6.25	What is OE-Specified String for the Application to be Instantiated?	56
6.26	STRS Radio Startup Process, Platform Diagnostics, and Built-in Test?	58
6.27	Cognitive, Navigation, and Other Services	59
6.28	C and C++ Compatibility?	59

TABLE OF CONTENTS (Continued)

SECTION		PAGE
7.	**STRS REQUIREMENTS, RATIONALE, AND VERIFICATION METHOD**	**60**
7.1	STRS-1 Power Up	61
7.2	STRS-2 Provide Platform Diagnostics	61
7.3	STRS-3 Use Platform Diagnostics (Deleted)	62
7.4	STRS-4 Document Resources	62
7.5	STRS-5 Document Capability	62
7.6	STRS-6 Document Radio Frequency (RF) Behavior	63
7.7	STRS-7 Document Module Interfaces	63
7.8	STRS-8 Document Module Control	64
7.9	STRS-9 Document Power	64
7.10	STRS-10 STRS Application Uses OE	65
7.11	STRS-11 OE Uses HAL	65
7.12	STRS-12 STRS Application Repository	66
7.13	STRS-13 OE Controls Signal-Processing Module (SPM)	67
7.14	STRS-14 Provide Platform-Specific Wrapper	68
7.15	STRS-15 Document Platform-Specific Wrapper	68
7.16	STRS-16 Use C/C++ Waveform (WF) Interface	69
7.17	STRS-17 OE Uses STRS Application Control API	69
7.18	STRS-18 Use C/C++ Compile-Time	70
7.19	STRS-19 Use C/C++Run-Time	70
7.20	STRS-20 Include STRS_ApplicationControl.h	70
7.21	STRS-21 Provide STRS_ApplicationControl.h	71
7.22	STRS-22 STRS_ApplicationControl Base Class	71
7.23	STRS-23 Include STRS_Sink.h	72
7.24	STRS-24 Provide STRS_Sink.h	72
7.25	STRS-25 STRS_Sink Base Class	72
7.26	STRS-26 Include STRS_Source.h	73
7.27	STRS-27 Provide STRS_Source.h	73
7.28	STRS-28 STRS_Source Base Class	74
7.29	STRS-29 APP_Configure	74
7.30	STRS-30 APP_GroundTest	75
7.31	STRS-31 APP_Initialize	76
7.32	STRS-32 APP_Instance	77
7.33	STRS-33 APP_Query	77
7.34	STRS-34 APP_Read	78
7.35	STRS-35 App_ReleaseObject	79
7.36	STRS-36 APP_RunTest	79
7.37	STRS-37 APP_Start	80

TABLE OF CONTENTS (Continued)

SECTION		PAGE
7.38	STRS-38 APP_Stop	81
7.39	STRS-39 APP_Write	81
7.40	STRS-40 STRS_Configure	82
7.41	STRS-41 STRS_GroundTest	83
7.42	STRS-42 STRS_Initialize	84
7.43	STRS-43 STRS_Query	84
7.44	STRS-44 STRS_ReleaseObject	85
7.45	STRS-45 STRS_RunTest	86
7.46	STRS-46 STRS_Start	87
7.47	STRS-47 STRS_Stop	87
7.48	STRS-48 STRS_AbortApp	88
7.49	STRS-49 STRS_GetErrorQueue	89
7.50	STRS-50 STRS_HandleRequest	89
7.51	STRS-51 STRS_InstantiateApp	90
7.52	STRS-52 STRS_IsOK	91
7.53	STRS-53 STRS_Log	92
7.54	STRS-54 STRS_Log Error	92
7.55	STRS-55 STRS_Log Fatal	93
7.56	STRS-56 STRS_Log Warning	93
7.57	STRS-57 STRS_Log Telemetry	93
7.58	STRS-58 STRS_Write	94
7.59	STRS-59 STRS_Read	94
7.60	STRS-60 Device Control (Deleted)	95
7.61	STRS-61 STRS_DeviceClose	95
7.62	STRS-62 STRS_DeviceFlush	96
7.63	STRS-63 STRS_DeviceLoad	96
7.64	STRS-64 STRS_DeviceOpen	97
7.65	STRS-65 STRS_DeviceReset	97
7.66	STRS-66 STRS_DeviceStart (Deleted)	98
7.67	STRS-67 STRS_DeviceStop (Deleted)	98
7.68	STRS-68 STRS_DeviceUnload	98
7.69	STRS-69 STRS_SetISR	99
7.70	STRS-70 STRS_FileClose	99
7.71	STRS-71 STRS_FileGetFreeSpace	100
7.72	STRS-72 STRS_FileGetSize	101
7.73	STRS-73 STRS_FileGetStreamPointer	101
7.74	STRS-74 STRS_FileOpen	102
7.75	STRS-75 STRS_FileRemove	103
7.76	STRS-76 STRS_FileRename	103

TABLE OF CONTENTS (Continued)

SECTION **PAGE**

7.77 STRS-77 Use Messaging API 104
7.78 STRS-78 STRS_QueueCreate (Deleted) 104
7.79 STRS-79 STRS_QueueDelete (Deleted) 104
7.80 STRS-80 STRS_Register 105
7.81 STRS-81 STRS_Unregister 105
7.82 STRS-82 Use Time Control API 106
7.83 STRS-83 STRS_GetNanoseconds 106
7.84 STRS-84 STRS_GetSeconds 107
7.85 STRS-85 STRS_GetTime 107
7.86 STRS-86 STRS_GetTimeWarp 108
7.87 STRS-87 STRS_SetTime 109
7.88 STRS-88 STRS_TimeSynch 109
7.89 STRS-89 Provide STRS.h 110
7.90 STRS-90 Provide POSIX® 110
7.91 STRS-91 Use POSIX® 111
7.92 STRS-92 Document HAL 112
7.93 STRS-93 OE Uses HAL (Deleted) 113
7.94 STRS-94 External Commands 113
7.95 STRS-95 Use STRS APIs 114
7.96 STRS-96 Use STRS_Query 114
7.97 STRS-97 Use STRS_Log (Deleted) 115
7.98 STRS-98 Document Platform for XML (Project Option) 115
7.99 STRS-99 Document WF for XML (Deleted) 115
7.100 STRS-100 Provide XML File (Project Option) 116
7.101 STRS-101 XML Content (Project Option) 116
7.102 STRS-102 Provide XML Schema (Project Option) 117
7.103 STRS-103 Provide XML Transformation Tool (Project Option) 118
7.104 STRS-104 Provide XML Transformed (Project Option) 118
7.105 STRS-105 OE Provides API in C 119
7.106 STRS-106 Use STRS.h 119
7.107 STRS-107 Document External Commands 119
7.108 STRS-108 Document Thermal and Power Limits 120
7.109 STRS-109 Provide General-Purpose Processing Module 120
7.110 STRS-110 Provide STRS_APIs.h 121
7.111 STRS-111 Include STRS_DeviceControl.h 121
7.112 STRS-112 Provide STRS_DeviceControl.h 121
7.113 STRS-113 STRS_DeviceControl Base Class 122
7.114 STRS-114 APP_Destroy 122
7.115 STRS-115 APP_GetHandleID 123

TABLE OF CONTENTS (Continued)

SECTION		PAGE
7.116	STRS-116 APP_GetHandleName	124
7.117	STRS-117 STRS_GetHandleName	124
7.118	STRS-118 STRS_ValidateHandleID	125
7.119	STRS-119 STRS_ValidateSize	126
7.120	STRS-120 DEV_Close	126
7.121	STRS-121 DEV_Flush	127
7.122	STRS-122 DEV_Load	128
7.123	STRS-123 DEV_Open	129
7.124	STRS-124 DEV_Reset	129
7.125	STRS-125 DEV_Unload	130
7.126	STRS-126 STRS_MessageQueueCreate	131
7.127	STRS-127 STRS_MessageQueueDelete	131
7.128	STRS-128 STRS_PubSubCreate	132
7.129	STRS-129 STRS_PubSubDelete	132
7.130	STRS-130 Document STRS Clock/Timer	133
7.131	STRS-131 STRS_GetTimeAdjust	134
7.132	STRS-132 STRS_SetTimeAdjust	134
7.133	STRS-133 STRS_Sleep	136
7.134	STRS-134 STRS Platform Queryable Parameters	136
7.135	STRS-135 STRS Application Queryable Parameters	137

LIST OF APPENDICES

APPENDIX		PAGE
A	Example Configuration Files	139
A.1	STRS Platform Configuration File Hardware Example	139
A.2	STRS Platform Configuration File Software Example	141
A.3	STRS Application Configuration File Example	144
B	References	148

LIST OF FIGURES

FIGURE		PAGE
1	STRS Application Recommended State Diagram	23
2	Roles and Products	25
3	Memory Map	29
4	Sample Publisher-Subscriber Sequence Diagram	34
5	STRS Application/Device Structure	37
6	Example of Predeployed Configuration File for Appendix A	41
7	Example of Predeployed Configuration File for Application WF_1 for Appendix A	42
8	Obtain Array of Pointers to Methods	44
9	Simplified Sequence Diagram for STRS_InstantiateApp	46
10	Simplified Sequence Diagram for STRS_AbortApp	47
11	Simplified Sequence Diagram for STRS_Configure	48
12	Multiple Connected Radios	54
13	XML Transformation and Validation	57
14	Example of Hardware Portion of STRS Platform Configuration File	139
15	Example of Software Portion of STRS Platform Configuration File	141
16	Example of STRS Waveform Configuration File	144
17	Example of STRS Device Configuration File	146

LIST OF TABLES

TABLE		PAGE
1	Substitutions for Figure 11	48
2	STRS Architecture Standard, Table 77, Replacements for Unsafe Functions	112

SPACE TELECOMMUNICATIONS RADIO SYSTEM (STRS) ARCHITECTURE STANDARD RATIONALE

1. SCOPE

1.1 Purpose

The purpose of this NASA Technical Handbook is to present the rationale which underlays the requirements contained in NASA-STD-4009A, Space Telecommunications Radio System (STRS) Architecture Standard, the companion document to this NASA Technical Handbook. Supporting examples and further descriptions for clarification of portions of NASA-STD-4009A are also provided. Answers prompted by questions from the Space Communications and Navigation (SCaN) Testbed partners, who created the first space implementation of STRS, are also included. As the NASA Technical Standard evolves, minor corrections and updates to obsolete information will be added to the NASA Technical Handbook. The NASA Technical Handbook is aimed at helping readers and implementers of NASA-STD-4009A understand the NASA Technical Standard.

NASA-STD-4009A provides an STRS overview, background, and detailed descriptions that might be useful to the reader not familiar with the STRS architecture.

1.2 Applicability

This NASA Technical Handbook is applicable to providing the rationale as well as additional information to NASA-STD-4009A, which is a standard for reconfigurable communication transceiver developments among NASA missions.

This NASA Technical Handbook is approved for use by NASA Headquarters and NASA Centers and Facilities. It may also apply to the Jet Propulsion Laboratory (a Federally Funded Research and Development Center (FFRDC)), other contractors, recipients of grants and cooperative agreements, and parties to other agreements only to the extent specified or referenced in their applicable contracts, grants, or agreements.

This NASA Technical Handbook, or portions thereof, may be referenced in contract, program, and other Agency documents for guidance. When it contains procedural or process requirements, they may be cited in contract, program, and other Agency documents.

2. APPLICABLE DOCUMENTS

2.1 General

The documents listed in this section are applicable to the guidance in this NASA Technical Handbook.

2.1.1 The latest issuances of cited documents shall apply unless specific versions are designated.

2.1.2 Non-use of specifically designated versions shall be approved by the delegated Technical Authority.

Applicable documents may be accessed at https://standards.nasa.gov or obtained directly from the Standards Developing Body or other document distributors. When not available from these sources, information for obtaining the document is provided.

2.2 Government Documents

NASA

NPR 7150.2	NASA Software Engineering Requirements
NASA-STD-4009A	Space Telecommunications Radio System (STRS) Architecture Standard
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
NASA/TM-2009-215478	Case Study: Using the OMG SWRADIO Profile and SDR Forum Input for NASA's Space Telecommunications Radio System
NASA/TM-2011-216948	Symbol Tables and Branch Tables: Linking Applications Together
NASA/TM-2011-217266	Space Telecommunications Radio System (STRS) Compliance Testing

2.3 Non-Government Documents

Consultative Committee for Space Data Systems (CCSDS)

Red Book Specification 876.0	Spacecraft Onboard Interface Services--XML Specification for Electronic Data Sheets for Onboard Devices
Red Book Specification 876.1	Specification for Dictionary of Terms for Electronic Data Sheets for Onboard Components

International Organization for Standardization (**ISO)**/International Electrotechnical Commission **(**IEC)

ISO/IEC 9899	Information technology - Programming languages - C
ISO/IEC 14882	Information technology - Programming languages - C++

See Appendix B for reference documents.

2.4 Order of Precedence

This NASA Technical Handbook provides guidance for the rationale which underlays the requirements contained in NASA-STD-4009A but does not supersede or waive established Agency requirements/guidance found in other documentation.

3. ACRONYMS, ABBREVIATIONS, AND DEFINITIONS

3.1 Acronyms and Abbreviations

API	application program interface
BIT	built-in test
BSP	board support package
CCSDS	Consultative Committee for Space Data Systems
cFS	Core Flight System
CORBA	Common Object Request Broker Architecture
COTS	commercial off the shelf
DLL	dynamic link library
DSP	digital signal processor
EDIF	electronic design interchange format
EDS	electronic data sheet
FFRDC	Federally Funded Research and Development Center
FPGA	field programmable gate array
GPM	general-purpose processing module
GPP	general purpose processor
GRC	Glenn Research Center
GUI	graphical user interface
HAL	hardware abstraction layer
HDBK	handbook
HDL	hardware description language
HID	hardware interface description
I/O	input/output
ICD	interface control document

ID	identification, identifier
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronic Engineers
IP	Intellectual property
ISO	International Organization for Standardization
JTRS	Joint Tactical Radio System
k	kilo
N/A	not applicable
NASA	National Aeronautics and Space Administration
NPR	NASA Procedural Requirements
NRE	non-recurring engineering
OE	operating environment
OMG	Object Management Group
OO	object oriented
OS	operating system
OTA	over the air
PIM	platform-independent model
POSIX®	Portable Operating System Interface
PSE51	minimal real-time system profile 51, defined in IEEE 1003.13
PSM	platform-specific model
RF	radio frequency
RPC	remote procedure call
RPN	Reverse Polish Notation
RTEMS	Real-Time Executive for Multiprocessor Systems
RTOS	real-time operating system
SCA	Software Communications Architecture
SCaN	Space Communications and Navigation
SDR	software-defined radio
SPM	signal-processing module
STD	Standard
STRS	Space Telecommunications Radio System
SWaP	size, weight, and power
SWRADIO	software radio
UML	Unified Modeling Language
UTC	Coordinated Universal Time
V&V	verification and validation
VDD	version description document
VHDL	VHSIC hardware description language
VHSIC	very high speed integrated circuits

W3C	World Wide Web Consortium (main international standards organization for the World Wide Web (abbreviated WWW or W3).
WF	Waveform
XML	Extensible Markup Language
XSD	XML 1.0 Schema Definition
XSL	Extensible Stylesheet Language
XSLT	Extensible Stylesheet Language Transformation

3.2 Definitions

Key terms and definitions are described in section 3.2 of NASA-STD-4009A.

4. HIGH-LEVEL RATIONALE

The rationales for the STRS requirements in NASA-STD-4009A were derived from the Level 1 requirements in NASA/TM-2007-215042, STRS Architecture Goals/Objectives and Level 1 Requirements (summarized below), the restrictions of the space environment, and the use cases in NASA/TP-2008-214813, STRS Software Architecture Concepts and Analysis. The Object Management Group (OMG) Software Radio (SWRADIO) profile was considered in NASA/TM-2009-215478, Case Study: Using the OMG SWRADIO Profile and SDR Forum Input for NASA's Space Telecommunications Radio System, and the platform-independent model (PIM) was used as the starting point for the application software requirements.

STRS Goals and Objectives

4.1 Usable across most NASA mission types (scalability and flexibility).
4.2 Decrease development time and cost.
4.3 Increase reliability of software-defined radios (SDRs).
4.4 Accommodate advances in technology with minimal rework (extensibility).
4.5 Adaptable to evolving requirements (adaptability).
4.6 Enable over-the-air interoperability with existing assets (interoperability).
4.7 Leverage existing or developing standards, resources, and experience (state-of-the-art and state-of-practices).
4.8 Maintain vendor independence.
4.9 Enable waveform application portability.

STRS Level 1 Requirements

5.1 Layered Architecture.
5.2 Open Architecture.
5.3 Flexibility in Form Factor.
5.4 Remote Reconfiguration.
5.5 Remote Reprogrammability.

5.6 External Hardware Control.
5.7 Standard Spacecraft Interfaces.
5.8 Existing Waveform Support.
5.9 Multiple Waveform Support.
5.10 Simultaneous Operation of Multiple Waveforms.
5.11 Multi-Service Support.
5.12 Suitable for Any Radio Frequency Bands.
5.13 Multiple Frequency Bands
5.14 Multi-Channel Capability.
5.15 Commanded Built-In-Test and Status Reporting.
5.16 Operational Diagnostics.
5.17 Automated System Recovery/Initialization.
5.18 Navigation Support.
5.19 Network Support.
5.20 Security Compatibility.
5.21 Secure Transmission.
5.22 Processor Sharing.
5.23 Autonomous Link Optimization.

4.1 Operational Requirements

Many Level 1 requirements describe what the architecture has to allow in the way of operations. However, any mission determines which of these are to be implemented in its specific radios to support the mission's needs. In NASA-STD-4009A, requirements were written in a layered way so as to describe many architectural requirements in terms of applications, devices, services, other artifacts, and how they are used to perform the necessary functions.

The Level 1 requirements include responding to commands sent from an external source for remote reconfiguration, remote reprogrammability, processor sharing, and commanded Built-In-Test (BIT) and status reporting. These Level 1 requirements become part of the rationale for those requirements in NASA-STD-4009A pertaining to the startup and configuration of the radio as well as commanding the radio and obtaining information back about the configuration and command success.

4.2 Operating Environment (OE) Requirements

The STRS OE, as used in NASA-STD-4009A, is the software environment in which the STRS applications are executed. Because an STRS radio is really a computer, it has an operating system (OS) usually created separately from the STRS infrastructure. Using an OS promotes the STRS goals and objectives for flexibility, decreasing development time and cost, and increasing the reliability of SDRs, by leveraging existing standards, resources, and experience. A real-time operating system (RTOS) is likely to be used to meet timing deadlines and to support other operations, even though most of the real-time capability currently resides in field programmable gate arrays (FPGAs). Furthermore, an OS will need real-time capability if the general purpose processors (GPPs) are faster and assume more of the telecommunication functions or if the

mission's telemetry requirements are stringent enough to warrant using an RTOS to be able to achieve processing constraints.

The Portable Operating System Interface (POSIX®) is a standard that is used by many OSs. Therefore, POSIX® was chosen to implement certain functions missing from the list of STRS Application-provided methods to minimize duplication of methods between STRS and POSIX®.

4.3 Documentation Requirements

NASA-STD-4009A requires specific information to be delivered with an STRS-compliant platform and application. This information is requested to support the goals and objectives for extensibility, adaptability, portability, reusability, and vendor independence. The specific mission procuring the platform and/or application will require additional documentation to integrate and operate the platform and/or application with the rest of the system for the mission.

The hardware abstraction layer (HAL) documentation is required by STRS to allow, for example, the independence of the providers of the infrastructure and the FPGA. The basic goals and objectives include accommodating advances in technology with minimal rework (extensibility) and maintaining vendor independence. The STRS infrastructure has access to the HAL, and the HAL is specific to a platform. Since STRS is designed for the applications to be as portable as possible, STRS hides the HAL from the STRS application using a bridge pattern so that an STRS application can use a standardized application program interface (API) to interact with any specialized hardware.

4.4 Source Code Requirements

Because of size, weight, and power (SWaP) restrictions for space, many of the requirements in the Department of Defense's Software Communications Architecture (SCA) version 2.2 were eliminated for STRS. These include eliminating the requirement for Common Object Request Broker Architecture (CORBA) and the onboard parsing of the Extensible Markup Language (XML). Although object-oriented Unified Modeling Language (UML) diagrams were used for parts of the description to clarify relationships, they did not become part of the requirements. As described in the Case Study, STRS was able to use almost the same PIM as the OMG's SWRADIO to come up with a very different implementation.

Because the STRS architecture is needed to support a C language interface to minimize SWaP, a pure object-oriented approach was unsatisfactory. To encapsulate functionality in a consistent manner, APIs were defined and the corresponding #include files were required to constrain the method signatures appropriately. A subset of the infrastructure APIs was required to call the appropriate application API. Because of the C language interface, the methods were differentiated so that the STRS infrastructure APIs had a different naming convention and two additional arguments:

a. The *fromWF* argument in many of the STRS infrastructure APIs is the handle identification (ID) used to indicate who the caller or source is.

b. The *toWF* argument in many of the STRS infrastructures APIs is the handle ID used to indicate who the responder or destination is. The STRS infrastructure uses the *toWF* to determine which application, device, file, or queue is to be used to perform any further processing. Then the infrastructure-provided method usually calls the corresponding Application-provided method.

A handle ID is used to control access to applications, devices, and so forth. The *fromWF* and *toWF* may be used by the infrastructure to validate whether the method is allowed to be called and to keep track of the history of the method call for error processing. They are used by the infrastructure when creating log messages to indicate the source of an error.

It is up to the infrastructure provider whether the handle ID is an index into a table, or an address of a structure, or a hash value used to look up the information that the infrastructure uses to access the application, device, file, or queue. The only restriction imposed by STRS is that a negative value indicates an error. It is the responsibility of the infrastructure to keep any additional information needed to intelligently populate the error logs. Each application is informed of its own handle ID and handle name using APP_Instance. The application uses its own handle ID as the *fromWF* argument to most STRS infrastructure-provided methods. There should be a handle ID for the infrastructure (or portions thereof) to indicate when the infrastructure is the source for error messages.

After considering the functionality required by the use cases and OMG SWRADIO profile, there needed to be APIs to standardize additional portions of the software. Therefore, APIs were created for devices, message queues, files, and timing. Some others were left to POSIX® to implement.

4.5 Configuration File or Script File Recommendation

Configuration files or script files are recommended as a self-documenting way for the STRS infrastructure to start STRS applications into a known state to support the Level 1 requirements for remote partial configuration, reconfigurability, adaptability, automated system recovery and initialization, and multiple waveform support. The basic requirement derives from the need for each component to start in a known state. Requirement STRS-1 produces a known state for the platform. Using application configuration files or script files produces a known state for the applications. The infrastructure configuration files are suggested to allow for evolution of the hardware and software in a standard way.

Using an application configuration file addresses the variability regarding what resources may be needed by the application, what variables need to be defined, and what state changes need to be made. There is not necessarily a one-to-one correspondence between an application and a loadable file. A loadable file can contain parts of multiple applications; likewise, an application can span multiple loadable images. In a common case, a single application has both a GPP part and an FPGA part. Another case occurs for the OS-like Real-Time Executive for Multiprocessor Systems (RTEMS), a free open-source RTOS designed for embedded systems, which does not support dynamic loading. In that case, there is no need to include separate information on the

GPP part of an application, since the build process does not make a loadable image for the application alone, separate from the infrastructure.

The application configuration files allow for the same application to be configured for different situations. For example, the environment may vary over the life of the radio and parameters may need to be adjusted accordingly. Another example might be allowing the original frequencies defined in a configuration file to be updated to avoid interference. It is less risky to send a short data file to the radio than to send a full software load. A current example for one of the SCaN Testbed SDRs is that the software and/or configurable hardware design has to remain stable at a certain point in the development process so that code review and test can be completed. However, there are parameters that are environmentally sensitive and can only be determined by testing under environmental conditions that simulate the conditions of space. These parameters are determined and entered into the application configuration file. A new application configuration file is then added to the SDR, but the software and configurable hardware design used for the code review and test remain unchanged.

When the waveform application is started, the state, resources, and the configurable parameters can be specified in a configuration file or script so that any reinitialization is predictable. For example, if the radio needs to cycle power to correct some glitch, it should be able to do so and restart the application without intervention. The project manager may require the inclusion of "initial or default values for all distinct operationally configurable parameters." The "operationally configurable parameters" were those that could be configured using the STRS_Configure/APP_Configure commands instead of using subordinate parameters that do not have to be configured separately or using merely queryable parameters. For example, if a data item can be initialized in multiple ways such as having parameters for both frequency and wavelength, only one would need to be configured. Also, a parameter could be queryable but not configurable (e.g., temperature, location, and power consumption).

When the project manager specifies a requirement for a configuration file, XML is recommended as a good starting point. The reasons for using XML for the predeployed application configuration files are as follows:

a. Using XML allows the data to have a standardized format that is easily created, read, and used by multiple entities.

b. XML is easy to learn, does not require much overhead, and is a World Wide Web Consortium (W3C) free and open standard.

c. Using XML saves time and money since tools already exist for displaying, editing, validating, parsing, and other functions.

d. Using XML allows the data to be self-documenting.

e. An exact format for the predeployed configuration files is not specified because the infrastructure for each vendor may need different information to start an application. The data in the application configuration files may not always be just name/value pairs.

f. Using XML allows for hierarchical as well as sequential data. Using hierarchical data allows for greater complexity. Using XML does not hinder the data from being used sequentially or in a specific order.

g. XML is easily transformed to more compact forms such that only the relevant subset appears in the deployed configuration file. The minimum data could be identification, any resources loaded, any parameters configured, and ending state information.

h. Using XML allows the STRS integrators to be able to verify that the data they enter meet some specified criteria so that the creation of the deployed configuration file will work properly.

i. STRS began as an approach to adapt SCA for NASA space applications. A waveform application was implemented using SCA and from that, it was apparent that having CORBA and an XML parser in the radio added quite a lot to the complexity, size, and weight. STRS eliminated the need for CORBA, dynamic features, and an XML parser in the OE. So, the best resolution was having XML to start with and a processed file to deploy on the radio.

j. The SCA's properties files in XML could be used for STRS with minimal changes. An Extensible Stylesheet Language Transformation (XSLT) could be written to transform an SCA properties file into any deployed format.

k. Using XML allows validation of input values. Part of the compliance testing is to verify that the configuration file follows the format specified in the schema. Using an XML schema saves NASA from having to keep a different validation tool for each vendor.

l. Using XML allows data to be labeled with tags and attribute names so that the data are more easily validated and changed. The minimum was specified for the format of the application configuration files so that each vendor could use the format appropriate to that vendor's implementation. For example, here are some alternatives for a "wait" command in XML:

(1) <command>wait 100</command>
(2) <wait>100</wait>
(3) <wait delay="100"/>
(4) <wait delay="100" units="microseconds"/>
(5) <wait> <delay>100</delay> <units>microseconds</units> </wait>

Validation is difficult for alternative (1) using a schema. When the data "100" is separately identified, as in alternatives (2)-(5), it is easier to validate using a schema. When "100" is labeled as a delay, it is easier for multiple entities to identify and modify. Alternative (3) is not really better than (2) because the word delay does not add to the definition of the number significantly but adding the units does. When the "100" is labeled as a delay in microseconds, as in alternatives (4)-(5), it is even easier for multiple entities to identify and modify. Both (4) and (5) contain the same explanatory information and may be clearly checked by a schema. Which one may be

considered optimal depends on the ability to transform the data into a deployed format.

XML 1.0 is recommended because the http://www.w3.org/ standards state that even though XML 1.1 is the current version, “You are encouraged to create or generate XML 1.0 documents if you do not need the new features in XML 1.1.” Furthermore, an error was obtained when using XMLSpy, a commercial-off-the-shelf (COTS) software product for testing with XML 1.1.

To avoid the drawback of requiring an XML parser to be part of the radio, the STRS allows the configuration files to be preprocessed into a simpler form. Although XSLT is suggested in NASA-STD-4009A as a simple mechanism for this transformation, it is not required. Alternatively, the preprocessing could be handled by a program or script. Although a manual process using a text editor is not impossible, an automated process is preferred.

The following state diagram, figure 1, STRS Application Recommended State Diagram, shows that an STRS application can have various states during execution. The files for the STRS application are to be accessible before execution can begin.

- STRS_InstantiateApp causes the deployed configuration file to be parsed and APP_Instance or the constructor to be called such that the STRS application starts in the INSTANTIATED state, but it may be transitioned to another state if specified in the STRS application configuration file.

- STRS_Initialize calls APP_Initialize on the appropriate STRS application.

- APP_Initialize transitions the STRS application to the STOPPED state upon successful completion.

- STRS_Start calls APP_Start on the appropriate STRS application.

- APP_Start transitions the STRS application from the STOPPED state to the RUNNING state upon successful completion.

- STRS_Stop calls APP_Stop on the appropriate STRS application.

- APP_Stop transitions the STRS application from the RUNNING state to the STOPPED state upon successful completion.

- When either APP_RunTest or APP_GroundTest is called, the application may be transitioned from the STOPPED state to a TESTING state, if necessary.

- STRS_ReleaseObject calls APP_ReleaseObject on the appropriate STRS application.

- The FAULT state may be set by the STRS application or detected by the fault monitoring and recovery functions, but any recovery is managed by the STRS infrastructure or by an external system.

The STRS application internal states shown in figure 1 are suggested. The STRS application developer may define and use any additional internal states that the STRS application developer sees fit. The infrastructure may use any additional states that are deemed necessary.

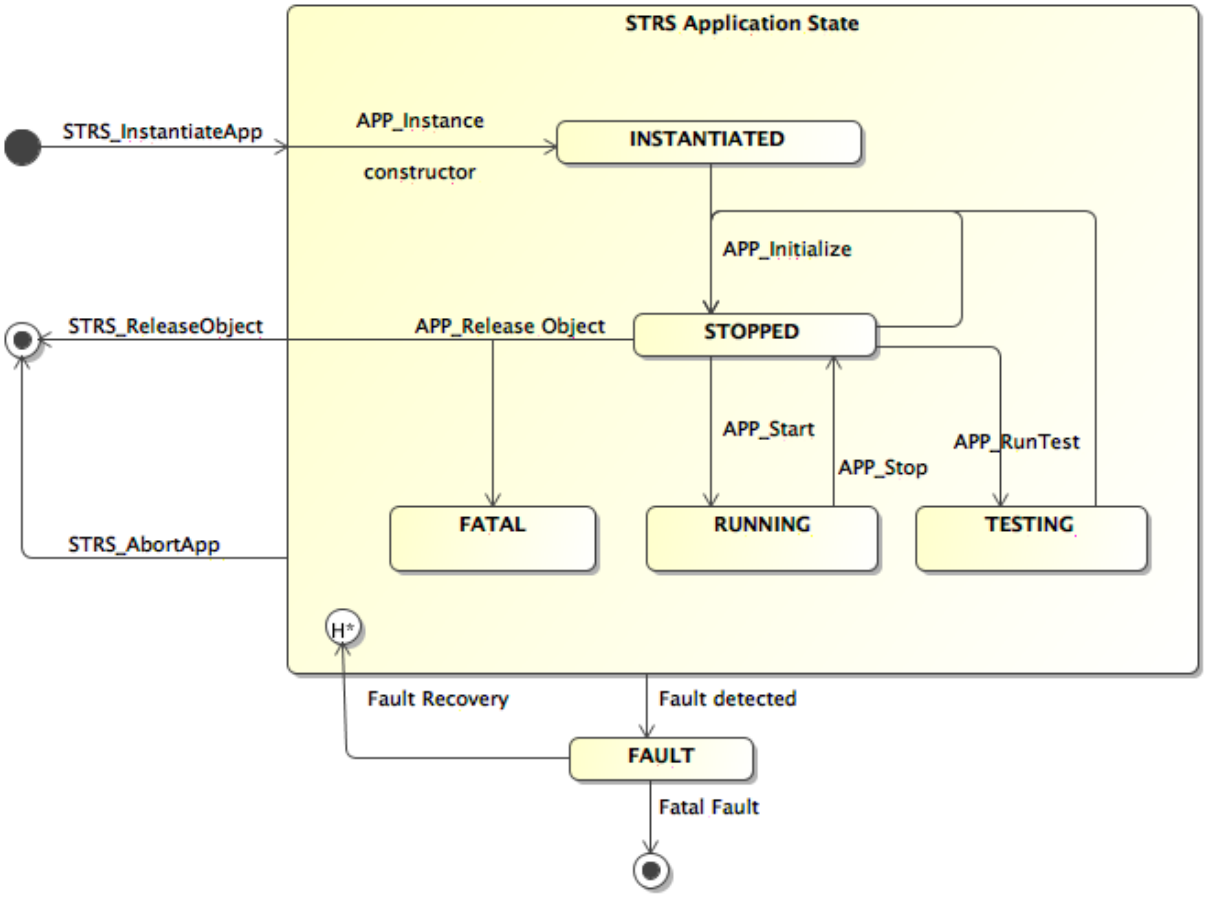

Figure 1—STRS Application Recommended State Diagram

4.6 Roles and Responsibilities

For STRS, roles are specified as abstractions for the responsible organizations. The roles and corresponding organizations are expected to change at different stages of the radio's life cycle. For example, a developer or provider of some component may act as an STRS integrator for that component and other components at a subsequent stage of production. Then, that STRS integrator may act as a provider for the next stage. NASA's goals are to promote vendor independence, scalability, flexibility, and extensibility while specifying the smallest number of clearly defined roles possible.

The basic roles that NASA-STD-4009A defines are the STRS platform provider, who delivers a platform upon which STRS applications can be executed, an STRS application developer who provides the desired functionality in the form of an STRS application, and an STRS integrator who is responsible for integrating the parts to work together. The STRS platform provider could subcontract for hardware and software, but the responsibility for coordination, integration, and delivery of the infrastructure and related artifacts would reside in one STRS platform provider organization.

The STRS platform provider would usually act as STRS application developer and STRS integrator for at least a sample application. The roles and associated products are depicted in figure 2, Roles and Products.

The roles could have been broken down differently, allowing for various combinations of providers and integrators that could be very complex. Some suggested roles were as follows:

a. Application developer or provider.
b. Application integrator (OE + applications).
c. Configurable hardware design provider.
d. HAL/board support package (BSP)/drivers provider.
e. Hardware integrator.
f. Hardware parts supplier.
g. Infrastructure provider.
h. Kernel integrator (OS + POSIX® + HAL).
i. OE integrator (OS + POSIX® + HAL + infrastructure).
j. OE provider (OS + POSIX® + HAL + infrastructure).
k. Operator.
l. OS provider.
m. Platform integrator.
n. Platform provider.
o. POSIX® provider.
p. Radio integrator (OE + applications).
q. Radio operator (concerned with the mechanics of commanding the radio).
r. Spacecraft operator (concerned with the functionality of the radio in the larger sense of including ground operations, experimenters, i.e., consumers of the data flowing through the radio).
s. System integrator (the entity that puts the radio into a system).

Some of these roles are duplicates or overlap one another. There were multiple interpretations for some of the roles, causing confusion. Therefore, the roles were simplified. Although a few operator roles were suggested, the operator roles have no STRS requirements, so these roles were not included. The operator roles are required for specific missions or projects rather than for the STRS architecture. There are no STRS requirements for specific external commands and how the commands and data get to the STRS radio. There are no STRS requirements for a specific process for an operator to turn the radio on and off, send configuration commands, consume and source data, deal with configuration management of software uploads, and other functions. There are no STRS requirements for the radio link parameters and for the actions of the experimenters

and consumers of the data flowing through the radio. These types of requirements are mission-specific and have to be included in the requirements for the particular mission or project, which are in addition to NASA-STD-4009A requirements.

Document preparer roles and reviewer roles are not included, because STRS has no requirements concerning the process by which the documents are generated. There will be mission or project requirements for additional roles (including stakeholders) not mentioned here, for which STRS has no requirements.

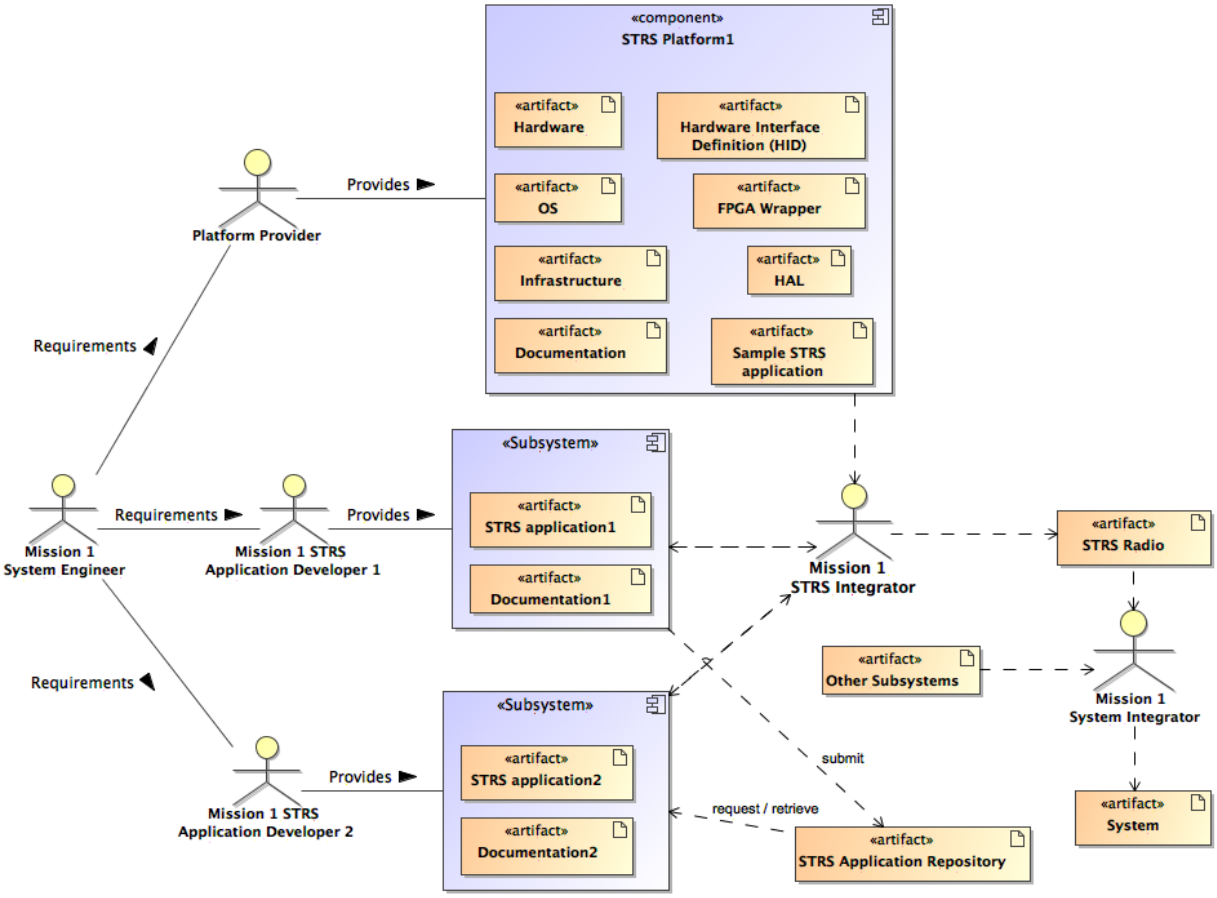

Figure 2—Roles and Products

5. HOW TO USE STRS APIs

This section contains recommendations and general information about how some operations would usually be performed using the STRS APIs.

5.1 How to Associate an FPGA with an STRS Application

For an FPGA, there should be a corresponding STRS Device (GPP code) to control the FPGA and transfer data. The handle name of the STRS Device for each FPGA may be named differently, especially on different platforms. To make it easy for the STRS application to access the appropriate FPGA with maximum portability, the handle name of the STRS Device that corresponds to the appropriate FPGA should be a configurable attribute for the STRS application. The STRS application would use the attribute supplying the handle name to obtain the handle ID using STRS_HandleRequest. The application would use the resulting handle ID of the STRS Device to invoke the STRS Infrastructure-Provided Application Control API methods (NASA-STD-4009A, section 7.3.2) and STRS Infrastructure-Provided Device Control API methods (NASA-STD-4009A, section 7.3.6) to interact with the FPGA. A likely implementation would have any methods in the STRS Infrastructure-Provided Device Control API invoke the corresponding method in an STRS Device-Provided Device Control API. A sample of an STRS Device-provided Device Control API is shown in Figure 5, STRS Application/Device Structure, as the Device API.

For example, there might be FPGA1 and FPGA2 available. If WF_1 uses FPGA2, a configuration file for WF_1 could contain a name/value pair to associate the name useFPGA with FPGA2. Let the function getValue obtain the value corresponding to the given name. Then, the STRS application could access the correct FPGA, as follows:

```
STRS_HandleID fromID = APP_GetHandleID();
char* fpgaName = getValue("useFPGA");
STRS_HandleID fpgaID = STRS_HandleRequest(fromID, fpgaName);
STRS_Result rtn = STRS_ValidateHandleID(fpgaID);
If ( ! STRS_IsOK( rtn )) {
      STRS_Message msg = "Handle ID error for useFPGA.";
      STRS_HandleID errQ = STRS_GetErrorQueue(rtn);
      STRS_Log (fromID, errQ, msg, (STRS_Buffer_Size) sizeof(msg));
}
```

5.2 How to Load an FPGA

An FPGA may be loaded directly by the infrastructure when it parses the configuration file, if supported, or may be loaded by an STRS_DeviceLoad call from the application GPP code. If the latter method is used, the name of the bitstream file should also be a configurable attribute set with the APP_Configure method. The STRS_HandleRequest method should be called to obtain the handle ID for the FPGA Device, and then the STRS_DeviceLoad method should be called for the FPGA to load the bitstream file. These STRS infrastructure calls may be performed in the APP_Configure directly or in the APP_Start method, as appropriate.

5.3 How to Set Attributes

An FPGA may be configured directly by the infrastructure when it parses the configuration file or by an STRS_Configure call to the STRS Device for the FPGA call from the application GPP code. If the latter method is used, the handle name of the FPGA Device should also be a configurable attribute set with the APP_Configure method. The STRS_HandleRequest method should be called to obtain the handle ID for the FPGA Device, and then the STRS_Configure method should be called for the FPGA Device to configure the FPGA. These STRS infrastructure calls may be performed in the APP_Configure directly or in the APP_Start method, as appropriate.

5.4 How to Get Attributes

Not all specialized hardware can be interrogated for its configuration; however, the infrastructure or the application may maintain any configuration data needed. The application has to implement APP_Query and, if appropriate, it should call the STRS_HandleRequest to obtain the handle ID for the FPGA followed by an STRS_Query call to the STRS FPGA Device to obtain the configuration data from the FPGA.

5.5 How to Push Packets

To push packets from an application, to a device, queue, file, or another application, STRS_Write is used. For generating packets in the same application used to send the packets, APP_Write may be used directly. If an application acts as a sink of packets pushed, it has to implement APP_Write, #include "STRS_Sink.h"; and if C++, the class has to implement STRS_Sink.

5.6 How to Pull Packets

To pull packets from a device, queue, file, or another application, STRS_Read is used. To pull packets from another module in the same application, APP_Read may be used directly. If an application acts as a source of packets pulled, it has to implement APP_Read, #include "STRS_Source.h", and, if C++, the class has to implement STRS_Source.

5.7 How to Process Errors

When a call to an STRS method is made, a variable of type STRS_Result is usually returned. To ensure consistent testing for errors, where an error is usually a negative value, STRS_IsOK tests that variable of type STRS_Result for errors, and returns a true or false boolean variable. The value returned from STRS_IsOK is true when there is no error and false when there is an error so that appropriate action may be taken.

When an error is detected in the operation of the application, STRS_Log should be invoked using an error queue handle ID (STRS_FATAL_QUEUE, STRS_ERROR_QUEUE, or STRS_WARNING_QUEUE), and a descriptive message. The error queue handle ID can be

determined using STRS_GetErrorQueue with the error return value as an argument. The error queues are monitored and passed to the infrastructure for further action.

The STRS methods use the error returns rather than using variable errno to indicate an error. STRS policy on errno is that it is undefined outside of the application methods. You cannot rely on it to indicate the particular error because it is not reset before system calls and may be reset to a different error later, before it is tested. It reduces portability by allowing different values on different operating systems with different compilers.

5.8 How to Make Multiple Instances of an Application

To create multiple instances of an application, be sure that the application is reentrant and that all pertinent data are configurable. Two configuration or script files may have to be created, if any of the initial data is different. A different handle name is specified for each instance. When using C language applications, there may be method name duplication, which is discussed further in section 6.10, C Language Naming Duplication.

5.9 How to Map Memory Locations

STRS Devices are allowed to use memory mapped locations in which storing/retrieving an item in shared memory automatically pushes/pulls the item to/from the specialized hardware. STRS Devices do not have to be portable, so non-standard methods may be encapsulated within an STRS Device. The STRS applications should not use the shared memory locations directly to communicate with the specialized hardware because that is not portable and violates the spirit of STRS. The addresses for the specialized hardware should not be defined in the application or its configuration file. Hard-coding of memory locations in an STRS application or STRS Device is strongly discouraged because hard-coding would limit the independence of the software and configurable hardware design and may cause problems with verification and validation if any FPGA code is changed that would affect that location.

There is no requirement that an STRS Device use configuration or script files containing location information, but it is strongly encouraged to be configured rather than hard-coded. Configuration data are usually specified in a configuration file where the OE parses the configuration file to call STRS_Configure and that calls APP_Configure in the STRS Device to accept the configuration data. However, there is nothing in the standard that restricts the OE from calling other method(s) such as DEV_SetMemoryMap(map) to specify more complicated configuration data. Similarly, a script file may be executed to call STRS_Configure and related commands.

The shared memory locations should be specified in the configuration or script files for the appropriate STRS Device. In the example of a configuration file shown in Appendix A for MEMORYMAP and MAPVALUE, there is a base name, associated relative location, offset, size, and access. These are illustrated in figure 3, Memory Map. The location for an individual item is specified relative to the base name in addressable storage units. The location may also have a bit offset and bit length. Then, when the configurable item is modified, the mapped

location is used. In the example of an STRS Device shown in Figure 5, the OE can use the DEV_SetMemoryMap(map) method to configure the mapping in the STRS Device.

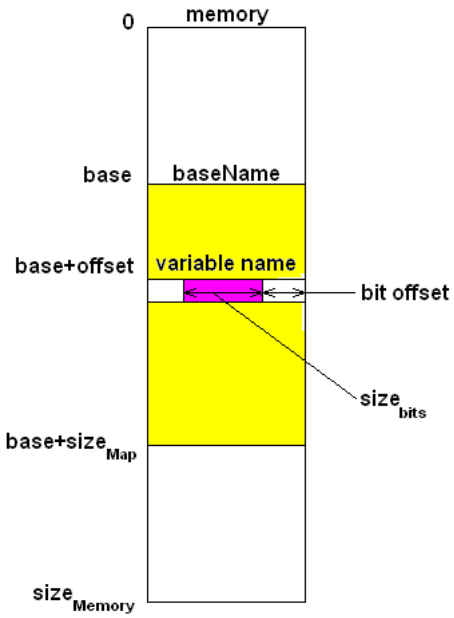

Figure 3—Memory Map

5.10 When to Use STRS_Log and STRS_Write

The two STRS infrastructure methods, STRS_Log and STRS_Write, have similar functionality except that STRS_Log adds a time stamp and possibly other identifying information. These methods should never be mixed for a given target. An STRS application developer should only write to the error queues using STRS_Log because the errors need to be identified further (STRS-54, STRS-55, STRS-56) and never with STRS_Write. Similarly, an STRS application developer should only write to the telemetry queues using STRS_Log because the telemetry data need to be identified further (STRS-57) and never using STRS_Write. Furthermore, the error queues are monitored for faults. An STRS application developer should only use STRS_Write to write buffered data to another application, service, device, file, or queue that does not require additional information added.

5.11 Difference Between Run Test and Ground Test

A run test is invoked using STRS_RunTest and implemented by APP_RunTest. A ground test is invoked using STRS_GroundTest and implemented by APP_GroundTest. A run test is invoked before or after deployment to determine whether the component is performing correctly. A ground test is generally invoked before deployment to perform unit testing and calibration. The ground tests help to automate and evaluate those tests. The term ground test was originally used to indicate testing for a satellite system, which is performed on the ground before launch.

Ground test may be invalid after deployment and indicates that such tests are normally completed before deployment and are not repeated thereafter. If allowed by the project and the ground tests will not be repeated after deployment, then the ground test code may be removed prior to deployment. The run tests and ground tests were separated because NASA generally requires significant testing prior to deployment; for example, vibration testing, environmental testing, radiation testing, etc.

5.12 When to Use Start/Stop, Load/Unload, and Open/Close

The commands STRS_Start, STRS_DeviceLoad, STRS_DeviceOpen are specified in NASA-STD-4009A along with the reverse commands, STRS_Stop, STRS_DeviceUnload, and STRS_DeviceClose. The following describes the interaction of these commands under various common circumstances.

Initialize (STRS_Initialize) is used while the application is in the STOPPED mode to set the application to a known initial condition. The application may be configured before and/or after initialize. Start (STRS_Start) is used to begin normal processing and change the state to RUNNING. If any part of the application is in specialized hardware, that portion needs to be loaded before starting. To load (STRS_Load) an STRS Device, the Device has to be opened (STRS_Open) first. It is suggested that any part executing in specialized hardware not begin execution upon being loaded but rather during the start process. Similarly, it is suggested that stopping execution (STRS_Stop) does not require any specialized hardware to be unloaded. Therefore, greater control is given to the application software for processing commands to start and stop waveform application operation in order to take advantage of windows of opportunity for execution as well as to promote consistency in control of the radio. Only certain allowed items may be configured after starting.

It is suggested that the STRS OE use configuration file(s) to start-up an application to a known initial state. STRS encourages that changeable data be specified in configuration files, rather than coding the data as constants within the application or device, so that greater portability and ease of modification is achieved. The STRS OE may process a configuration file to instantiate, open and load the device, initialize, configure, and start the application, or use any subset of these as determined by the project/mission and STRS platform provider.

As an example, the following use case is written for a waveform application using specialized hardware to send signals over the air to another radio assuming that the specialized hardware device has already been instantiated and initialized by the OE:

1. Radio receives a command that a new waveform application is needed. This may be multiple commands received or one command that invokes a series of operations. In either case, those operations follow.
2. OE checks for availability of the application and memory to instantiate it.
3. OE instantiates application (STRS_InstantiateApp).
4. OE initializes application (STRS_Initialize).
5. OE opens specialized hardware device (STRS_DeviceOpen).

6. OE loads specialized hardware device (STRS_DeviceLoad).
7. OE configures application (STRS_Configure).
8. OE starts application (STRS_Start).

The use case for the reverse process is as follows:

1. The radio receives a signal to stop and remove the application. This may be multiple commands received or one command that invokes a series of operations. In either case, those operations follow.
2. OE stops the application (STRS_Stop).
3. OE unloads the specialized hardware device (STRS_DeviceUnload).
4. OE closes the specialized hardware device (STRS_DeviceClose).
5. OE releases resources for application (STRS_ReleaseResources).

If there is no specialized hardware device, the steps pertaining to such a device may be eliminated. If the application merely performs calculation, start may mean perform the calculation and, before each calculation, the data is reconfigured. Alternately, start may mean ready to perform the calculation and start invokes a thread that loops waiting for new data so that each time new data is obtained, the computation is performed. Similarly, for a waveform application, start may mean to tell the specialized hardware device to begin processing signals or alternately, start may invoke a thread to perform the communication functions. A separate thread is used so that other commands may be processed independently.

5.13 How to Read and Write Data

An application may read data using STRS_Read and write data using STRS_Write. A description of content, format, and usage of buffers is provided as part of the user documentation. As Consultative Committee for Space Data Systems (CCSDS) Electronic Data Sheets mature, these should be considered as precise machine-readable descriptions to facilitate reading and writing data. Currently, CCSDS Electronic Data Sheets are Red Book specifications 876.0 and 876.1.

It is suggested that CCSDS be used to document the format of buffers. CCSDS book 876.0 specifies "Electronic Data Sheets" (EDS), which is an XML schema for describing data exchange between system components. It provides a means to specify the exact format of the data, binary or otherwise, the types of interfaces provided by a component, as well as any handshaking or protocol-level requirements of an interface. The objective is to specify these details in a machine-readable language, and with sufficient detail such that it eliminates the need for a separate interface control document (ICD). The system components described can be physical hardware devices, where the device manufacturer would author the EDS, or software components where EDS can serve as a common interface description between the sender and receiver.

The STRS architecture does not dictate a specific data format for all APIs, leaving this decision up to application/waveform developers. By standardizing the description of the data format using EDS, flexibility is retained while still providing a compatibility point. Because the electronic data sheets are implemented in a machine-readable format, parts of the data exchange between components can be machine assisted, thereby easing the efforts required to port applications between different systems. Using the EDS, any other independently developed system or component that also implements the CCSDS electronic data sheet technology can exchange data with the component.

The core Flight System (cFS) is a platform independent reusable software framework and set of reusable software applications that is reused on NASA flight projects and/or embedded software systems at a significant cost savings. The cFS contains an implementation of the CCSDS electronic data sheets. The cFS-based STRS OE utilizes this technology to describe the sink and source interfaces as well as the various configurable properties of STRS applications, which aids portability to other cFS installations or other independently developed systems implementing EDS.

5.14 How to Add Asynchronous Publisher/Subscriber Functionality

A queue can be used to disassociate a publisher directly from a subscriber. The Pub/Sub implementation is synchronous. To get the asynchronous publisher/subscriber effect, the publisher writes to a queue and, in a different thread, a copy service reads from the queue and writes the data to a Pub/Sub to which the subscriber has registered. The copy service is to be configured knowing a queue source, a Pub/Sub sink, and the maximum number of bytes. Functionally, the copy service merely obtains the source and sink handle IDs and uses them to read from the source and write to the sink.

A copy service may be useful to copy other sources to other sinks. It may need additional options to do so, especially concerning initialization, finalization, and timing considerations. These will have to be handled on a case-by-case basis.

6. QUESTIONS AND ANSWERS

The following questions have been asked by implementers of STRS-compliant platforms and applications and are included to provide additional insight for readers who might have similar questions.

6.1 Fault State and Use of the ERROR, WARNING, and FATAL Queues

NASA-STD-4009A does not specify how the fault state is set or detected. The fault state may be determined in a number of ways as specified by the mission or project. When STRS_Log sends a message to the error queue or fatal queue, it is assumed that there is an error or fatal error in that component and that the fault state is set accordingly. The fault state could also be set when a Health Manager or Watchdog Timer detects a problem (neither of which is required). The fault state could also be detected when the telemetry returns improper values. The fault state as shown

in the state diagram (figure 1) implies that the radio detects and possibly recovers; however, it could be designed so that the fault state is kept by the flight computer or equivalent.

The use of the ERROR, WARNING, and FATAL queues are expected to be defined more closely by the mission or project. The three relevant types of queues are as follows:

a. The STRS_FATAL_QUEUE is the queue used when an STRS_FATAL error is encountered. STRS_FATAL_QUEUE denotes the queue for a unrecoverable error in an attempt to capture information about the situation in a logging trail used to reconstruct the original cause of the error. Furthermore, sending a message to the STRS_FATAL_QUEUE is one way of initiating an orderly shutdown and reboot of the radio to a known state. The processing for a fatal error could imply turning off the heartbeat; that is, rebooting the radio and, if that does not work, reloading the software and/or configurable hardware design. It could imply that additional diagnostic tests need to be run. It is up to the mission to define whether there are alternative ways of rebooting under different circumstances such as after trying three times. It may make a difference if the problem is overheating or if a bit has been changed in the radio so that it does not work properly.

b. STRS_ERROR_QUEUE denotes the queue for a recoverable error. The most likely reason is an invalid set of configuration parameters. The recovery would be to get a valid set of configuration parameters.

c. STRS_WARNING_QUEUE denotes the queue for a recoverable error that has little or no effect on the operation of the radio. The most likely reasons are trying to run a test in a state for which the test is not allowed, trying to configure or query a parameter when the value is not available in that state, or trying to run APP_Start when the application is already started.

6.2 Pub/Sub Messaging and Queues Need Clarification

In NASA-STD-4009A, a Pub/Sub is distinguished from a message queue. In a Pub/Sub, messages written to the message passing facility by one application are delivered to all subscribers of that publisher. The STRS does not require implementing Pub/Sub using the observer/publish-subscribe design pattern where the class inherits a notify method. The STRS is designed to work in C without inheritance, but the idea is that the publisher does not know the identity of the subscriber such that one or more applications or devices can funnel data to one or more different applications, devices, files, or queues. Figure 4, Sample Publisher-Subscriber Sequence Diagram, is just one possibility for a sequence diagram showing the creation and possible use of a messaging queue using one form of the publisher-subscriber paradigm.

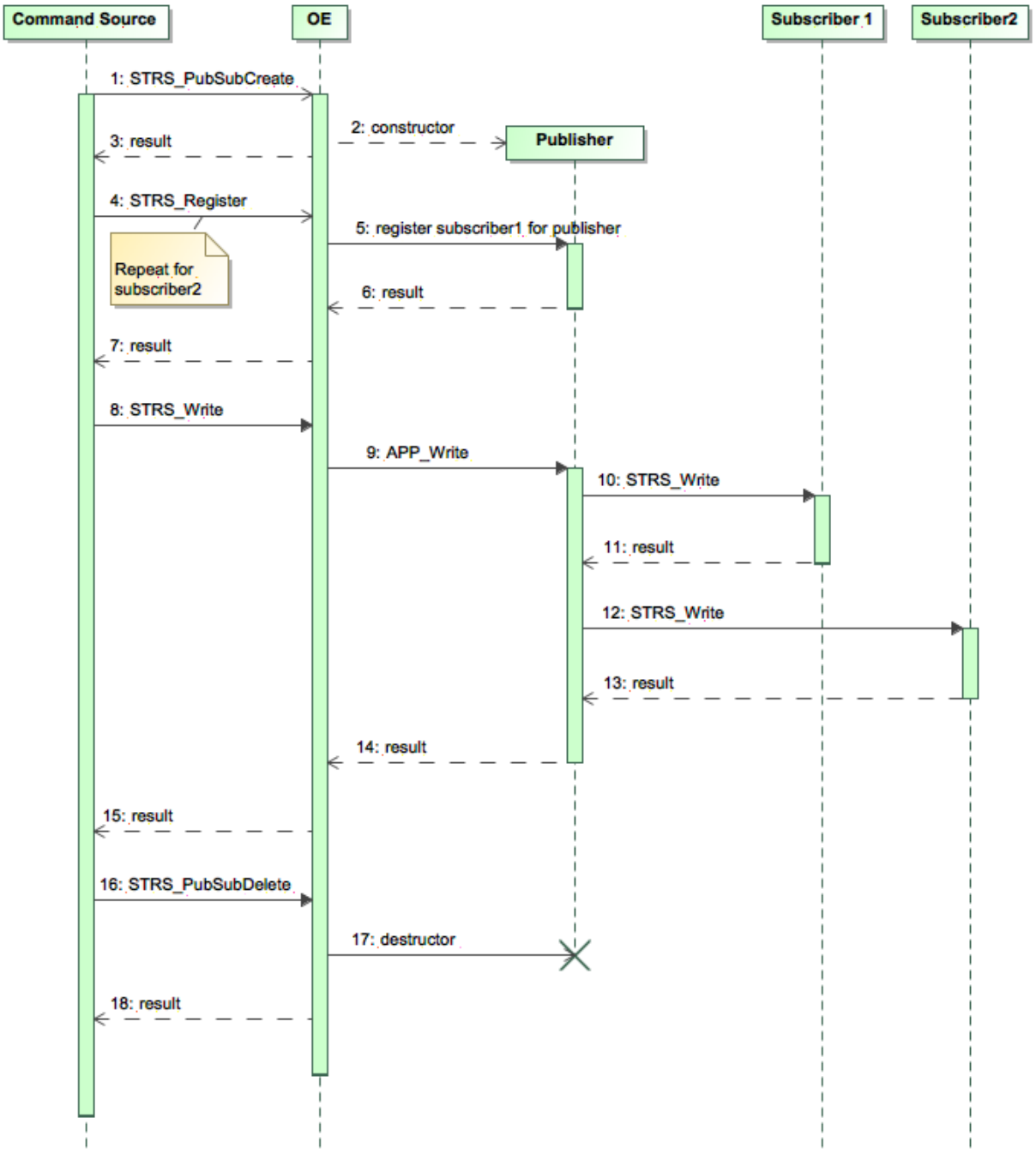

Figure 4—Sample Publisher-Subscriber Sequence Diagram

Detecting circularity and duplication is difficult with just the sequence diagram shown without adding additional methods. Circularity is where the message published eventually ends up back at the original publisher and is sent again in an infinite loop. Duplication is where the message published ends up at the same destination twice.

There is a problem of notification when it is a message queue. STRS_Write will put the message on the queue, but there is no standard way of determining when there is something waiting on the queue; that is, when does STRS_Read obtain it from the queue and when does the message get deleted from the queue? Also, can the queue fill up so that further messages are rejected? The resolution to the message-queuing behavior is not included in the current version of NASA-STD-4009A but has to be covered by the specific design for the mission or project.

6.3 What is an STRS Device?

An STRS Device is software that responds to both STRS Infrastructure-provided Application Control methods and STRS Infrastructure Device Control methods. An STRS Device is used to separate an abstraction in the form of its interface from its underlying implementation. An STRS Device is a bridge between a waveform application and the specialized hardware, used to insulate the waveform application developer from knowing how the data gets to its final destination. This encourages encapsulation of non-portable functionality. STRS Devices do not have to be portable. However, it will be advantageous to follow the STRS Device-Provided Device Control API wherever possible to maximize consistency and portability. The functionality of the STRS Device will need to be ported to each successive radio while allowing an application to access the STRS Device methods in a consistent way that makes the STRS applications more portable and more understandable.

If a waveform application intends to transfer a data value from/to the specialized hardware, the waveform application would contain the data value and the corresponding STRS Device would be used to transfer the data value from/to the specific location in the specialized hardware. The STRS Device could use memory mapping or POSIX® or HAL, whichever is the documented way to get/set data in the specialized hardware while keeping the waveform application portable.

An STRS application is expected to use an STRS Device method to transfer data to and from the physical device. It was expected that the STRS Device configure/query methods could be used to get or set values, but there was a complaint that there were problems with performance, porting from legacy applications, and scheduling. Any exceptions to the standard are handled individually.

a. An STRS Device can act as a sink by implementing APP_Write in the STRS Device. An STRS_Write/APP_Write is used to send data to a buffer in the specialized hardware. As an alternative, the STRS Device's APP_Configure could be used.

b. An STRS Device can act as a source by implementing APP_Read in the STRS Device. An STRS_Read/APP_Read is used to retrieve data from a buffer in the specialized hardware. As an alternative, the STRS Device's APP_Query could be used.

c. An STRS_Configure/APP_Configure is used to send the value of a name/value pair to the corresponding location in the specialized hardware.

d. An STRS_Query/APP_Query is used to retrieve the value of a name/value pair from the corresponding location in the specialized hardware.

When an FPGA is implemented as an STRS Device, the FPGA may be loaded, configured, started, stopped, unloaded, and so forth, using the corresponding STRS Infrastructure Device Control API. There is no requirement that an STRS Device actually exists as separate software or hardware item. There is no requirement that the Device API is implemented as shown in figure 5. They are required when portability and reusability are desired as measured by their inclusion into the STRS Application Repository.

Many SDRs use memory mapped locations in which storing/retrieving an item in memory automatically pushes/pulls the item to/from the specialized hardware (see section 5.9). Other SDRs use special APIs that comprise the HAL to send/retrieve the item to/from a specific address in the specialized hardware. Still others use POSIX®. Having the STRS Device as a portable interface for data transfer allows the HAL methods, POSIX® methods, or memory mapping to be used as appropriate and easily changed. The STRS application can use the appropriate STRS Device methods when the STRS application just knows the handle name of the STRS Device, which gives more flexibility in configuring a data source or sink. For example, an application might be used to transmit data over the air obtained from a data source that might be configured as an application, device, queue, or file. Similarly, an application might be used to receive data over the air and send it to a data sink that might be configured as an application, device, queue, or file. Thus, an STRS Device may be used either to distribute functionality over multiple waveform applications or to abstract hardware functionality, further giving greater flexibility. This is analogous to redirection or pipes in UNIX®.

Since STRS Devices are only partially standardized by the STRS Device-provided Device Control API, extra methods may be implemented in an STRS Device to be used by the OE to establish the proper use of the HAL. A DEV_SetMemoryMap method is suggested in figure 5 to specify how a named value is associated with the appropriate location in the specialized hardware. This may be a non-portable construction but it does not violate the NASA Technical Standard. The idea is to have the STRS application code as portable as possible with the STRS Devices as lean as possible.

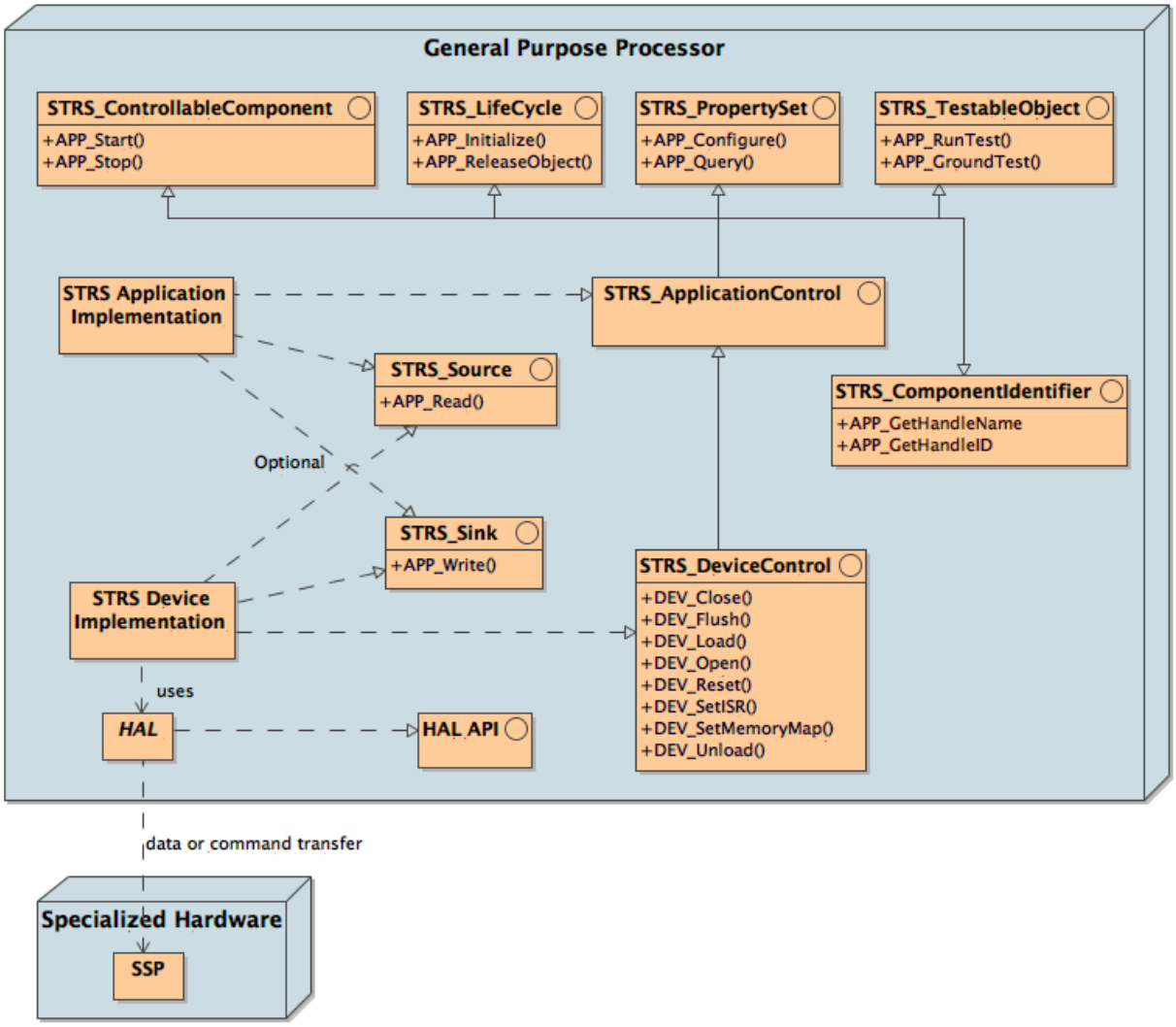

Figure 5—STRS Application/Device Structure

6.4 How to Configure and Control SDR Hardware

The configuration and control of the SDR hardware depends on where the intelligence exists; that is, which software component knows how to configure and control SDR hardware? A combination of software components (waveform application, STRS infrastructure, STRS Device, and HAL) knows how to configure and control SDR hardware.

a. The application knows about data values, the STRS Device knows about mappings in the GPP, and the HAL knows about how to take values and transfer them to the hardware.

b. The STRS application should be the target component for the parameters it controls and could pass to an STRS Device those parameters that need to be passed to the HAL.

c. STRS Application.

(1) The STRS application would control what data are configured.

(2) The STRS application has limited intelligence on how and where to enable the application to be portable.

(3) The STRS application knows the handle ID of the STRS Device to use.

(4) For example, an STRS application processes FREQUENCY, converts from floating point to integer in a format that is recognized by the STRS Device, and calls the appropriate method to configure the STRS Device.

d. STRS Device.

(1) The STRS Device controls how data get to the FPGA or other hardware.

(2) The STRS Device knows how to send data to the proper register in the FPGA using the HAL.

(3) The HAL may be external functions or inline functions that know the mappings from data addresses to registers in the FPGA.

e. STRS Infrastructure.

(1) The STRS infrastructure reads the configuration files or receives an external command and calls the STRS_Configure method for the appropriate target component.

(2) The STRS_Configure in the infrastructure calls the corresponding APP_Configure within the target component.

6.5 STRS Infrastructure Methods Do Not Belong to Any Class

The STRS infrastructure-provided methods beginning with “STRS_” do not belong to any class, since they have to be the same when called from C language implementations. If one is coding in C++, these methods should be defined using extern "C" {...}.

In NASA-STD-4009A, the STRS infrastructure provides the STRS infrastructure-provided Application Control API that supports application operation using the STRS Application-provided Application Control API in section 7.3.1. The STRS Infrastructure-provided Application Control API methods (section 7.3.2) that begin with “STRS_” correspond to the STRS Application-provided Application Control API methods (section 7.3.1) that begin with “APP_” and are used to access those methods. The STRS infrastructure implements these

methods for use by any STRS application or by any part of the infrastructure that is desired to be implemented in a portable way.

Since the C language is optional for STRS applications (see STRS-16, 18, 19), the STRS Application-provided application control methods beginning with “APP_” may belong to a class.

6.6 Explain Clocks and Timers

The clocks/timers are used for determining when something happens, how long something takes, and coordinating internal and external events, including timestamps for messages. As computer speeds increase, more real-time functions for communication may be performed in the GPP. Some functions currently in the FPGA(s) may be transitioned to the GPP when the GPPs are fast enough and capable enough to handle the additional signal processing functionality. These GPP functions would need access to high speed clocks/timers.

NASA-STD-4009A is designed to allow a clock/timer to be an extension of an STRS Device so that the functionality can be embedded in specialized hardware, if necessary. Multiple timers are only defined when they are required by the mission. An offset is usually specified to ensure that the clock is monotonically increasing from a previous power reset or is synchronized with another clock/timer.

Normally, each clock/timer has a base time, usually measured from when it is turned on. An offset may be used to keep the time monotonically increasing with each power cycle. An offset may also be used to coordinate with external events. The timing of external events, such as another satellite coming over the horizon or the availability of experimenters, may be used to power parts of the radio off and on so that the radio optimizes its power consumption and availability.

It is recommended that one clock/timer match the required timestamp for STRS_Log so that an application, service, the OE, or even STRS_Log itself could obtain that time in a consistent way. It was suggested that the time for the timestamp be retrieved via STRS_GetTime using the handle ID corresponding to handle name "STRS_DEFAULT_CLOCK_NAME" and kind given by property "STRS_DEFAULT_CLOCK_KIND". The use of this handle ID for STRS_SetTime may be restricted as necessary.

As specified by the mission, there should be at least one clock/timer with an epoch fixed to some Earth time zone, e.g., Coordinated Universal Time (UTC), so that the epoch could be adjusted to seconds from 1/1/1970 0:0:0, known adjustments for leap days and leap seconds applied, and the standard POSIX® time functions used. For example:

a. From seconds to calendar time: Obtain the number of seconds from the TimeWarp object, apply any offset needed, and use gmtime_r to convert a given time since epoch (a time_t value) into calendar time, expressed in UTC in the struct tm format.

b. From calendar time to seconds: Fill a struct tm with calendar time, convert to UNIX® time using mktime, apply any offset, and create a TimeWarp object.

6.7 FPGA Partial Reconfiguration

Partial reconfiguration is the process of configuring selected areas of an FPGA after its initial configuration. Xilinx indicates that a bitstream file can contain all the configuration commands and data necessary for partial reconfiguration. Therefore, STRS_DeviceLoad will work, and no new methods need to be defined. Whether the partial reconfiguration is fully transparent to the application(s) running on the FPGA or requires stopping and restarting or reconfiguring any other application(s) depends on the specifics of the partial reconfiguration.

6.8 Compliance Testing

STRS compliance of a vendor- or partner-provided SDR is assessed by source code inspection, document inspection, configuration file inspection, adding an application containing a reference to each STRS infrastructure method and testing that application. The name of that application is the STRS Command and Compliance, also known as WFCCN. WFCCN may be compiled with an STRS infrastructure to determine whether or not there are any missing constants, typedefs, or structs. These techniques are described in the STRS Compliance Testing document, which should be reviewed because of their complexity.

Since many of the STRS requirements are source code requirements, a standard test suite cannot test them fully. Since STRS is designed to allow multiple vendors to work together, certain source code artifacts have to be made available so that a subsequent STRS application developer or STRS integrator can use the methods, constants, typedefs, and structs required. The following is an example of a problem that WFCCN cannot be used to find: One vendor used noncompliant method signatures with *int* instead of STRS_Buffer_Size, but on that platform, both integer items compiled as the same type. Using a type that happens to correspond to the vendor's implementation of an STRS type is not necessarily portable to the next platform.

The STRS compliance could be evaluated at and by each vendor or partner and the results shared and discussed in one or more workshops at various points in the project life cycle. This alternative is to be decided by the mission or project. A full release and delivery of all STRS OE source code is not required in order to perform STRS compliance testing. Each vendor or partner should inspect his or her own software and documents before delivery. However, NASA found its own review to be invaluable to ensure greater compliance and promote understanding of differences among submitters.

Once the STRS radio artifacts are tested for STRS compliance, any noncompliances will be reported to the supplier and the mission or project, along with any suggestions. It is the responsibility of the mission or project to decide whether to grant deviations and waivers for any noncompliances that are not resolved.

6.9 Configuration Files Examples

To help the reader and implementer of the STRS architecture understand the development and use of the configuration files described in Appendix A, Example Configuration Files, an example of a configuration file based on that format was developed. This example of a configuration file, for a sample application WF_1, is shown in this NASA Technical Handbook in figure 6, Example of Predeployed Configuration File for Appendix A.

```
1  <?xml version="1.0" ?>
2  <?xml-stylesheet type="text/xsl" href="STRS_4009A.xsl"?>
3  <!-- ?xml version="1.0" encoding="UTF-8"? -->
4  <STRS xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="STRS_4009A.xsd">
5      <CONFIGURATION>
6          <W_HANDLE>
7              <HANDLENAME>WF1</HANDLENAME>
8              <WAVEFORM>%STRS_BASE%/WF1/WF1.inc</WAVEFORM>
9          </W_HANDLE>
10     </CONFIGURATION>
11 </STRS>
```

Figure 6—Example of Predeployed Configuration File for Appendix A

Here is the explanation, line by line:

(1) XML declaration.
(2) Extensible Stylesheet Language (XSL) file declaration.
(3) Comment.
(4) Open tag STRS and corresponding XML schema declaration.
(5) Open tag CONFIGURATION.
(6) Open tag W_HANDLE.
(7) Tag HANDLENAME containing WF_1 as the handle name.
(8) Tag WAVEFORM containing the OE-specific name used to instantiate WF_1 as a path to additional items to process.
(9) Close tag W_HANDLE.
(10) Close tag CONFIGURATION.
(11) Close tag STRS.

The OE-specific name is a file name containing the additional items to process, which are specified with their own XML configuration file and transformation process. This example of an application configuration file, for a sample application WF_1, is shown in this NASA Technical Handbook in figure 7, Example of Predeployed Configuration File for Application WF_1 for Appendix A.

```
<?xml version="1.0" ?>
<?xml-stylesheet type="text/xsl" href="STRS_4009A_WF.xsl"?>
<!-- ?xml version="1.0" encoding="UTF-8"? -->
<WAVEFORM xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="STRS_4009A_WF.xsd">
    <WFNAME>WF1</WFNAME>
    <WFACCESS>NONE</WFACCESS>
    <WFSTATE>INSTANTIATED</WFSTATE>
    <!-- The waveform is comprised of two files: WF1.out for the GPP, and WF1.bit for the D1.  -->
    <LOADFILE>
        <LOADFILENAME>%STRS_BASE%/WF1/%STRS_TARGET%/bin/WF1.out</LOADFILENAME>
        <LOADTARGET>SELF</LOADTARGET>
        <LOADMEMORY>
            <MEMORYSIZE>136546</MEMORYSIZE>
            <MEMORYUNITS>BYTES</MEMORYUNITS>
        </LOADMEMORY>
    </LOADFILE>
    <LOADFILE>
        <LOADFILENAME>%STRS_BASE%/WF1/%STRS_TARGET%/WF1.bit</LOADFILENAME>
        <LOADTARGET>D1</LOADTARGET>
        <LOADMEMORY>
            <MEMORYSIZE>2733252</MEMORYSIZE>
            <MEMORYUNITS>GATES</MEMORYUNITS>
        </LOADMEMORY>
    </LOADFILE>
    <ATTRIBUTE>
        <NAME>A</NAME>
        <VALUE>5</VALUE>
    </ATTRIBUTE>
    <ATTRIBUTE>
        <NAME>B</NAME>
        <VALUE>27</VALUE>
    </ATTRIBUTE>
    <ATTRIBUTE>
        <NAME>C</NAME>
        <VALUE>Non-numeric</VALUE>
    </ATTRIBUTE>
</WAVEFORM>
```

Figure 7—Example of Predeployed Configuration File for Application WF_1 for Appendix A

Here is the explanation, line by line:

(1) XML declaration.
(2) Extensible Stylesheet Language (XSL) file declaration.
(3) Comment.
(4) Open tag WAVEFORM and corresponding XML schema declaration.
(5) Tag WFNAME containing WF_1as the class name.
(6) Tag WFACCESS with WF_1 having no READ/WRITE access; that is, neither APP_Read nor APP_Write are implemented.
(7) Tag WFSTATE with final state as INSTANTIATED.
(8) Comment.
(9) Open tag LOADFILE.
(10) Tag LOADFILENAME containing the path to load WF1.out.
(11) Tag LOADTARGET containing SELF to indicate that it is loaded on the current GPP.
(12) Open tag LOADMEMORY.
(13) Tag MEMORYSIZE indicating that the size is 134k bytes.
(14) Tag MEMORYUNITS indicating that the size is measured in bytes.
(15) Close tag LOADMEMORY.
(16) Close tag LOADFILE.
(17) Open tag LOADFILE.
(18) Tag LOADFILENAME containing the path to WF1.bit.
(19) Tag LOADTARGET containing FPGA to indicate that it is loaded on the FPGA.
(20) Open tag LOADMEMORY.

(21) Tag MEMORYSIZE indicating that the size is 2670K gates.
(22) Tag MEMORYUNITS indicating that the size is measured in gates.
(23) Close tag LOADMEMORY.
(24) Close tag LOADFILE.
(25) Open tag ATTRIBUTE.
(26) Tag NAME containing A.
(27) Tag VALUE containing 5 as the value for A.
(28) Close tag ATTRIBUTE.
(29) Open tag ATTRIBUTE.
(30) Tag NAME with B.
(31) Tag VALUE containing 27 as the value for B.
(32) Close tag ATTRIBUTE.
(33) Open tag ATTRIBUTE.
(34) Tag NAME with C.
(35) Tag VALUE containing “Non-numeric” as the value for C.
(36) Close tag ATTRIBUTE.
(37) Close tag WAVEFORM.

The example in Appendix A splits the formatting into three parts to group the information logically. The following explanations further clarify the necessity and intent of Appendix A:

a. There is no necessity or requirement for splitting the platform configuration files into hardware and software parts as shown in Appendix A.1 and A.2. Splitting up the description this way was just a logical way to organize the description.

b. The format of the platform configuration file should allow some applications, devices, and services to be instantiated at boot-up or restart. Therefore, to use STRS_InstantiateApp to instantiate applications in all cases, the platform configuration file would specify the arguments. For this example, the OE-specific name argument referenced is the application configuration file name whose predeployed format is shown in Appendix A.3. The application configuration file may be independent of the platform configuration file.

6.10 C Language Naming Duplication

There will most likely be more than one application in an STRS radio. In the C language, there is no namespace support as in C++ or other object-oriented (OO) languages that scope the member functions to the class. Thus, there will be multiple implementations of the same STRS Application-provided API method names, starting with “APP_”, one in each implementation of an application.

One technique to allow multiple “instances” of C language applications could use APP_Instance to return a pointer to a table of pointers to the methods. Then, the OE could use these method locations to call the methods. This technique and variations are described in NASA/TM-2011-216948, Symbol Tables and Branch Tables: Linking Applications Together. The techniques specify the creation of a branch table or indirect address table for each application. To suppress

the common method names, compile and link each application separately. When the table is registered with the OE, the OE could use the table to call the appropriate method.

Another technique to allow multiple “instances” of C language applications with the same method names depends on loading new applications one at a time, sequentially, and capturing the new method locations instead of the old at the appropriate point in the process. The method locations are saved in a structure associated with the STRS application and the appropriate method is called as needed. The flow chart shown in figure 8, Obtain Array of Pointers to Methods, gives some highlights. Note that, besides the method illustrated in figure 8, there are other ways of creating an array of pointers to the C language methods. This array of pointers may be used to invoke those methods later.

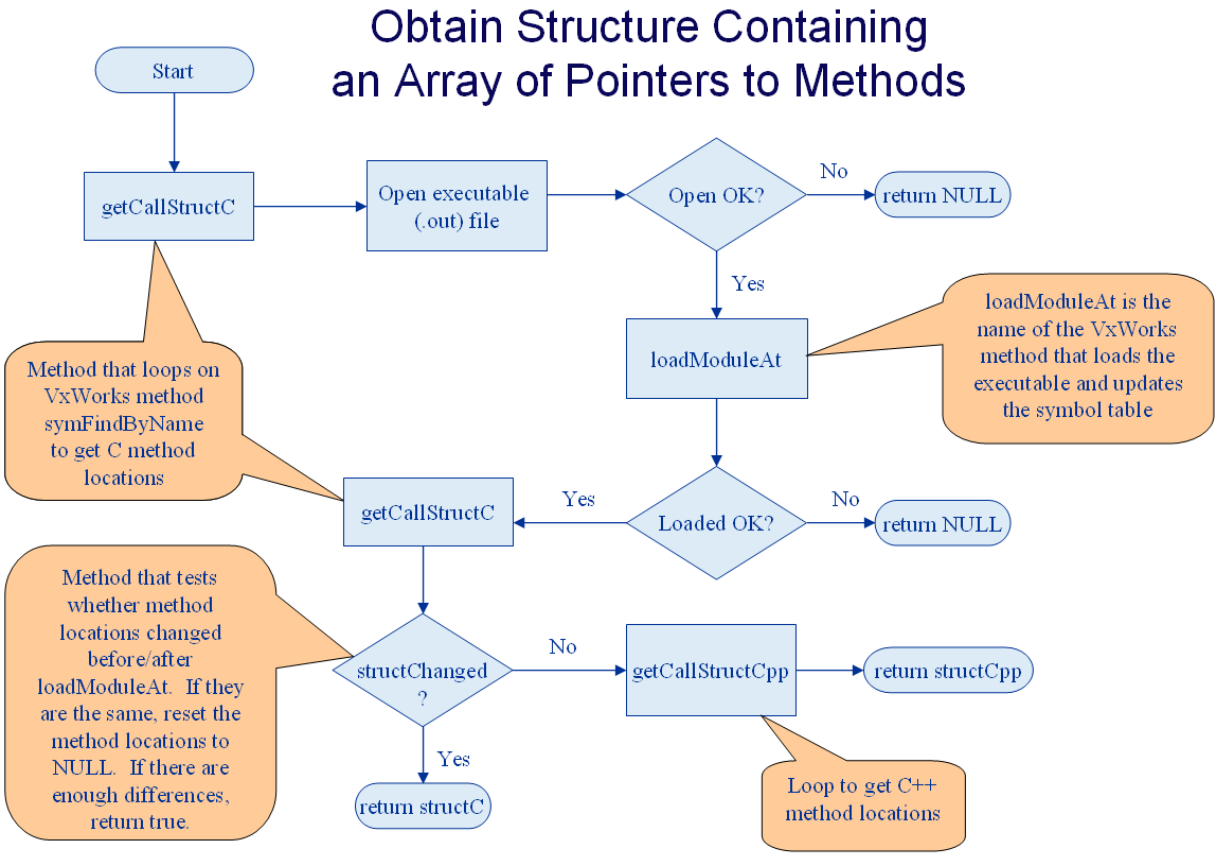

Figure 8—Obtain Array of Pointers to Methods

The technique demonstrated in figure 8 might not be possible on a platform that needs everything to be compiled and linked together ahead of time. Another technique would be to use message queuing to communicate between independent applications, but this technique might be awkward to use in practice. Another technique is to prepend the method name with a C-language class name equivalent.

6.11 Sequence Diagrams Depicting STRS API Calls

The following sequence diagrams depict the relationship between the STRS infrastructure-provided Application Control API beginning with “STRS_” and the corresponding STRS Application-provided Application Control API beginning with “APP_.” The methods described for figure 1 are those that cause a change in state. In this NASA Technical Handbook, the methods depicted in figure 9, Simplified Sequence Diagram for STRS_InstantiateApp, figure 10, Simplified Sequence Diagram for STRS_AbortApp, and figure 11, Simplified Sequence Diagram for STRS_Configure, contain both those that cause a change in state as well as those that do not. In this NASA Technical Handbook, since an STRS Device inherits all the methods from an STRS application, as shown in figure 5, the methods in figures 9, 10, and 11 for STRS applications could apply to STRS Devices as well. In figures 9, 10, and 11, “Command Source” is used for the object, internal to the radio, either an STRS application or part of the OE, which calls the STRS infrastructure methods.

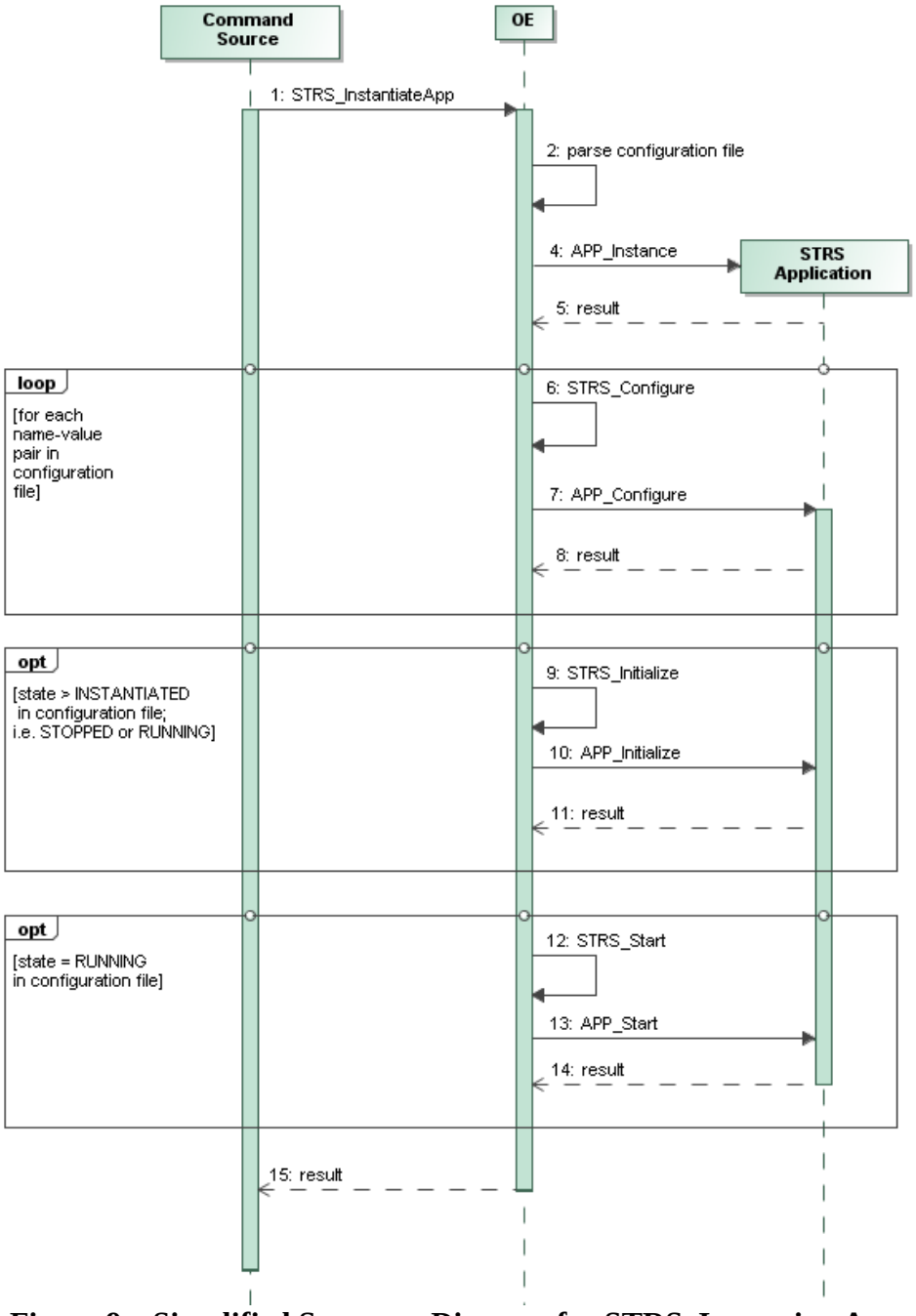

Figure 9—Simplified Sequence Diagram for STRS_InstantiateApp

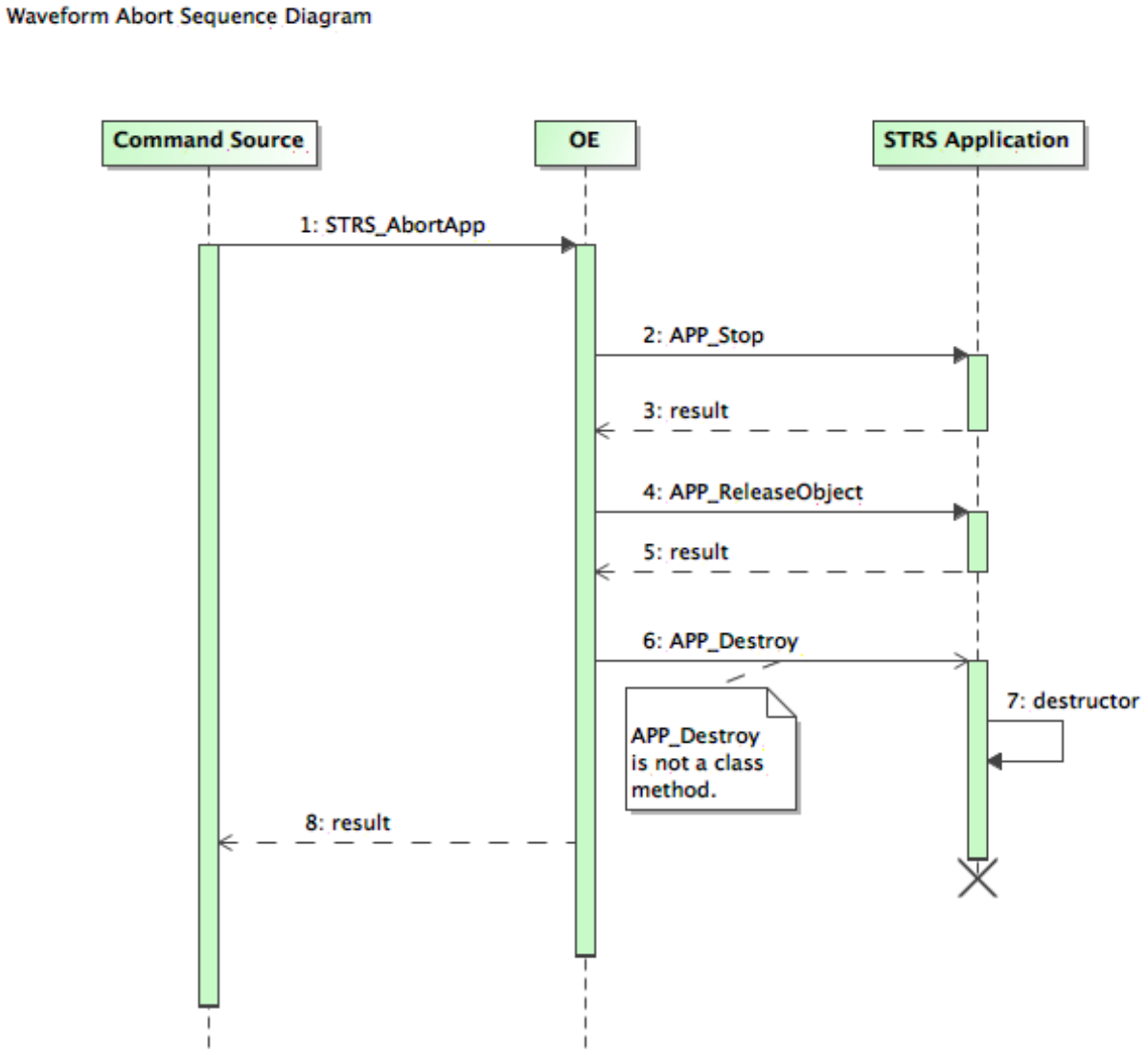

Figure 10—Simplified Sequence Diagram for STRS_AbortApp

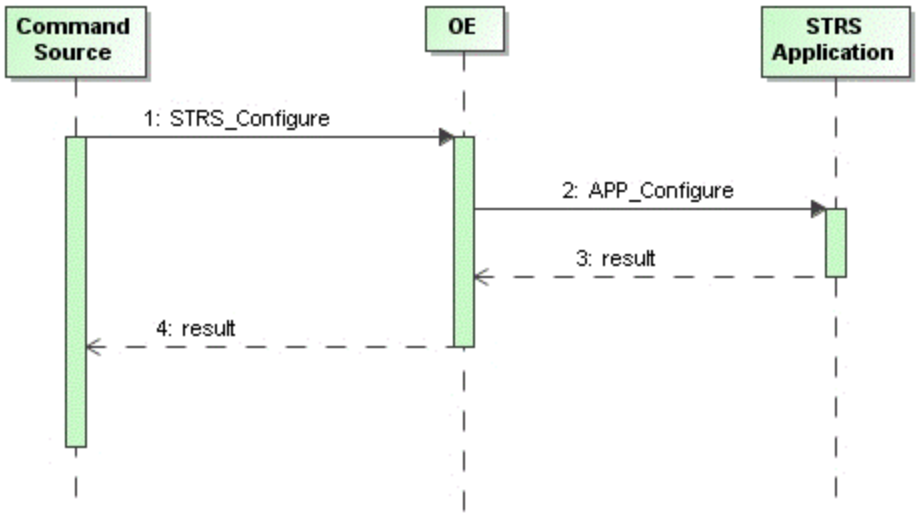

Figure 11—Simplified Sequence Diagram for STRS_Configure

A sequence diagram for each row in table 1, Substitutions for Figure 11, can be made from the diagram in figure 11, by substituting the "COMMAND SOURCE TO OE" method in place of STRS_Configure and the corresponding "OE TO STRS APPLICATION" method in place of APP_Configure.

Table 1—Substitutions for Figure 11

COMMAND SOURCE TO OE	OE TO STRS APPLICATION
STRS_GroundTest	APP_GroundTest
STRS_Initialize	APP _Initialize
STRS_Query	APP _Query
STRS_Read	APP _Read
STRS_ReleaseObject	APP _ReleaseObject
STRS_RunTest	APP _RunTest
STRS_Start	APP _Start
STRS_Stop	APP _Stop
STRS_Write	APP _Write

6.12 Why are APP_Instance and APP_Initialize Separate?

The APP_Instance and APP_Initialize methods are often used together successively but should not be combined because they have different functionality. The separation of APP_Instance and APP_Initialize supports encapsulation. It allows configuration to occur before APP_Initialize. In figure 9, STRS_InstantiateApp calls APP_Instance and then it may call APP_Configure and APP_Initialize, as specified by the configuration file. STRS_InitializeApp may do everything in one call or additional calls may be needed thereby giving the greatest flexibility. Also, note that

APP_Instance is a convenience function containing a constructor and saving any application identifying information.

6.13 Why Start with SCA?

The Wireless Innovation Forum (formerly SDR Forum) and SWRADIO by the OMG put so much effort into SCA that it was decided to investigate these architectures. The result of that investigation was that the CORBA requirements and XML parser requirements took up a lot of memory and machine cycles but were not really necessary for NASA. One study showed that eliminating CORBA and an XML parser reduced the memory footprint by an order of magnitude. So, to save on SWaP for NASA's space platforms, it was decided to create a similar STRS architecture without those disadvantages. After looking at use cases for NASA radios, very similar functionality to the SCA and SWRADIO was decided to be necessary. Similar method names to the application method names in SCA and SWRADIO were chosen for STRS. The reasoning was that it would be easy to take advantage of the many man-years of effort that had gone into defining those architectures, the Wireless Innovation Forum could comment on the STRS architecture due to the similarities, and that SDR design tools might be easier to use/generate for STRS applications.

A key recommendation from the Forum's space working group was to align with the OMG SWRADIO specification where possible. To that end, mappings from the OMG SWRADIO PIM to the STRS platform-specific model (PSM) were discussed. There were only minor differences in the Space PIM that could map into STRS from OMG's SWRADIO PIM that mapped into SCA. A quote from the Forum's study:

> *The SDR Forum recommended that the STRS align with the SDR Forum, the OMG, and the IEEE SCC41 for purposes of distributing the burden and cost of non-recurring engineering (NRE) across NASA and all consortia members contributing to the STRS, and to further broaden and enhance the quality of the implementation and deployment of STRS-based standards.*

NASA's configuration files could be much simpler, because NASA's radios were less distributed with no dynamic aspects needed to be specified in the configuration files. Furthermore, it was decided that by preprocessing any XML configuration files, a much simpler parser could be used on much simpler data.

In considerations for NASA-STD-4009A, it was determined that NASA's radios could be even simpler and that some further complexities could be eliminated. Such changes eliminated the possibility of harmonization with SCA but were more flexible regarding use with Core Flight Software (cFS) and other frameworks.

6.14 Security for STRS

Security aspects need to be considered for any STRS radio. There are currently no STRS requirements for security, and it is assumed to be up to the project/mission to define any security requirements. It was determined that NASA radios typically do not require DO-178, Software Considerations in Airborne Systems and Equipment Certification; and red/black separation. Security is needed to:

a. Verify/validate external commands such that:

(1) They come from the appropriate source (e.g., using data in a CCSDS wrapper).

(2) They have not been compromised (e.g., using encryption, signing, parity bit, checksum, cyclic redundancy check).

(3) They are in the appropriate format for commands.

This is usually defined by the command and control for the mission and not by STRS. The security functions should be encapsulated, separate from the external command and control interpreter, so that the functionality may be changed if necessary without affecting the STRS application implementation. This functionality may be invoked by the STRS OE implementation or as a service for over-the-air command and control.

b. Verify/validate internal commands; i.e., do not allow the radio to try to do anything risky or make itself inoperable.

(1) Do not allow radio to call methods for which the handle ID is inappropriate.

(2) Restricting the radio as to what it can do is left as a possible project/mission requirement. For example, one might restrict one waveform from aborting any other waveform. In this case, it is suggested that a table be configured containing allowed or disallowed commands and the associated source(s) handle names and target(s) handle names for which the command applies such that the table could be used to validate a command.

(3) Specifying a key to allow the radio to override the restrictions of item “b(2)” is also left as a possible project/mission requirement. In which case, security keys and an authentication method is required.

Security requirements are defined by the project/mission and not by STRS. The security functions of item “b(2)” and item “b(3)” should be encapsulated so that the functionality may be changed if necessary, without affecting the STRS application implementation. This functionality may then be invoked by the STRS infrastructure implementation.

6.15 What is Configurable Hardware Design?

The term “configurable hardware design” is used throughout the STRS documentation to signify the items required to capture the digital logic of the hardware that can be configured remotely, such as an FPGA. Configurable hardware design includes the items created to document the design of the hardware, including the source code (e.g., very high speed integrated circuits (VHSIC) hardware description language (VHDL), Verilog®) and the loadable files (e.g., FPGA image).

The term “configurable hardware design” replaces the commonly used term “firmware” in earlier versions of STRS documentation. Many definitions for firmware, including the latest IEEE definition which was written in 1990, state that firmware resides in read-only memory and cannot be modified. The changing definition of “firmware” would likely lead to confusion, and a new term was selected for the STRS documentation. Additional terms such as “complex electronics,” “configurable logic device,” “programmable logic device,” and “software” were considered, but each term was rejected due to potential confusion or implied limitations if the term was used.

The SDR community is unique in that it uses GPPs and configurable hardware design in a single application. The term “software” in some contexts of the STRS (and other SDR-related) documentation may include configurable hardware design. For example, whenever the term software defined radio is used, both GPP and configurable hardware design are included. The STRS architecture does not dictate processes or organizational structure for use in developing the software or configurable hardware design. The project developing the SDR or application has to dictate the required process.

6.16 Why is there STRS_InstantiateAPP and no STRS_Instance method?

The application instantiation process should not only instantiate the application but ensure that the application starts in a known state. The application may be comprised of multiple items that are loaded as part of the application instantiation process. The application instantiation process should also ensure that data is configured consistently. This potentially multi-step process was most easily customized by putting the variable instantiation data in a configuration or script file. The configuration file is parsed and processed by the OE in STRS_InstantiateApp before calling any methods in the application. Figure 10 illustrates that STRS_InstantiateApp could call multiple application-provided methods as needed to begin execution in a consistent well-defined manner. APP_Instance is the only method that is required to be called, but APP_Configure is highly encouraged so that data does not have to be hard-coded in the application but may be changed according to predetermined conditions and known upon application start-up in a consistent manner. This is not a simple one-to-one relationship between STRS_InstantiateApp and the other methods it invokes; whereas, it is a simple one-to-one relationship for most of the other STRS application-provided application control methods and the corresponding STRS infrastructure-provided application control methods.

To ensure that a user could not instantiate an application in an unknown state, STRS_Instance that invoked APP_Instance was not allowed. STRS_Instance calling APP_Instance was disallowed to discourage a user from instantiating an application in an unknown state. The method name STRS_Instance was not reused for the multi-step process in order to highlight the difference between simple instantiation of an application and instantiation of an application using a configuration file.

6.17 Uniqueness of Handle Names and IDs

A handle name is a C language character string that specifies the name of a specific instance of an STRS resource that may be an application, service, device, file, or queue. It is used in external commands and messages. It is usually the same at each start-up of the radio platform unless specifically changed. A handle ID is a numerical value determined by the infrastructure that could contain an index, hash, address, or other data that help the OE locate the information necessary to access the resource. It is used in internal method invocations and is restricted so that STRS_ValidateHandleID can determine whether there is an error and STRS_GetErrorQueue can indicate which error queue to use. It is often different at each start-up of the radio platform even if the handle name remains the same.

6.18 Are there any exception-safety rules?

The methods in the STRS APIs are not allowed to throw exceptions. Suppose a pre-existing library contains functions that can throw exceptions and suppose a method in the STRS APIs calls the library function. Then the method in the STRS APIs must catch all exceptions that the library function can throw. The return result from the method in the STRS APIs will indicate whether there has been an error or not.

6.19 What does it mean that an STRS Device or STRS Service may be part of the OE?

An STRS Device or STRS Service is an optional part of the OE depending on the project/mission requirements. The intention was that STRS Devices and STRS Services encapsulate non-portable functionality or functionality that applies over multiple waveform applications. Also, a further intention was that the STRS Devices have additional capabilities beyond that allowed to an STRS application as if the STRS Device were part of the OE. It should be required by the project that the platform provider create a sample application that uses a sample STRS Device to exercise both the hardware and software for testing and that serves as a model for what can be done by the application developer. The STRS Device would be included with the corresponding sample STRS application. An STRS Device would be specified as part of the OE, for example, if the device is not really programmable but can be adjusted or turned on/off by the HAL.

To access the specialized hardware containing application functionality, the STRS Device must correspond to an STRS application such that the application developer must be able to modify the sample and compile it with the user's STRS application. The application developer needs to

wrap the HAL invocation(s) into STRS Devices and handle the message format to/from the application and specialized hardware.

6.20 How does the application know how to put the data (address, command, data) into the buffer in the specialized hardware?

Use the STRS_Configure method to change settings that are configuration related and do not change at every message. The contents and format of the buffer can be defined for your application.

An STRS Device is a bridge between the waveform application and the HAL. So, an application could be a sink for some data created elsewhere by implementing APP_Write. That application could call STRS_Write to send the data from the application to the STRS Device that also acts as a sink by implementing APP_Write in the STRS Device. Then the STRS Device could use the HAL to send the data to the specialized hardware.

It has been suggested that EDS be used to configure STRS Devices. The CCSDS has a Draft Recommended Standard for Spacecraft Onboard Interface Services—XML specification for Electronic Data Sheets for Onboard Devices, CCSDS 876.0. Allowing the use of CCSDS EDS has merit and does not conflict with the STRS requirements since the data sheet configuration file would be in XML as required and may be transformed to a deployed format if needed. Since the CCSDS EDS allows extensions, any specific requirements in this area are better left to the Project.

6.21 Can STRS applications run in multiple address spaces?

Yes; however, multiple address spaces for multiple STRS applications would require some means of enabling the OE to call the STRS application-provided methods and vice versa. Things can get as complicated as necessary depending on the desired distribution of the software across the address spaces and the availability of the appropriate middleware or equivalent. Using STRS_API as a representative infrastructure-provided interface called from some command source and APP_API as a corresponding application-provided interface implemented within some application, some examples could be depicted as:

1. $STRS\_API_1$ -> APP_API_1
2. $STRS\_API_1$ -> APP_API_1 -> STRS_API_Ethernet-> $STRS\_API_2$ -> APP_API_2
3. $STRS\_API_1$ -> APP_API(tx) ~~~ APP_API(rx) -> $STRS\_API_2$ -> APP_API_2
4. $STRS\_API_1$ -> distribution point -> ... -> collection point -> APP_API_2
5. $STRS\_API_1$ -> distribution point -> ... -> collection point -> APP_API_1 -> $STRS\_API_2$ -> distribution point -> ... -> collection point -> APP_API_2

These are not exhaustive.

For 1 above, where the call is direct, dynamic linking may be used; but both must be in the same address space. The "this" pointer or equivalent is used as needed for the STRS API methods to

call the corresponding APP API methods. For 2 or 3 above, the call is direct to some application object that controls the access to another application in another address space (or even another radio) via Ethernet, remote procedure call (RPC), over-the-air (OTA) signal, Bluetooth, etc. For situations where this will not work, including multiple address spaces as depicted in 4 above, any middleware may be inserted in between, such as CORBA. In 5 above, an additional level of complication was added, just to show one of the many combinations. Some of these ideas were used when a graphical user interface (GUI) was added that was situated on a different computer using a different operating system.

The following figure 12, Multiple Connected Radios, can be used to illustrate 2 or 3 above when the networking is encapsulated in applications for sending and receiving using Ethernet or other means of message passing across the address spaces. For the figure below, WF_1 is the command source for $STRS\_API_1$ and WF_{tx} is the sink for the corresponding command. After passing through the connection, the same command is repeated where now the dependent command source for $STRS\_API_2$ is WF_{rx} and WF_2 is the sink. One cannot get greater separation than when the applications reside in what appear to be different radios. The following figure can be used to illustrate 4 above when the connection from WF_{tx} to WF_{rx} is replaced by middleware marshalling and unmarshalling the data.

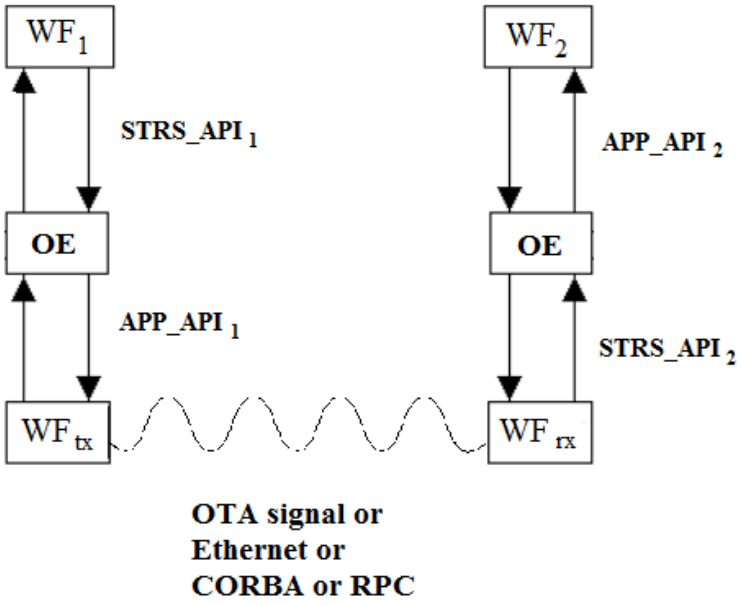

Figure 12—Multiple Connected Radios

6.22 Does an STRS application require a main entry point?

No; the STRS Architecture Standard does not require that a main program be defined for an STRS application. An STRS application may require a main entry point if specified by the platform provider.

The idea was to leave as much leeway as possible while having much of the initialization and control code being portable. The word "application" was used in its broadest sense so that it can include a main program or not, depending on the STRS infrastructure-required interfaces. The infrastructure-provider is obligated to describe the interface mechanism so that additional applications may be written or ported. The infrastructure must contain a main entry point or the equivalent so it can be started.

The integrator must have all applications, shared libraries as well as main at compile and link time for proper linking on the target. The dynamic linking/loading capabilities of some operating systems may be useful to add additional applications at a later time as needed.

6.23 How is STRS_TimeSynch used to adjust time?

Pseudocode for STRS_TimeSynch might be:

```
STRS_TimeSynch(me,refDev,kind',tgtDev,kind,stepMax) {
   STRS_GetTime(me,refDev,ref_base, kind', ref_kind); // Get reference time
   STRS_GetTime(me,tgtDev,tgt_base, kind,  tgt_kind); // Get target time
   tgt_old = tgt_kind;                  // Old time
   Δold = tgt_old - tgt_base;           // Old delta
   tgt_kind = ref_kind';                // Synchronization/new time
   Δtgt = tgt_kind - tgt_base;          // Compute/new delta
   chg = tgt_kind - tgt_old;            // compute change.
   If (stepMax == 0)                    // Test if jump.
   {                                    // No stepMax => jump.
      rtn = 0;                          // OK as jump.  Return zero.
   } else if (chg < - stepMax)          // Test change against step size.
   {                  // Change is negative and bigger than step size.
      rtn = (-chg – stepMax) / stepMax;    // Return amount left to do.
      Δtgt = Δold - stepMax;  // Increase negatively by a step size.
   } else if (chg ≤ stepMax)  // Test change against step size.
   {                          // Change is smaller than step size.
      rtn = 0;                // OK, as jump.  Return zero.
   } else {           // Change is positive and larger than step size.
     rtn = (chg – stepMax) / stepMax;     // Return amount left to do.
     Δtgt = Δold + stepMax;   // Increase by a step size.
   }
   STRS_SetTime(me,tgtDev, kind, Δtgt);
   return rtn;
}
```

6.24 How is Clock Rate Adjustment Used?

A clock may drift due to age or environmental factors or relativistic changes. STRS provides for a clock rate adjustment using STRS_SetTimeAdjust. A user would need to set up loops to check the clock periodically for drift. One way is to use STRS_TimeSynch with a maximum time step that returns the number of maximum time steps left to do. If this seldom returns zero, a time adjustment is in order. The size of the adjustment is clock-dependent. Iterate until the clocks generally stay in synch.

6.25 What is OE-Specified String for the Application to be Instantiated?

The STRS_InstantiateApp method contains an argument described as: The string used to identify the application for instantiation that may impose additional operations to be performed as documented by the platform provider. The project manager and platform provider will decide whether this parameter is to be a class name, or a file name. Usually, the latter is selected to be a software configuration file or script file to give the greatest flexibility for the STRS infrastructure to specify default or initial attributes of software items pertaining to the platform or applications, services, and devices contained on an STRS radio. Application-specific information for configuration and customization of installed applications may be provided, as well as information for the STRS infrastructure to use to instantiate applications on the radio GPP. Such files provide STRS application developers with flexibility in choosing parameters and values deemed pertinent to the implementation. In a specific case actually encountered, the parameters within the application changed based on ambient conditions such that a set of calibration tests were performed and the results stored in a configuration file. Using a configuration file allowed the code to remain unchanged.

For a configuration file, the use of XML version 1.0 is recommended to define the STRS platform and application configuration data because XML has the ability to identify configuration information in a standard, human-legible, precise, flexible, and adaptable method. XML is a markup language for documents containing structured information that contains both content and some indication of what role that content plays. XML defines tags containing or delimiting content and showing the relationships between them (see http://www.w3.org/XML/). Because of the extra overhead required to transmit and process XML-formatted data, it is anticipated that the XML configuration file would be preparsed, additional error checking on the file will be performed, and the XML file transformed into a simpler, more compact form prior to transmission. This process will reformat the configuration file into an appropriately optimized configuration file, which will subsequently be loaded into the radio.

An XML Schema Definition (XSD) file contains an XML schema describing the structure and constraining the content of XML documents (See http://www.w3.org/XML/Schema). An XML schema can provide error checking of allowable values, dependencies, and range limits of configuration parameters.

An XML interface tool could be used to create and modify platform and application configuration files. When used with an XML schema, these tools standardize the XML data entry, enforcing error checking and interdependency checks to ensure that the entered data are correct and within the hardware and software limits.

The XML should be preprocessed to a platform-specific format to optimize space on the STRS radio while keeping the equivalent content. Figure 13, XML Transformation and Validation, illustrates the relationships between an XML file and its corresponding schema, as well as representing the preprocessing of the XML file in a simplified form using Extensible Stylesheet Language (XSL). XSL is a family of recommendations for defining XML document transformation into text for presentation. (See http://www.w3.org/Style/XSL/.)

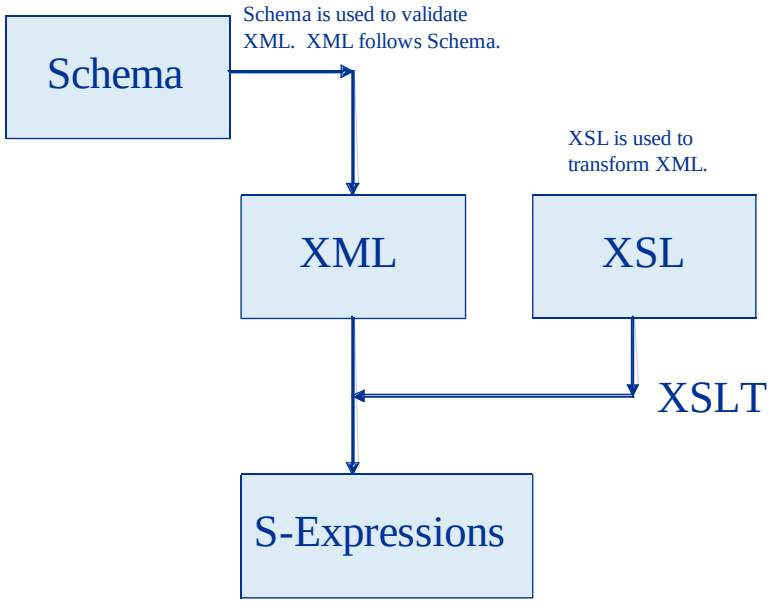

Figure 13—XML Transformation and Validation

The application configuration files would be developed by the STRS application developer and STRS integrators using information obtained from both the STRS platform provider and the STRS application developers. The STRS integrators use the application configuration files to install the applications on the platform. These roles may overlap at times.

The STRS platform provider documents the method to describe and use the hardware and software environment for the STRS infrastructure. Developing platform configuration file(s) is the likely method to be used by an STRS platform provider to identify the existence of the different hardware modules and their associated configuration files to allow the OE to instantiate drivers and test applications. An STRS platform configuration file may be used when starting the STRS infrastructure to configure various properties of the STRS platform. Configuring these properties at run-time allows greater flexibility than configuring them at compile-time. To increase the runtime flexibility of the STRS platform, the STRS infrastructure is likely to use deployed platform configuration files to determine the existence and attributes of the files, devices, queues, waveforms, and services contained on the STRS radio. Attributes of files, devices, and queues could include access (read/write, both, or append), type (text or binary), and other properties. The name of the starting configuration file(s) may be provided to the STRS infrastructure upon initialization. The predeployed platform configuration files should contain platform configuration information such as the following:

a. Hardware module names and types.
b. Memory types, sizes, and access.
c. Memory mapping.
d. Unique names and attributes of files, devices, queues, services, and applications known to the OE at boot-up.

A predeployed STRS application configuration file could be created by the STRS integrator using platform information, the XML schema supplied by the STRS platform provider, and application information provided by the STRS application developer. The deployed application configuration file would be used by the infrastructure (see the STRS_InstantiateApp method) when starting the STRS application to configure various properties of the STRS application. Configuring these properties at run time allows greater flexibility than configuring them at compile time. For example, one might configure the STRS handle names of files, devices, queues, waveforms, and services needed by the STRS application so that these can be easily changed. Since a service is actually an application that has been incorporated into the STRS infrastructure, the format of the application configuration file should be a subset of the format of the platform configuration file as specified by the schema. If any STRS application resources need to be loaded separately into memory or into a device, such as an FPGA, before the STRS application can function properly, these should be specified in the configuration file for that STRS application.

The predeployed STRS application configuration file may identify the following application attributes and default values:

(1) Identification.
   A. Class name.
(2) State after processing the configuration file (if applicable) or script to execute to get there. This is as documented by the platform provider.
(3) Any resources to be loaded separately (if applicable).
   A. Filename of loadable image.
   B. Target on which to put loadable image file.
   C. Target memory in bytes, number of gates, or logic elements.
(4) Initial or default values for all distinct operationally configurable parameters.

The most common transformation of STRS configuration files uses name/value pairs divided by a separator for the deployed format.

6.26 STRS Radio Startup Process, Platform Diagnostics, and Built-in Test?

The platform boot process generally has many parts, depending on the project requirements for safety and confidence. Built-in testing is often performed even before the STRS OE is initialized. Once the STRS OE is started, it is expected that further platform diagnostics be reported to the STRS OE using the STRS APIs as required by STRS-2. The STRS radio is expected to have a known power up condition where it is ready to receive commands, send telemetry, and transmit and receive communications according to the project requirements. Upgrades to the boot process could allow changes to the OS, OE, and/or applications to be made in a secure fashion.

6.27 Cognitive, Navigation, and Other Services

The STRS architecture allows STRS radios to provide cognitive, radiometric tracking, navigation, and other services that are integrated with communication services. Cognitive services obtain metrics to monitor the radios' operation, learns how to optimize the performance of the radio, and modifies the operation of the radio accordingly. Radiometric tracking is the process of measuring the characteristics of radio signals that have been transmitted (potentially over several legs) in order to extract information relating to the signal's change in frequency and/or time of transit. A radio has the fundamental component needed for tracking—a radio signal. The SDR simplifies the navigation architecture because it minimizes mass, power, and volume requirements while maximizing flexibility. An SDR provides the flexibility to respond to different mission phase requirements and to dynamic application requirements where signal structures may change. This is the fundamental reason for considering the implementation of an SDR with tracking and navigation functionality.

6.28 C and C++ Compatibility?

Various editors and compilers usually use the file extension (after the last period) to make a decision about the type of file and how to use it so that it is helpful to have different code and header file extensions for C and C++. Furthermore, when C code is used in a C++ method, 'extern "C"' may be used to indicate that that the interface is to be that for C rather than C++. For example, to declare a C language method in C++:

```
extern "C" {
        // Function to report statistics (C language)
        void MyServiceReportStatistics( MyServiceType myType );
} // End of extern C
```

There is a way of using a #ifdef to check whether the code is in C code or C++ code, if necessary, when the same header file is used for both C and C++. For example:

```
#ifdef __cplusplus
extern "C" {
#endif
                …
#ifdef __cplusplus
}
#endif
```

7. STRS REQUIREMENTS, RATIONALE, AND VERIFICATION METHOD

The following sections address each requirement in turn, displaying the rationale, the related higher-level requirements, verification method, and other pertinent information, which augment the general rationale given earlier in this document.

In each section, the title line contains the requirement number and the title of the requirement. That is followed by the text of the requirement. The *rationale* describes why the requirement is needed. The *category* contains one or more of the summary capabilities from NASA/TM-2007-215042. The categories are chosen from the following list: adaptability, availability, extensibility, flexibility, interoperability, portability, (implying reusability too), scalability, reliability, and reconfigurability. The *traced-from* specifies the section numbers in NASA/TM-2007-215042 that apply to this requirement. The *use case* specifies the names of the use case sections in NASA/TP-2008-214813 that apply to this requirement. The *related to* specifies the part of the STRS radio that has to satisfy the requirement where *platform* indicates that the hardware and related documentation are tested, *OE* indicates that the OS and infrastructure and related documentation are tested, and *application* indicates that the application and related documentation are tested. The *notes* contain additional explanations for the requirement.

Verification methods are used to show that the requirement has been met. The verification method is chosen from the following list: Analysis, inspection, observation, similarity, or test. Tests are not used because tests are expected to be mission requirements rather than STRS requirements.

a. Analysis is the process of utilizing analytical techniques to verify that requirements have been satisfied. This method may be used when qualification by test is not possible, when a test would introduce significant risk into the software, or when analysis is an appropriate, cost-effective qualification method.

b. Inspection is a qualification method consisting of investigation without the use of special tests. Inspection is usually a visual examination, but it may be computer-aided. Using a script or WFCCN refers to the STRS compliance tools as described in the NASA/TM-2011-217266**,** STRS Compliance Testing document. A compliance certification testing facility is available at Glenn Research Center (GRC) to perform compliance testing and will test all STRS applications submitted to the STRS Application Repository. The users may use their own tools as an independent check of an OE or of an application prior to submitting the application to the STRS Application Repository.

(1) Using a script or WFCCN is a type of inspection that is computer-aided.
(2) Using a compliance tool implies a script or WFCCN.
(3) Using a program, such as XMLSpy, validates the XML schema and the predeployed configuration file against its schema.

c. Observation is a method of qualification that is limited to readily observable functional operation to determine compliance with requirements. This method of qualification does not require the use of special equipment or sophisticated instrumentation.

d. Similarity is the process of using analysis and/or "delta testing" to prove the design adequacy of an item by reference to the prior qualification of an identifiable item that has been qualified for a similar application.

e. Test is a qualification method that employs technical means including, but not limited to, the evaluation of functional characteristics by the use of special equipment or instrumentation, simulation techniques, and the application of established principles and procedures to determine compliance with requirements. The analysis of data derived from a test is an integral part of the method.

Note: The variables STRS_APP_INSTANTIATED, STRS_APP_STOPPED, and STRS_APP_RUNNING, shown in the examples, are variables for states, which would have to be defined locally.

7.1 STRS-1 Power Up

Requirement	An STRS platform shall have a known state after completion of the power-up process.
Rationale	To increase the reliability of the STRS platform after reboot or power cycle, the radio has to be able to return to full operation autonomously without the need for external equipment or procedures.
Category	Availability, Reliability
Traced-from	4.3, 5.17
Use Case	Power On
Applicable to	OE developer: usually platform provider
Notes	A known state is one that is predictable from documentation or from configuration file(s) or scripts or some combination thereof.
Verification Method	Observation of radio operation.

7.2 STRS-2 Provide Platform Diagnostics

Requirement	A module's diagnostic information shall be available via the STRS APIs.
Rationale	To increase the reliability and availability of the STRS platform, there has to be a means of providing data to identify configuration information as well as status and fault identification. Data for both BITs and recognition of operational degradation and malfunction have to be available.
Category	Reliability, Availability
Traced-from	4.3, 5.15, 5.16

Use Case	Fault Management, Built-In Test
Applicable to	OE developer: usually platform provider
Notes	None
Verification Method	Observation of radio operation.

7.3 STRS-3 Use Platform Diagnostics (Deleted)

Rationale	This requirement was deleted because STRS-2 wording was made more robust, STRS-3 became redundant. STRS-3 previously stated: Self-diagnostic and fault-detection data shall be created for each module so that it is accessible to the STRS OE.

7.4 STRS-4 Document Resources

Requirement	The STRS platform provider shall describe, in the HID document, the behavior and capability of each major functional device or resource available for use by waveforms, services, or other applications (e.g., FPGA, GPP, DSP, or memory), noting any operational limitations.
Rationale	Waveform developers need to know the features and limitations of the platform for their applications. Once the radio has been procured, NASA has the knowledge to procure or produce new or additional modules using HID information. Also, future module replacement or additions will be possible without designing a new platform.
Category	Adaptability
Traced-from	4.7, 4.8, 5.2, 5.3
Use Case	None
Applicable to	Platform provider
Notes	None
Verification Method	Inspection of HID document.

7.5 STRS-5 Document Capability

Requirement	The STRS platform provider shall describe, in the HID document, the reconfigurability behavior and capability of each reconfigurable component.
Rationale	Waveform developers need to know the features and limitations of the platform for their applications. Once the radio has been procured, NASA has the knowledge to procure or produce new or additional modules using HID information. Also, future module replacement or additions will be possible without designing a new platform.
Category	Adaptability
Traced-from	4.7, 4.8, 5.2, 5.3
Use Case	None

Applicable to	Platform provider
Notes	None
Verification Method	Inspection of HID document.

7.6 STRS-6 Document Radio Frequency (RF) Behavior

Requirement	The STRS platform provider shall describe, in the HID document, the behavior and performance of the RF modular component(s).
Rationale	Waveform developers need to know the features and limitations of the platform for their applications. Once the radio has been procured, NASA has the knowledge to procure or produce new or additional modules using HID information. Also, future module replacement or additions will be possible without designing a new platform.
Category	Interoperability, Adaptability
Traced-from	4.6, 4.7, 4.8, 5.2, 5.3
Use Case	None
Applicable to	Platform provider
Notes	None
Verification Method	Inspection of HID document.

7.7 STRS-7 Document Module Interfaces

Requirement	The STRS platform provider shall describe, in the HID document, the interfaces that are provided to and from each modular component of the STRS platform.
Rationale	Waveform developers need to know the features and limitations of the platform for their applications. Once the radio has been procured, NASA has the knowledge to procure or produce new or additional modules using HID information. Also, future module replacement or additions will be possible without designing a new platform.
Category	Interoperability, Adaptability
Traced-from	4.2, 4.7, 4.8, 5.2, 5.3
Use Case	None
Applicable to	Platform provider
Notes	None
Verification Method	Inspection of HID document.

7.8 STRS-8 Document Module Control

Requirement	The STRS 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).
Rationale	Waveform developers need to know the features and limitations of the platform for their applications. Once the radio has been procured, NASA has the knowledge to procure or produce new or additional modules using HID information. Also, future module replacement or additions will be possible without designing a new platform.
Category	Interoperability, Adaptability
Traced-from	4.7, 4.8, 5.2, 5.3
Use Case	None
Applicable to	Platform provider
Notes	None
Verification Method	Inspection of HID document.

7.9 STRS-9 Document Power

Requirement	The STRS platform provider shall describe, in the HID document, the behavior and performance of any power supply or power converter modular component(s).
Rationale	Waveform developers need to know the features and limitations of the platform for their applications. Once the radio has been procured, NASA has the knowledge to procure or produce new or additional modules using HID information. Also, future module replacement or additions will be possible without designing a new platform.
Category	Reliability, Adaptability
Traced-from	4.7, 4.8, 5.2, 5.3
Use Case	None
Applicable to	Platform provider
Notes	See also STRS-108.
Verification Method	Inspection of HID document.

7.10 STRS-10 STRS Application Uses OE

Requirement	An STRS application shall use the STRS infrastructure-provided APIs and POSIX® API for access to platform resources.
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. Rather than creating many more APIs in STRS, the currently available POSIX® standard was selected. Thus, POSIX® subsets were chosen to implement certain OS services missing from the list of STRS Application-provided methods. The POSIX® subsets are widely available, implemented by multiple OSs, and scalable. Layering of the architecture separates and encapsulates functionality so that the parts are less influenced by changes to the other. This separation of functionality promotes portability.
Category	Portability
Traced-from	4.2, 4.7, 5.1
Use Case	None
Applicable to	Application developer
Notes	None
Verification Method	Inspection using STRS compliance tool.

7.11 STRS-11 OE Uses HAL

Requirement	The STRS infrastructure shall use the STRS platform HAL APIs to communicate with application components on the platform specialized hardware via the physical interface defined by the STRS platform provider.
Rationale	The HAL API is to be published so that specialized hardware made by one company may be integrated with the STRS infrastructure made by a different company. The HAL API documentation is to include a description of each method or function used, including its calling sequence, return values, an explanation of its functionality, any preconditions before using the method or function, and the status after using the method or function. The HAL API documentation is to also contain information about the underlying hardware such as address and data interfaces, interrupt input and output, power connections, and other control and data lines necessary to operate in the STRS platform environment.
Category	Adaptability, Extensibility
Traced-from	4.4, 4.5, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	See also STRS-92.
Verification Method	Inspection of HAL document.

7.12 STRS-12 STRS Application Repository

Requirement

The following application or OE development artifacts shall be submitted to the NASA STRS Application Repository:

(1) Application (or OE) or system component software and configurable hardware design simulation model(s) and/or documentation). (Design Description Document)
(2) Documentation of external interfaces for STRS application, devices, or configurable hardware design (e.g., signal names, descriptions, polarity, format, data type, and timing constraints). (HID)
(3) Documentation of STRS application or OE behavior, initialization, and adaptability (e.g., configurable and queryable data items). (Design Description Document, User's Guide)
(4) Application or OE function sources (e.g., C, C++, header files, VHSIC VHDL, and Verilog®). (Artifacts)
(5) Application or OE libraries, if applicable (e.g., electronic design interchange format (EDIF) and Dynamic Link Library (DLL)). (Artifacts)
(6) Documentation of application (or OE) development environment and/or tool suite as follows: (Design Description Document)
    A. Include the development environment and/or tool suite name, purpose, developer, version, and configuration specifics (e.g., ISE Design Suite System, Xilinx, 14.4, EDK and SDK; MATLAB® Simulink®, Model base design support automatic code generation, MathWorks, R2016a).
    B. Include a description of the hardware on which the development environment and/or tool suite is executed, its OS, OS developer, OS version, and OS configuration specifics (e.g., Microsoft® Windows 7, Service pack 2; Linux® Ubuntu, (Xenial Xerus) 16.04).
    C. Include a description of the output of the development environment and/or tool suite, its STRS infrastructure/OE description, developer, version, and unique implementation items (e.g., type of file, .mdl, .slx; GRC's STRS Reference Implementation; Intellectual Property (IP) generated from Xilinx).
    D. Include a description of licensing agreements for development environment and/or tool suite.
(7) Test plans, procedures, and results documentation. (Verification and Validation (V&V) Plan, V&V Procedure, and V&V Results)
(8) Identification of software development standards used. (Version Description Document (VDD)/Metadata)
(9) Version of this NASA Technical Standard used. (VDD/Metadata)
(10) Information, along with supporting documentation, required to make the appropriate decisions regarding ownership, distribution rights, and release (technology transfer) of the application or OE and associated artifacts. (Transfer Rights/Agreements)

(11) Version Description Document, if available, or other document containing the version numbers of each separable artifact in the release, defined down to the lowest level components. (VDD)
(12) Documentation of the platform component hardware used by the application or OE, its function, and the interconnections. If the component executes an operating system, document the OS, OS developer, OS version, and OS configuration. (HID)
(13) Documentation when an OE is submitted to the STRS Application Repository, providing guidelines to aid a waveform/application developer and integrator in the task of developing an STRS compliant waveform/application. (OE-Specific Developer's Guide)

Rationale	To understand how to use the radio, information must be provided about its design and implementation. To have confidence that the design and implementation meet the NASA Procedural Requirements (NPR) 7150.2, NASA Software Engineering Requirements, additional information about standards, reviews, and tests must be provided.
Category	Portability
Traced-from	4.2, 4.9, 5.2
Use Case	None
Applicable to	Application or OE developer
Notes	See also STRS-92.
Verification Method	Inspection of deliverable items and documentation.

7.13 STRS-13 OE Controls Signal-Processing Module (SPM)

Requirement	If the STRS application has a component resident outside the GPM (e.g., in configurable hardware design), then the component shall be controllable from the STRS OE.
Rationale	The layering of the architecture introduces the need for the GPP to be able to control, configure, and monitor many aspects of the SPM. For portability, waveform applications use STRS APIs, which access the HAL or POSIX® API within the STRS OE as needed.
Category	Portability, Reconfigurability, Adaptability
Traced-from	4.5, 4.9, 5.1, 5.4, 5.22
Use Case	None
Applicable to	Application developer
Notes	None
Verification Method	Observation of operation of radio.

7.14 STRS-14 Provide Platform-Specific Wrapper

Requirement	The STRS 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 STRS application developer. This may be a complete abstraction of the actual FPGA pinouts with only waveform application signal names provided.
Rationale	To aid in the portability of waveform applications within an FPGA, a platform-specific wrapper provides an additional layer separating the interface between the GPP and SPM/FPGA from the signal processing functionality within the FPGA.
Category	Portability, Extensibility
Traced-from	4.2, 4.4, 4.9, 5.1, 5.8
Use Case	None
Applicable to	Platform provider
Notes	None
Verification Method	Inspection of document and code.

7.15 STRS-15 Document Platform-Specific Wrapper

Requirement	The STRS 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 noninterface functionality.
Rationale	When functions, interfaces, components, and/or design rules are defined and published, the architecture is open. Open architecture facilitates interoperability among commercial and government developers and minimizes the operational impact of upgrading hardware and software components.
Category	Portability, Adaptability
Traced-from	4.2, 4.4, 4.7, 4.8, 4.9, 5.1, 5.2

Use Case	None
Applicable to	Platform provider
Notes	None
Verification Method	Inspection of document.

7.16 STRS-16 Use C/C++ Waveform (WF) Interface

Requirement	The STRS Application-provided Application Control API shall be implemented using ISO/IEC C or C++.
Rationale	Because portability is a basic goal but middleware is not required, a totally language-independent solution was not available. The lowest common denominator turns out to be a C or C++ language interface. Using a standard ISO/IEC 9899 C or ISO/IEC 14882 C++ aids portability. The year is not included in the requirement, so that obsolete compilers are not mandated.
Category	Portability, Scalability
Traced-from	4.1, 4.2, 4.7, 4.9, 5.1
Use Case	None
Applicable to	Application developer
Notes	None
Verification Method	Inspection of code.

7.17 STRS-17 OE Uses STRS Application Control API

Requirement	The STRS infrastructure shall use the STRS Application-provided Application Control API to control STRS applications.
Rationale	Layering of the architecture separates and encapsulates functionality so that the parts are less influenced by changes to the other. This separation of functionality promotes portability.
Category	Portability
Traced-from	4.1, 4.2, 4.7, 4.9, 5.1
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The STRS Application-provided Application Control API refers to the API defined in STRS-29 through STRS-39, STRS-114 through STRS-116, and the corresponding tables 5 through 18. The method names in the STRS Application-provided Application Control API begin with “APP_”.
Verification Method	Inspection using OE script and using compliance tool.

7.18 STRS-18 Use C/C++ Compile-Time

Requirement	The STRS OE shall support ISO/IEC C or C++, or both, language interfaces for the STRS Application-provided Application Control API at compile-time.
Rationale	Because portability is a basic goal but middleware is not required, a totally language-independent solution was not available. The lowest common denominator turns out to be a C or C++, or both, language interface. Using a standard ISO/IEC 9899 C or ISO/IEC 14882 C++ aids portability. The year is not included in the requirement, so that obsolete compilers are not mandated.
Category	Portability
Traced-from	4.1, 4.2, 4.7, 4.9, 5.1
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	None
Verification Method	Inspection

7.19 STRS-19 Use C/C++ Run-Time

Requirement	The STRS OE shall support ISO/IEC C or C++, or both, language interfaces for the STRS Application-provided Application Control API at run-time.
Rationale	Because portability is a basic goal but middleware is not required, a totally language-independent solution was not available. The lowest common denominator turns out to be a C or C++, or both, language interface. Using a standard ISO/IEC 9899 C or ISO/IEC 14882 C++ aids portability. The year is not included in the requirement, so that obsolete compilers are not mandated.
Category	Portability
Traced-from	4.1, 4.2, 4.7, 4.9, 5.1
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	None
Verification Method	Inspection using STRS compliance tool.

7.20 STRS-20 Include STRS_ApplicationControl.h

Requirement	Each STRS application shall contain: *#include "STRS_ApplicationControl.h".*
Rationale	For portability, standard names are defined for various constants, data types, and method prototypes in the API.
Category	Portability
Traced-from	4.9, 5.1, 5.2

Use Case	None
Applicable to	Application developer
Notes	None
Verification Method	Inspection using STRS compliance tool.

7.21 STRS-21 Provide STRS_ApplicationControl.h

Requirement	The STRS platform provider shall provide an "STRS_ApplicationControl.h" that contains the method prototypes for each STRS application and, for C++, the class definition for the base class STRS_ApplicationControl.
Rationale	For portability, standard names are defined for various constants, data types, and method prototypes in the API.
Category	Portability
Traced-from	4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	None
Verification Method	Inspection

7.22 STRS-22 STRS_ApplicationControl Base Class

Requirement	If the STRS Application-provided Application Control API is implemented in C++, the STRS application class shall be derived from the STRS_ApplicationControl base class.
Rationale	For portability, standard names are defined for various constants, data types, and method prototypes in the API.
Category	Portability
Traced-from	4.9, 5.1, 5.2
Use Case	None
Applicable to	Application developer
Notes	None
Verification Method	Inspection

Example	In C++, a MyWaveform.h file should contain a class definition of the form: `class MyWaveform: public STRS_ApplicationControl {…};`

7.23 STRS-23 Include STRS_Sink.h

Requirement	If the STRS application provides the APP_Write method, the STRS application shall contain: *#include "STRS_Sink.h"*.
Rationale	For portability, standard names are defined for various constants, data types, and method prototypes in the API.
Category	Portability
Traced-from	4.9, 5.1, 5.2
Use Case	None
Applicable to	Application developer
Notes	None
Verification Method	Inspection using STRS compliance tool.

7.24 STRS-24 Provide STRS_Sink.h

Requirement	The STRS platform provider shall provide an “STRS_Sink.h” that contains the method prototypes for APP_Write and, for C++, the class definition for the base class STRS_Sink.
Rationale	For portability, standard names are defined for various constants, data types, and method prototypes in the API.
Category	Portability
Traced-from	4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	None
Verification Method	Inspection

7.25 STRS-25 STRS_Sink Base Class

Requirement	If the STRS Application-provided Application Control API is implemented in C++ and the STRS application provides the APP_Write method, the STRS application class shall be derived from the STRS_Sink base class.
Rationale	For portability, standard names are defined for various constants, data types, and method prototypes in the API.

Category	Portability
Traced-from	4.9, 5.1, 5.2
Use Case	None
Applicable to	Application developer
Notes	None
Verification Method	Inspection
Example	In C++, a MyWaveform.h file should contain a class definition of the form:

```
class MyWaveform:  public STRS_ApplicationControl,
                   public STRS_Sink
{…};
```

7.26 STRS-26 Include STRS_Source.h

Requirement	If the STRS application provides the APP_Read method, the STRS application shall contain: *#include "STRS_Source.h"*.
Rationale	For portability, standard names are defined for various constants, data types, and method prototypes in the API.
Category	Portability
Traced-from	4.9, 5.1, 5.2
Use Case	None
Applicable to	Application developer
Notes	None
Verification Method	Inspection using STRS compliance tool.

7.27 STRS-27 Provide STRS_Source.h

Requirement	The STRS platform provider shall provide an “STRS_Source.h” that contains the method prototypes for APP_Read and, for C++, the class definition for the base class STRS_Source.
Rationale	For portability, standard names are defined for various constants, data types, and method prototypes in the API.
Category	Portability
Traced-from	4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	None
Verification Method	Inspection

7.28 STRS-28 STRS_Source Base Class

Requirement	If the STRS Application-provided Application Control API is implemented in C++ and the STRS application provides the APP_Read method, the STRS application class shall be derived from the STRS_Source base class.
Rationale	For portability, standard names are defined for various constants, data types, and method prototypes in the API.
Category	Portability
Traced-from	4.9, 5.1, 5.2
Use Case	None
Applicable to	Application developer
Notes	None
Verification Method	Inspection
Example	In C++, the MyWaveform.h file should contain a class definition of the form

```
class MyWaveform:   public STRS_ApplicationControl,
                    public STRS_Source
{…};
```

If both APP_Read and APP_Write are provided in the same waveform, the C++ class will be derived from all three base classes named in requirements (STRS-22, STRS-25, and STRS-28). For example, the MyWaveform.h file should contain a class definition of the form

```
class MyWaveform:   public STRS_ApplicationControl,
                    public STRS_Sink,
                    public STRS_Source
{…};
```

7.29 STRS-29 APP_Configure

Requirement	Each STRS application shall contain a callable APP_Configure method as described in table 5, APP_Configure().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to implement and use standard interfaces across all platforms. In addition, APP_Configure was patterned after the configure method in the PropertySet interface in JTRS/SCA and OMG/SWRADIO.
Category	Portability, Extensibility
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.7, 4.8, 4.9, 5.1, 5.2, 5.3
Use Case	Set Waveform Parameter
Applicable to	Application developer

Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	STRS_Configure
Example	

```
STRS_Result APP_Configure(STRS_Property_Name name,
                          STRS_Property_Value *value,
                          STRS_Buffer_Size nb) {
    STRS_Result rtn = STRS_OK;
    if (strcmp("A", name)==0  &&  nb <=  maxLa){
         strncpy(a, value, nb);
    } else
      if (strcmp("B",name)==0  &&  nb <= maxLb){
         if (myState == STRS_APP_RUNNING) {
            rtn = STRS_WARNING;
         } else {
            strncpy(b, value, nb);
            rtn = strlen(b);
         }
      } else {
           rtn = STRS_WARNING;
      }
    return rtn;
}
```

7.30 STRS-30 APP_GroundTest

Requirement	Each STRS application shall contain a callable APP_GroundTest method as described in table 9, APP_GroundTest().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to implement and use standard interfaces across all platforms. In addition, APP_GroundTest was patterned after the runTest method in the TestableObject interface in JTRS/SCA and OMG/SWRADIO. It performs system and unit testing usually done before deployment.
Category	Portability, Extensibility
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.7, 4.8, 4.9, 5.1, 5.2, 5.3, 5.15
Use Case	Built-In Test
Applicable to	Application developer
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	STRS_GroundTest

Example

```
STRS_Result APP_GroundTest(STRS_TestID testID) {
    STRS_HandleID fromWF = APP_GetHandleID();
    if (testID == 0) {
       …
       return STRS_OK;
    } else {
       STRS_Buffer_Size nb = strlen(
          "Invalid APP_GroundTest argument.");
       STRS_Log(fromWF, STRS_ERROR_QUEUE,
          "Invalid APP_GroundTest argument.", nb);
       return STRS_ERROR;
    }
}
```

7.31 STRS-31 APP_Initialize

Requirement	Each STRS application shall contain a callable APP_Initialize method as described in table 10, APP_Initialize().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to implement and use standard interfaces across all platforms. In addition, APP_Initialize was patterned after the initialize method in the LifeCycle interface in JTRS/SCA and OMG/SWRADIO.
Category	Portability, Extensibility
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.7, 4.8, 4.9, 5.1, 5.2, 5.3
Use Case	Waveform Instantiation
Applicable to	Application developer
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	STRS_Initialize

Example

```
STRS_Result APP_Initialize() {
    STRS_HandleID fromWF = APP_GetHandleID();
    if (myState == STRS_APP_RUNNING) {
        STRS_Buffer_Size nb = strlen(
           "Can't Init when STRS_APP_RUNNING.");
        STRS_Log(fromWF,STRS_WARNING_QUEUE,
           "Can't Init when STRS_APP_RUNNING.", nb);
        return STRS_WARNING;
    } else {
       …
       myState = STRS_APP_STOPPED;
    }
    return STRS_OK;
 }
```

7.32 STRS-32 APP_Instance

Requirement	Each STRS application shall contain a callable APP_Instance method as described in table 11, APP_Instance().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to implement and use standard interfaces across all platforms.
Category	Portability, Extensibility
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.7, 4.8, 4.9, 5.1, 5.2, 5.3
Use Case	Waveform Instantiation
Applicable to	Application developer
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	N/A

Example for C++

```
STRS_Instance *ThisSTRSApplication::APP_Instance(
            STRS_HandleID handleID, char *name) {
   return new ThisSTRSApplication(handleID,name);
}
```

Example for C

```
char handleName[nMax];
STRS_Instance *APP_Instance(
          STRS_HandleID handleID, char *name) {
   myQ = handleID;
   strncpy(handleName, name, nMax);
   myState = STRS_APP_INSTANTIATED;
   OE_DEFINED_MACRO_TO_SET_INST();
   return inst;
}
```

7.33 STRS-33 APP_Query

Requirement	Each STRS application shall contain a callable APP_Query method as described in table 12, APP_Query().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to implement and use standard interfaces across all platforms. In addition, APP_Query was patterned after the query method in the PropertySet interface in JTRS/SCA and OMG/SWRADIO.
Category	Portability, Extensibility
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.7, 4.8, 4.9, 5.1, 5.2, 5.3
Use Case	Get Waveform Parameter
Applicable to	Application developer
Notes	The table in the requirement is in NASA-STD-4009A.

Verification Method	Inspection using STRS compliance tool.
See Also	STRS_Query
Example	`STRS_Result APP_Query(STRS_Property_Name name, STRS_Property_Value *value, STRS_Buffer_Size nb) { if (strcmp("A",name)==0) { /* Variable “a” is declared as a * character string, and typically * contains a value set by APP_Configure. */ if (a == NULL || strlen(a) >= nb) { rtn = STRS_ERROR; } else { strncpy(value, a, nb); rtn = strlen(value); } } return rtn; }`

7.34 STRS-34 APP_Read

Requirement	If the STRS application provides data to the infrastructure, then the STRS application shall contain a callable APP_Read method as described in table 13, APP_Read().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to implement and use standard interfaces across all platforms.
Category	Portability, Extensibility
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.7, 4.8, 4.9, 5.1, 5.2, 5.3
Use Case	Transmit a Packet
Applicable to	Application developer
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	STRS_Read
Example	`STRS_Result APP_Read(STRS_Message buffer, STRS_Buffer_Size nb) { if (nb <= 4) return STRS_ERROR; strcpy (buffer,"ABCD"); return strlen(buffer); }`

7.35 STRS-35 APP_ReleaseObject

Requirement	Each STRS application shall contain a callable APP_ReleaseObject method as described in table 14, APP_ReleaseObject().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to implement and use standard interfaces across all platforms. In addition, APP_ReleaseObject was patterned after the releaseObject method in the LifeCycle interface in JTRS/SCA and OMG/SWRADIO.
Category	Portability, Extensibility
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.7, 4.8, 4.9, 5.1, 5.2, 5.3
Use Case	Waveform Deallocation
Applicable to	Application developer
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	STRS_ReleaseObject
Example	

```
STRS_Result APP_ReleaseObject() {
      STRS_HandleID fromWF = APP_GetHandleID();
      if (myState == STRS_APP_RUNNING) {
         STRS_Buffer_Size nb = strlen(
             "Can't free resources when RUNNING.");
         STRS_Log(fromWF,STRS_WARNING_QUEUE,
             "Can't free resources when RUNNING.",
             nb);
          return STRS_WARNING;
      } else {
      …
      }
      return STRS_OK;
}
```

7.36 STRS-36 APP_RunTest

Requirement	Each STRS application shall contain a callable APP_RunTest method as described in table 15, APP_RunTest().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to implement and use standard interfaces across all platforms. In addition, APP_RunTest was patterned after the runTest method in the TestableObject interface in JTRS/SCA and OMG/SWRADIO. It performs system and unit testing usually done after deployment.
Category	Portability, Extensibility

Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.7, 4.8, 4.9, 5.1, 5.2, 5.3, 5.15
Use Case	Built-In Test
Applicable to	Application developer
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	STRS_RunTest
Example	

```
STRS_Result APP_RunTest(STRS testID) {

    if (testID == 1 ) {
       …
     } else {
        STRS_HandleID fromWF = APP_GetHandleID();
        STRS_Buffer_Size nb = strlen(
           "Invalid APP_RunTest argument testID.");
        STRS_Log(fromWF, STRS_ERROR_QUEUE,
           "Invalid APP_RunTest argument testID.",
           nb);
        return STRS_ERROR;
     }
     return STRS_OK;
}
```

7.37 STRS-37 APP_Start

Requirement	Each STRS application shall contain a callable APP_Start method as described in table 16, APP_Start().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to implement and use standard interfaces across all platforms. In addition, APP_Start was patterned after the start method in the Resource interface in JTRS/SCA and ControllableComponent interface in OMG/SWRADIO.
Category	Portability, Extensibility
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.7, 4.8, 4.9, 5.1, 5.2, 5.3
Use Case	Waveform Start
Applicable to	Application developer
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	STRS_Start

Example

```
STRS_Result APP_Start() {
      if (myState == STRS_APP_STOPPED) {
            …
            myState = STRS_APP_RUNNING;
            …
      } else {
            return STRS_ERROR;
      }
      return STRS_OK;
}
```

7.38 STRS-38 APP_Stop

Requirement	Each STRS application shall contain a callable APP_Stop method as described in table 17, APP_Stop().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to implement and use standard interfaces across all platforms. In addition, APP_Stop was patterned after the stop method in the Resource interface in JTRS/SCA and ControllableComponent interface in OMG/SWRADIO.
Category	Portability, Extensibility
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.7, 4.8, 4.9, 5.1, 5.2, 5.3
Use Case	Waveform Stop
Applicable to	Application developer
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	STRS_Stop

Example

```
STRS_Result APP_Stop() {
      if (myState == STRS_APP_RUNNING) {
            ...
            myState = STRS_APP_STOPPED;
            ...
      } else {
            return STRS_ERROR;
      }
      return STRS_OK;
}
```

7.39 STRS-39 APP_Write

Requirement	If the STRS application receives data from the infrastructure, then the STRS application shall contain a callable APP_Write method as described in table 18, APP_Write().

Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to implement and use standard interfaces across all platforms.
Category	Portability, Extensibility
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.7, 4.8, 4.9, 5.1, 5.2, 5.3
Use Case	Receive a Packet
Applicable to	Application developer
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	STRS_Write
Example	`STRS_Result APP_Write(STRS_Message buffer, STRS_Buffer_Size nb) { /* Data in buffer is character data. */ if (strlen(buffer) != nb -1) return STRS_ERROR; int nco = fprintf(stdout,"%s\n",buffer); return (STRS_Result) nco; }`

7.40 STRS-40 STRS_Configure

Requirement	The STRS infrastructure shall contain a callable STRS_Configure method as described in table 19, STRS_Configure().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to implement and use standard interfaces across all platforms. The signature of the infrastructure method is different from the signature of the corresponding application method because there has to be a C language interface to the infrastructure method and it has to contain additional information that allows the infrastructure to determine whether the target component is C or C++ and call the corresponding application method appropriately.
Category	Portability, Extensibility
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.7, 4.8, 4.9, 5.1, 5.2
Use Case	Set Waveform Parameter
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	APP_Configure

Example

```
/* Set A=5, B=27. */
STRS_Result rtn;
STRS_HandleID fromWF = APP_GetHandleID();
Do while (true) {
    rtn = STRS_Configure(fromWF,toWF, "A", "5", 1);
    if ( ! STRS_IsOK(rtn)) break;
    rtn = STRS_Configure(fromWF,toWF, "B","27", 2);
    if ( ! STRS_IsOK(rtn)) break;
     break;
}
if ( ! STRS_IsOK(rtn)) {
   STRS_Buffer_Size nb = strlen(
      "STRS_Configure fails.");
    STRS_Log(fromWF, STRS_ERROR_QUEUE,
      "STRS_Configure fails.", nb);
}
```

7.41 STRS-41 STRS_GroundTest

Requirement	The STRS infrastructure shall contain a callable STRS_GroundTest method as described in table 20, STRS_GroundTest().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to implement and use standard interfaces across all platforms. The signature of the infrastructure method is different from the signature of the corresponding application method because there has to be a C language interface to the infrastructure method and it has to contain additional information that allows the infrastructure to determine whether the target component is C or C++ and call the corresponding application method appropriately. It performs system and unit testing usually done before deployment.
Category	Portability, Extensibility
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.7, 4.8, 4.9, 5.1, 5.2, 5.15
Use Case	Built-In Test
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	APP_GroundTest

Example

```
STRS_HandleID fromWF = APP_GetHandleID();
STRS_Result rtn =
   STRS_GroundTest(fromWF,toWF,testID);
if ( ! STRS_IsOK(rtn)) {
   STRS_Buffer_Size nb = strlen(
      "GroundTest fails.");
   STRS_Log(fromWF, STRS_ERROR_QUEUE,
      "GroundTest fails.", nb);
}
```

7.42 STRS-42 STRS_Initialize

Requirement	The STRS infrastructure shall contain a callable STRS_Initialize method as described in table 21, STRS_Initialize().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to implement and use standard interfaces across all platforms. The signature of the infrastructure method is different from the signature of the corresponding application method because there has to be a C language interface to the infrastructure method and it has to contain additional information that allows the infrastructure to determine whether the target component is C or C++ and call the corresponding application method appropriately.
Category	Portability, Extensibility
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.7, 4.8, 4.9, 5.1, 5.2
Use Case	Waveform Instantiation
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	APP_Initialize

Example

```
STRS_HandleID fromWF = APP_GetHandleID();
STRS_Result rtn = STRS_Initialize(fromWF,toWF);
if ( ! STRS_IsOK(rtn)) {
   STRS_Buffer_Size nb = strlen(
      "STRS_Initialize fails.");
   STRS_Log(fromWF, STRS_ERROR_QUEUE,
      "STRS_Initialize fails.", nb);
}
```

7.43 STRS-43 STRS_Query

Requirement	The STRS infrastructure shall contain a callable STRS_Query method as described in table 22, STRS_Query().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform

	applications and services have to implement and use standard interfaces across all platforms. The signature of the infrastructure method is different from the signature of the corresponding application method because there has to be a C language interface to the infrastructure method and it has to contain additional information that allows the infrastructure to determine whether the target component is C or C++ and call the corresponding application method appropriately.
Category	Portability, Extensibility
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.7, 4.8, 4.9, 5.1, 5.2
Use Case	Get Waveform Parameter
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	APP_Query

Example	`STRS_Result rtn;`
	`STRS_HandleID fromWF = APP_GetHandleID();`
	`Do while (true) {`
	`  rtn = STRS_Query(fromWF,toWF,”A”, a, maxLa);`
	`  if ( ! STRS_IsOK(rtn)) break;`
	
	`  rtn = STRS_Query(fromWF,toWF,”B”, b, maxLb);`
	`  if ( ! STRS_IsOK(rtn)) break;`
	`  break;`
	`}`
	`if ( ! STRS_IsOK(rtn)) {`
	`  STRS_Buffer_Size nb = strlen(`
	`  "STRS_Query fails.");`
	`  STRS_Log(fromWF, STRS_ERROR_QUEUE,`
	`  "STRS_Query fails.", nb);`
	`}`
	`cout << “A = “ << a << std::endl;`
	`cout << “B = “ << b << std::endl;`

7.44 STRS-44 STRS_ReleaseObject

Requirement	The STRS infrastructure shall contain a callable STRS_ReleaseObject method as described in table 23, STRS_ReleaseObject().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to implement and use standard interfaces across all platforms. The signature of the infrastructure method is different from the signature of the corresponding application method because there has to be a C language interface to the infrastructure method and it has to contain additional information that allows the infrastructure to determine whether the target component is C or C++ and call the corresponding application method appropriately.

Category	Portability, Extensibility
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.7, 4.8, 4.9, 5.1, 5.2
Use Case	Get Waveform Parameter
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	APP_ReleaseObject

Example	`STRS_HandleID fromWF = APP_GetHandleID();`
	`STRS_Result rtn =`
	`  STRS_ReleaseObject(fromWF,toWF);`
	`if ( ! STRS_IsOK(rtn)) {`
	`  STRS_Buffer_Size nb = strlen(`
	`  "STRS_ReleaseObject fails.");`
	`  STRS_Log(fromWF, STRS_ERROR_QUEUE,`
	`  "STRS_ReleaseObject fails.", nb);`
	`}`

7.45 STRS-45 STRS_RunTest

Requirement	The STRS infrastructure shall contain a callable STRS_RunTest method as described in table 24, STRS_RunTest().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to implement and use standard interfaces across all platforms. The signature of the infrastructure method is different from the signature of the corresponding application method because there has to be a C language interface to the infrastructure method and it has to contain additional information that allows the infrastructure to determine whether the target component is C or C++ and call the corresponding application method appropriately. It performs system and unit testing usually done after deployment.
Category	Portability, Extensibility
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.7, 4.8, 4.9, 5.1, 5.2, 5.15
Use Case	Built-In Test
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	APP_RunTest

Example	`STRS_HandleID fromWF = APP_GetHandleID();`
	`STRS_Result rtn =`
	`  STRS_RunTest(fromWF,toWF,1);`
	`if ( ! STRS_IsOK(rtn)) {`
	`  STRS_Buffer_Size nb = strlen(`
	`  "STRS_RunTest fails.");`
	`  STRS_Log(fromWF, STRS_ERROR_QUEUE,`
	`  "STRS_RunTest fails.”, nb);`
	`}`

7.46 STRS-46 STRS_Start

Requirement	The STRS infrastructure shall contain a callable STRS_Start method as described in table 25, STRS_Start().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to implement and use standard interfaces across all platforms. The signature of the infrastructure method is different from the signature of the corresponding application method because there has to be a C language interface to the infrastructure method and it has to contain additional information that allows the infrastructure to determine whether the target component is C or C++ and call the corresponding application method appropriately.
Category	Portability, Extensibility
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.7, 4.8, 4.9, 5.1, 5.2
Use Case	Waveform Start
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	APP_Start

Example	`STRS_HandleID fromWF = APP_GetHandleID();`
	`STRS_Result rtn = STRS_Start(fromWF,toWF);`
	`if ( ! STRS_IsOK(rtn)) {`
	`  STRS_Buffer_Size nb = strlen(`
	`  "STRS_Start fails.");`
	`  STRS_Log(fromWF, STRS_ERROR_QUEUE,`
	`  "STRS_Start fails.", nb);`
	`}`

7.47 STRS-47 STRS_Stop

Requirement	The STRS infrastructure shall contain a callable STRS_Stop method as described in table 26, STRS_Stop().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform

	applications and services have to implement and use standard interfaces across all platforms. The signature of the infrastructure method is different from the signature of the corresponding application method because there has to be a C language interface to the infrastructure method and it has to contain additional information that allows the infrastructure to determine whether the target component is C or C++ and call the corresponding application method appropriately.
Category	Portability, Extensibility
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.7, 4.8, 4.9, 5.1, 5.2
Use Case	Waveform Start
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	APP_Stop

Example	`STRS_HandleID fromWF = APP_GetHandleID();`
	`STRS_Result rtn = STRS_Stop(fromWF,toWF);`
	`if ( ! STRS_IsOK(rtn)) {`
	`  STRS_Buffer_Size nb = strlen(`
	`  "STRS_Stop fails.");`
	`  STRS_Log(fromWF, STRS_ERROR_QUEUE,`
	`  "STRS_Stop fails.", nb);`
	`}`

7.48 STRS-48 STRS_AbortApp

Requirement	The STRS infrastructure shall contain a callable STRS_AbortApp method as described in table 27, STRS_AbortApp().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to implement and use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.7, 4.8, 4.9, 5.1, 5.2
Use Case	Waveform Abort
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.

Example

```
STRS_HandleID fromWF = APP_GetHandleID();
STRS_Result rtn = STRS_AbortApp(fromWF,toWF);
if ( ! STRS_IsOK(rtn)) {
   STRS_Buffer_Size nb = strlen(
      "AbortApp fails.");
   STRS_Log(fromWF, STRS_ERROR_QUEUE,
      "AbortApp fails.", nb);
}
```

7.49 STRS-49 STRS_GetErrorQueue

Requirement	The STRS infrastructure shall contain a callable STRS_GetErrorQueue method as described in table 28, STRS_GetErrorQueue().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	STRS_IsOK

Example

```
STRS_HandleID fromWF = APP_GetHandleID();
STRS_Result rtn = STRS_AbortApp(fromWF,toWF);
if ( ! STRS_IsOK(rtn)){
   STRS_Buffer_Size nb = strlen(
      "AbortApp fails.");
   STRS_Log(fromWF, STRS_GetErrorQueue(rtn),
      "AbortApp fails.", nb);
}
```

7.50 STRS-50 STRS_HandleRequest

Requirement	The STRS infrastructure shall contain a callable STRS_HandleRequest method as described in table 30, STRS_HandleRequest().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2

Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	STRS_ValidateHandleID, STRS_GetHandleName
Example	

```
STRS_HandleID fromWF = APP_GetHandleID();
char toResourceName[] = "WF1";
STRS_HandleID toID = STRS_HandleRequest(fromWF,
   toResourceName);
STRS_Result rtn = STRS_ValidateHandleID(toID);
if ( ! STRS_IsOK(rtn)) {
   STRS_HandleID errQ = STRS_GetErrorQueue(rtn);
   STRS_Buffer_Size nb = strlen(
           "Did not find handle ID.");
   STRS_Log(fromWF,errQ,
           "Did not find handle ID.", nb);
} else {
     std::cout << "Found Handle for "
               << toResourceName << ": "
               << toID << std::endl;
}
```

7.51 STRS-51 STRS_InstantiateApp

Requirement	The STRS infrastructure shall contain a callable STRS_InstantiateApp method as described in table 31, STRS_InstantiateApp().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	Waveform Instantiation
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.

Example

```
char toWF[STRS_MAX_PATH_LENGTH];
char handleName[STRS_MAX_HANDLE_NAME_LENGTH];
STRS_HandleID fromWF = APP_GetHandleID();
// OE-specific string defines a configuration file.
strcpy(toWF,"/path/STRS_WFxxx.cfg");
strcpy(handleName,”WFxxx”);
STRS_HandleID wfID =
   STRS_InstantiateApp(fromWF,handleName,toWF);
STRS_Result rtn = STRS_ValidateHandleID(wfID);
if ( ! STRS_IsOK(rtn)) {
   STRS_Buffer_Size nb = strlen(
      "InstantiateApp fails.");
    STRS_Log(fromWF, STRS_ERROR_QUEUE,
      "InstantiateApp fails.", nb);
}
```

7.52 STRS-52 STRS_IsOK

Requirement	The STRS infrastructure shall contain a callable STRS_IsOK method as described in table 32, STRS_IsOK().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	STRS_GetErrorQueue

Example

```
char toWF[MAX_PATH_LENGTH];
char handleName[STRS_MAX_HANDLE_NAME_LENGTH];
STRS_HandleID fromWF = APP_GetHandleID();
strcpy(toWF,"/path/STRS_WFxxx.cfg");
strcpy(handleName,”WFxxx”);
STRS_HandleID wfID =
   STRS_InstantiateApp(fromWF,handleName,toWF);
STRS_Result rtn = STRS_ValidateHandleID(wfID);
if ( ! STRS_IsOK(rtn) {
   STRS_Buffer_Size nb = strlen(
      "InstantiateApp fails.");
   STRS_Log(fromWF, STRS_GetErrorQueue(wfID),
      "InstantiateApp fails.", nb);
}
```

7.53 STRS-53 STRS_Log

Requirement	The STRS infrastructure shall contain a callable STRS_Log method as described in table 33, STRS_Log().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	Fault Management
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	See STRS_RunTest or APP_RunTest for further examples.

Example	`STRS_HandleID fromWF = APP_GetHandleID();`
	`STRS_Buffer_Size nb =`
	`  strlen("file does not exist.");`
	`STRS_Log(fromWF,STRS_ERROR_QUEUE,`
	`  "file does not exist.", nb);`
	`// This could produce a line something like:`
	`//  19700101000000;WF1,ERROR,file does not exist.`

7.54 STRS-54 STRS_Log Error

Requirement	When an STRS application has a nonfatal error, the STRS application shall use the callable STRS_Log method as described in table 33, STRS_Log(), with a target handle ID of constant STRS_ERROR_QUEUE.
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	Fault Management
Applicable to	Application developer
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection

7.55 STRS-55 STRS_Log Fatal

Requirement	When an STRS application has a fatal error, the STRS application shall use the callable STRS_Log method as described in table 33, STRS_Log(), with a target handle ID of constant STRS_FATAL_QUEUE.
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Applicable to	Application developer
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection

7.56 STRS-56 STRS_Log Warning

Requirement	When an STRS application has a warning condition, the STRS application shall use the callable STRS_Log method as described in table 33, STRS_Log(), with a target handle ID of constant STRS_WARNING_QUEUE.
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	Fault Management
Applicable to	Application developer
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection

7.57 STRS-57 STRS_Log Telemetry

Requirement	When an STRS application needs to send telemetry, the STRS application shall use the callable STRS_Log method as described in table 33, STRS_Log(), with a target handle ID of constant STRS_TELEMETRY_QUEUE.
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2

Use Case	None
Applicable to	Application developer
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection

7.58 STRS-58 STRS_Write

Requirement	The STRS infrastructure shall contain a callable STRS_Write method as described in table 36, STRS_Write().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability, Extensibility
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	Receive a Packet
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	APP_Write

Example	`STRS_HandleID fromWF = APP_GetHandleID();`
	`char buffer[32];`
	`strcpy(buffer,"ABCDE");`
	`STRS_Buffer_Size nb = strlen(buffer);`
	`STRS_Result rtn =`
	`  STRS_Write(fromWF,toID,buffer,nb);`

7.59 STRS-59 STRS_Read

Requirement	The STRS infrastructure shall contain a callable STRS_Read method as described in table 37, STRS_Read().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability, Extensibility
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	Transmit a Packet
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	APP_Read

Example

```
STRS_HandleID fromWF = APP_GetHandleID();
char buffer[32];
STRS_Buffer_Size nb = 32;
STRS_Result rtn =
     STRS_Read(fromWF,pullID,buffer,nb);
```

7.60 STRS-60 Device Control (Deleted)

Rationale: This requirement was deleted because when STRS-10 wording was made more robust, STRS-60 became superfluous. STRS-60 previously stated: The STRS applications shall use the methods in the STRS infrastructure Device Control API, STRS infrastructure-provided Application Control API, Infrastructure Data Source API (if appropriate), and Infrastructure Data Sink API (if appropriate) to control the STRS Devices.

7.61 STRS-61 STRS_DeviceClose

Requirement: The STRS infrastructure shall contain a callable STRS_DeviceClose method as described in table 38, STRS_DeviceClose().

Rationale: For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.

Category: Portability

Traced-from: 4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2

Use Case: None

Applicable to: OE developer: usually platform provider

Notes: The table in the requirement is in NASA-STD-4009A.

Verification Method: Inspection using STRS compliance tool.

Example

```
STRS_HandleID fromWF = APP_GetHandleID();
STRS_Result rtn =
   STRS_DeviceClose(fromWF,toDev);
if ( ! STRS_IsOK(rtn)) {
   STRS_Buffer_Size nb = strlen(
      "DeviceClose fails.");
   STRS_Log(fromWF, STRS_ERROR_QUEUE,
      "DeviceClose fails.", nb);
}
```

7.62 STRS-62 STRS_DeviceFlush

Requirement	The STRS infrastructure shall contain a callable STRS_DeviceFlush method as described in table 39, STRS_DeviceFlush().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.

Example	`STRS_HandleID fromWF = APP_GetHandleID();`
	`STRS_Result rtn =`
	`  STRS_DeviceFlush(fromWF,toDev);`
	`if ( ! STRS_IsOK(rtn)) {`
	`  STRS_Buffer_Size nb = strlen(`
	`  "DeviceFlush fails.");`
	`  STRS_Log(fromWF, STRS_ERROR_QUEUE,`
	`  "DeviceFlush fails.", nb);`
	`}`

7.63 STRS-63 STRS_DeviceLoad

Requirement	The STRS infrastructure shall contain a callable STRS_DeviceLoad method as described in table 40, STRS_DeviceLoad().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.

Example

```
STRS_HandleID fromWF = APP_GetHandleID();
STRS_Result rtn =
   STRS_DeviceLoad(fromWF,toDev,
      "/path/WF1.FPGA.bit");
if ( ! STRS_IsOK(rtn)) {
   STRS_Buffer_Size nb = strlen(
      "DeviceLoad fails.");
   STRS_Log(fromWF, STRS_ERROR_QUEUE,
      "DeviceLoad fails.", nb);
}
```

7.64 STRS-64 STRS_DeviceOpen

Requirement	The STRS infrastructure shall contain a callable STRS_DeviceOpen method as described in table 41, STRS_DeviceOpen().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.

Example

```
STRS_HandleID fromWF = APP_GetHandleID();
STRS_Result rtn =
   STRS_DeviceOpen(fromWF,toDev);
if ( ! STRS_IsOK(rtn)) {
   STRS_Buffer_Size nb = strlen(
      "DeviceOpen fails.");
   STRS_Log(fromWF, STRS_ERROR_QUEUE,
      "DeviceOpen fails.", nb);
}
```

7.65 STRS-65 STRS_DeviceReset

Requirement	The STRS infrastructure shall contain a callable STRS_DeviceReset method as described in table 42, STRS_DeviceReset().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2

Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.

Example	`STRS_HandleID fromWF = APP_GetHandleID();`
	`STRS_Result rtn =`
	`  STRS_DeviceReset(fromWF,toDev);`
	`if ( ! STRS_IsOK(rtn)) {`
	`  STRS_Buffer_Size nb = strlen(`
	`  "DeviceReset fails.");`
	`  STRS_Log(fromWF, STRS_ERROR_QUEUE,`
	`  "DeviceReset fails.", nb);`
	`}`

7.66 STRS-66 STRS_DeviceStart (Deleted)

Rationale	This requirement was deleted because it is redundant; STRS_Start is used instead. STRS-66 previously stated: The STRS infrastructure shall contain a callable STRS_DeviceStart method.

7.67 STRS-67 STRS_DeviceStop (Deleted)

Rationale	This requirement was deleted because it is redundant; STRS_Stop is used instead. STRS-67 previously stated: The STRS infrastructure shall contain a callable STRS_DeviceStop method.

7.68 STRS-68 STRS_DeviceUnload

Requirement	The STRS infrastructure shall contain a callable STRS_DeviceUnload method as described in table 43, STRS_DeviceUnload().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	Waveform Deallocation
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.

Example	`STRS_HandleID fromWF = APP_GetHandleID();`
	`STRS_Result rtn =`
	`  STRS_DeviceUnload(fromWF,toDev);`

```
if ( ! STRS_IsOK(rtn)) {
   STRS_Buffer_Size nb = strlen(
      "DeviceUnload fails.");
    STRS_Log(fromWF, STRS_ERROR_QUEUE,
      "DeviceUnload fails.", nb);
}
```

7.69 STRS-69 STRS_SetISR

Requirement	The STRS infrastructure shall contain a callable STRS_SetISR method as described in table 44, STRS_SetISR().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A. The callback function whose address is passed to the STRS_SetISR method may be different for different OE's. The OE-specific documentation tells whether there are any arguments and if so, what they are. The most likely argument would be an instance variable obtained from the handle ID with which the function may obtain and/or set instance data.
Verification Method	Inspection using STRS compliance tool.

Example	`STRS_HandleID myQ = APP_GetHandleID();`
	`qnew=myQ;`
	`fp = (STRS_ISR_Function) Test_ISR_Method;`
	`fprintf(stdout,`
	`"Pointer to function Test_ISR_Method: %p\n", fp);`
	`rtn = STRS_SetISR(myQ,qnew,(STRS_ISR_Function) fp);`
	`if ( ! STRS_IsOK(rtn)) {`
	`STRS_Buffer_Size nb = strlen(`
	`“STRS_SetISR fails for Test_ISR_Method.”);`
	`STRS_Log(myQ, STRS_ERROR_QUEUE,`
	`“STRS_SetISR fails for Test_ISR_Method.”,nb);`
	`}`

7.70 STRS-70 STRS_FileClose

Requirement	The STRS infrastructure shall contain a callable STRS_FileClose method as described in table 51, STRS_FileClose().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.

Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	STRS_FileOpen

Example	`STRS_HandleID fromWF = APP_GetHandleID();`
	`STRS_Result rtn = STRS_FileClose(fromWF,toFile);`
	`if ( ! STRS_IsOK(rtn)) {`
	`  STRS_Buffer_Size nb = strlen(`
	`  "FileClose fails.");`
	`  STRS_Log(fromWF, STRS_ERROR_QUEUE,`
	`  "FileClose fails.", nb);`
	`}`

7.71 STRS-71 STRS_FileGetFreeSpace

Requirement	The STRS infrastructure shall contain a callable STRS_FileGetFreeSpace method as described in table 52, STRS_FileGetFreeSpace().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.

Example

```
STRS_HandleID fromWF = APP_GetHandleID();
STRS_File_Size size =
   STRS_FileGetFreeSpace(fromWF,NULL);
if (size < 0) {
   STRS_Buffer_Size nb = strlen(
      "FileGetFreeSpace fails.");
   STRS_Log(fromWF, STRS_ERROR_QUEUE,
       "FileGetFreeSpace fails.", nb);
}
```

7.72 STRS-72 STRS_FileGetSize

Requirement	The STRS infrastructure shall contain a callable STRS_FileGetSize method as described in table 53, STRS_FileGetSize().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.

Example

```
STRS_HandleID fromWF = APP_GetHandleID();
STRS_File_Size size =
   STRS_FileGetSize(fromWF,"/path/WF1.FPGA.bit");
if (size < 0) {
   STRS_Buffer_Size nb = strlen(
       "FileGetSize fails.");
   STRS_Log(fromWF, STRS_ERROR_QUEUE,
       "FileGetSize fails.", nb);
}
```

7.73 STRS-73 STRS_FileGetStreamPointer

Requirement	The STRS infrastructure shall contain a callable STRS_FileGetStreamPointer method as described in table 54, STRS_FileGetStreamPointer().
Rationale	STRS-73 solves the potential problem of I/O methods missing from NASA-STD-4009A. Since not all SDRs will have a file system, this method should be used sparingly with comments describing its purpose.
Category	Extensibility
Traced-from	4.4, 4.5
Use Case	None

Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	STRS_FileOpen

Example	`STRS_HandleID fromWF = APP_GetHandleID();`
	`FILE *fsp =`
	`  STRS_FileGetStreamPointer(fromWF,toFile);`
	`if (fsp == NULL) {`
	`  STRS_Buffer_Size nb = strlen(`
	`  "FileGetStreamPointer fails.");`
	`  STRS_Log(fromWF, STRS_ERROR_QUEUE,`
	`  "FileGetStreamPointer fails.", nb);`
	`} else {`
	`  rewind(fsp);`
	`}`

7.74 STRS-74 STRS_FileOpen

Requirement	The STRS infrastructure shall contain a callable STRS_FileOpen method as described in table 55, STRS_FileOpen().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.

Example	`STRS_HandleID fromWF = APP_GetHandleID();`
	`STRS_HandleID frd =`
	`  STRS_FileOpen(fromWF,filename,`
	`  STRS_ACCESS_READ,`
	`  STRS_TYPE_TEXT);`
	`STRS_Result rtn = STRS_ValidateHandleID(frd);`
	`if ( ! STRS_IsOK(rtn)) {`
	`  STRS_Buffer_Size nb = strlen(`
	`  "FileOpen fails.");`
	`  STRS_Log(fromWF, STRS_ERROR_QUEUE,`
	`  "FileOpen fails.", nb);`
	`}`

7.75 STRS-75 STRS_FileRemove

Requirement	The STRS infrastructure shall contain a callable STRS_FileRemove method as described in table 56, STRS_FileRemove().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	Waveform Remove
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.

Example	`STRS_HandleID fromWF = APP_GetHandleID();`
	`STRS_Result rtn =`
	`  STRS_FileRemove(fromWF,oldName);`
	`if ( ! STRS_IsOK(rtn)) {`
	`  STRS_Buffer_Size nb = strlen(`
	`  "FileRemove fails.");`
	`  STRS_Log(fromWF, STRS_ERROR_QUEUE,`
	`  "FileRemove fails.", nb);`
	`}`

7.76 STRS-76 STRS_FileRename

Requirement	The STRS infrastructure shall contain a callable STRS_FileRename method as described in table 57, STRS_FileRename().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.

Example

```
STRS_HandleID fromWF = APP_GetHandleID();
STRS_Result rtn =
   STRS_FileRename(fromWF,oldName,newName);
if ( ! STRS_IsOK(rtn)) {
   STRS_Buffer_Size nb = strlen(
      "FileRename fails.");
   STRS_Log(fromWF, STRS_ERROR_QUEUE,
      "FileRename fails.", nb);
}
```

7.77 STRS-77 Use Messaging API

Requirement	The STRS applications shall use the STRS Infrastructure Messaging, STRS Infrastructure Data Source, and STRS Infrastructure Data Sink methods to establish queues to send messages between components.
Rationale	In an SDR executing multiple threads or processes, messages have to be processed using message passing methods so that they do not interfere with each other. One example might be a receive application queuing its data for a subsequent transmit application. Another example might be the queuing of error messages from STRS_Log.
Category	Portability, Adaptability
Traced-from	4.5, 4.9, 5.1, 5.2
Use Case	None
Applicable to	Application developer
Notes	The STRS Infrastructure Messaging methods refer to the API defined in STRS-77, STRS-80, STRS-81 and STRS-126 through STRS-129 as well as corresponding tables 58 through 63.
Verification Method	Inspection using STRS compliance tool.

7.78 STRS-78 STRS_QueueCreate (Deleted)

Rationale	This requirement was replaced by two: STRS-128, STRS_PubSubCreate and STRS-126, STRS_MessageQueueCreate, with similar functionality but having different calling sequences to avoid confusion.

7.79 STRS-79 STRS_QueueDelete (Deleted)

Rationale	This requirement was replaced by two: STRS-129, STRS_PubSubDelete and STRS-119, STRS_MessageQueueDelete, with similar functionality but having different calling sequences to avoid confusion.

7.80 STRS-80 STRS_Register

Requirement	The STRS infrastructure shall contain a callable STRS_Register method as described in table 62, STRS_Register().
Rationale	The publish-subscribe design pattern provided a way for the publisher of a message to send the message to all subscribers without knowing the details. For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.

Example	`STRS_HandleID fromWF = APP_GetHandleID();`
	`STRS_Result rtn = STRS_Register(fromWF,psX,qFC);`
	`if (! STRS_IsOK(rtn)) {`
	`  STRS_Buffer_Size nb = strlen(`
	`  "Can't register subscriber.");`
	`  STRS_Log(fromWF,STRS_ERROR_QUEUE,`
	`  "Can’t register subscriber.", nb);`
	`}`

7.81 STRS-81 STRS_Unregister

Requirement	The STRS infrastructure shall contain a callable STRS_Unregister method as described in table 63, STRS_Unregister().
Rationale	The publish-subscribe design pattern provides a way for the publisher of a message to send the message to all subscribers without knowing the details. For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.

Example

```
STRS_HandleID fromWF = APP_GetHandleID();
// After unregistering, upon STRS_Write to psX,
// the message does not get sent to qFC.
STRS_Result rtn = STRS_Unregister(fromWF,psX,qFC);
if (! STRS_IsOK(rtn)) {
   STRS_Buffer_Size nb = strlen(
      "Can't unregister subscriber.");
   STRS_Log(fromWF,STRS_ERROR_QUEUE,
      "Can’t unregister subscriber.", nb);
}
```

7.82 STRS-82 Use Time Control API

Requirement	Any portion of the STRS Applications on the GPP needing time control shall use the STRS Infrastructure Time Control methods to access the hardware and software timers.
Rationale	For portability of waveform applications, a standard API for using timers in the GPP was necessary. The timers are expected to be used for relatively low accuracy timing such as time stamps, timed events, and time constraints. As the speed of new GPPs increases over time, the timers are expected to be used for signal processing.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	Application developer
Notes	The STRS Infrastructure Time Control methods refer to the API defined in STRS-82 through STRS-88, STRS-130 through STRS-133 and corresponding tables 64 through 73 in NASA-STD-4009A.
Verification Method	Inspection

7.83 STRS-83 STRS_GetNanoseconds

Requirement	The STRS infrastructure shall contain a callable STRS_GetNanoseconds method as described in table 65, STRS_GetNanoseconds().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.

Verification Method	Inspection using STRS compliance tool.
See Also	STRS_GetTime, STRS_GetTimeWarp, STRS_GetSeconds

Example	`STRS_TimeWarp base, timx;`
	`STRS_Nanoseconds nsec;`
	`STRS_Result rtn;`
	`STRS_HandleID fromWF = APP_GetHandleID();`
	`STRS_Clock_Kind kx = 1;`
	`rtn =`
	`  STRS_GetTime(fromWF,toDev,*base,kx,*timx);`
	`nsec = STRS_GetNanoseconds(base);`

7.84 STRS-84 STRS_GetSeconds

Requirement	The STRS infrastructure shall contain a callable STRS_GetSeconds method as described in table 66, STRS_GetSeconds().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	STRS_GetTime, STRS_GetTimeWarp, STRS_GetNanoseconds

Example	`STRS_TimeWarp base,timx;`
	`STRS_Seconds isec;`
	`STRS_Result rtn;`
	`STRS_HandleID fromWF = APP_GetHandleID();`
	`STRS_Clock_Kind kx = 1;`
	`rtn = STRS_GetTime(fromWF,toDev,*base,kx,*timx);`
	`isec = STRS_GetSeconds(base);`

7.85 STRS-85 STRS_GetTime

Requirement	The STRS infrastructure shall contain a callable STRS_GetTime method as described in table 67, STRS_GetTime().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2

Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	STRS_SetTime
Example	

```
STRS_Result rtn;
STRS_HandleID fromWF = APP_GetHandleID();
STRS_HandleID refDev = STRS_HandleRequest(fromWF,
  ”Name of Reference clock”);
STRS_HandleID tgtDev = STRS_HandleRequest(fromWF,
  ”Name of drifting clock”);
STRS_HandleID defDev = STRS_HandleRequest(fromWF,
     STRS_DEFAULT_CLOCK_NAME);
STRS_Clock_Kind kRef = 1;
STRS_Clock_Kind kTgt = 1;
STRS_TimeWarp refbase, refkind, tgtbase, tgtkind;

rtn = STRS_GetTime(fromWF,refDev, refbase, kRef, refkind);
rtn = STRS_GetTime(fromWF,tgtDev, tgtbase, kTgt, tgtkind);
STRS_TimeWarp initial_difference = refkind – tgtkind;

while(TRUE)
{
    STRS_Sleep(fromWF, defDev, STRS_DEFAULT_CLOCK_KIND,
                POLL_INTERVAL, false);
        rtn = STRS_GetTime(fromWF, refDev, refbase, kRef,
refkind);
    rtn = STRS_GetTime(fromWF, tgtDev, tgtbase, kTgt, tgtkind);
    STRS__TimeWarp drift = (refkind – tgtkind) –
      initial_difference;
    STRS_TimeAdjust tRate = drift * FEEDBACK_COEFFICIENT;

    rtn = STRS_SetTimeAdjust(fromWF, tgtDev, tRate);
}
```

7.86 STRS-86 STRS_GetTimeWarp

Requirement	The STRS infrastructure shall contain a callable STRS_GetTimeWarp method as described in table 69, STRS_GetTimeWarp().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.

Verification Method	Inspection using STRS compliance tool.
See Also	STRS_GetNanoseconds, STRS_GetSeconds, STRS_SetTime

Example	`STRS_TimeWarp delta;`
	`STRS_Seconds isec = 1; /* Leap second. */`
	`STRS_Nanoseconds nsec = 0;`
	`delta = STRS_GetTimeWarp(isec,nsec);`

7.87 STRS-87 STRS_SetTime

Requirement	The STRS infrastructure shall contain a callable STRS_SetTime method as described in table 70, STRS_SetTime().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	STRS_GetTime, STRS_GetTimeWarp

Example	`STRS_TimeWarp delta;`
	`STRS_Seconds isec = 1; /* Leap second */`
	`STRS_Nanoseconds nsec = 0;`
	`STRS_Result rtn;`
	`STRS_HandleID fromWF = APP_GetHandleID();`
	`STRS_Clock_Kind k1 = 1;`
	`delta = STRS_GetTimeWarp(isec,nsec);`
	`rtn = STRS_SetTime(fromWF,toDev,k1,delta);`

7.88 STRS-88 STRS_TimeSynch

Requirement	The STRS infrastructure shall contain a callable STRS_TimeSynch method as described in table 73, STRS_TimeSynch().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider

Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.

Example	`STRS_HandleID fromWF = APP_GetHandleID();`
	`qref = STRS_HandleRequest(fromWF,”ReferenceClock”);`
	`iref = 0;`
	`qtgt = STRS_HandleRequest(fromWF,”TargetClock”);`
	`itgt = 0;`
	`// To use previous technique, step max is`
	`// defined as jump = 0.`
	`STRS_TimeWarp jump = STRS_GetTimeWarp(0,0);`
	`rtn = STRS_TimeSynch(fromWF,qref,iref,qtgt,itgt,`
	`jump);`
	`if ( ! STRS_IsOK(rtn)) {`
	`STRS_Buffer_Size nb = strlen(`
	`“STRS_ Synch fails.”);`
	`STRS_Log(fromWF, STRS_ERROR_QUEUE,`
	`“STRS_ Synch fails.”, nb);`
	`}`

7.89 STRS-89 Provide STRS.h

Requirement	The STRS platform provider shall provide an STRS.h file containing the STRS predefined data shown in table 74, STRS Predefined Data.
Rationale	For portability, standard names are defined for various constants and data types.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using OE script and using STRS compliance tool.

7.90 STRS-90 Provide POSIX®

Requirement	The STRS OE shall provide the interfaces described in POSIX® IEEE Standard 1003.13 profile PSE51.
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. Rather than creating many more APIs in STRS, the currently available POSIX® standard was selected. Thus POSIX® subsets were chosen to implement certain OS services missing from the list of STRS Application-provided methods. The POSIX® subsets are widely available, are implemented by multiple OSs, and are scalable. Layering of the architecture separates and encapsulates

	functionality so that the parts are less influenced by changes to the other. This separation of functionality promotes portability.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.7, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	None
Verification Method	Inspection

7.91 STRS-91 Use POSIX®

Requirement	STRS applications shall use POSIX® methods except for the unsafe functions listed in table 77, Replacements for Unsafe Functions.
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. Rather than creating many more APIs in STRS, the currently available POSIX® standard was selected. Thus POSIX® subsets were chosen to implement certain OS services missing from the list of STRS Application-provided methods. The POSIX® subsets are widely available, are implemented by multiple OSs, and are scalable. Layering of the architecture separates and encapsulates functionality so that the parts are less influenced by changes to the other. This separation of functionality promotes portability.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.7, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	Application developer
Notes	Table 2, STRS Architecture Standard, Table 77, Replacements for Unsafe Functions, is a copy of table 77 of NASA-STD-4009A cited in the requirement.
Verification Method	Inspection using STRS compliance tool.

Table 2—STRS Architecture Standard, Table 77, Replacements for Unsafe Functions

Unsafe Function Do Not Use!	Reentrant Counterpart OK to Use
abort	STRS_AbortApp
asctime	asctime_r
atexit	-
ctermid	ctermid_r
ctime	ctime_r
exit	STRS_AbortApp
getlogin	getlogin_r
gmtime	gmtime_r
localtime	localtime_r
rand	rand_r
readdir	readdir_r
strtok	strtok_r
tmpnam	tmpnam_r

7.92 STRS-92 Document HAL

Requirement	The STRS platform provider shall provide STRS platform HAL documentation that includes the following: (1) For each method or function, its calling sequence, return values, an explanation of its functionality, any preconditions for using the method or function, and the postconditions after using the method or function. (2) Information required to address the underlying hardware, including interrupt input and output, memory mapping, and the configuration data necessary to operate in the STRS platform environment.
Rationale	The HAL API is to be published so that specialized hardware made by one company may be integrated with the STRS infrastructure made by a different company. The HAL API documentation is to include a description of each method or function used, including its calling sequence, return values, an explanation of its functionality, any preconditions before using the method/function, and the status after using the method or function. The HAL API documentation is to also contain information about the underlying hardware such as address and data interfaces, interrupt input and output, power connections, plus other control and data lines necessary to operate in the STRS platform environment.
Category	Reconfigurability, Adaptability, Extensibility
Traced-from	4.4, 4.5, 4.7, 4.8, 5.1, 5.2
Use Case	None

Applicable to	Platform
Notes	See STRS-11
Verification Method	Inspection of HAL document.

Example

HAL API	RESULT OPEN(HANDLE* resourceHandle, RESOURCE_NAME resourceName)
Description	Open a resource by name. If no errors are encountered, use the resourceHandle to access the resource.
Parameters	• resourceHandle - [out] A pointer to place the opened handle into • resourceName - [in] The name of the resource to open
Return	A 32-bit signed integer used to determine whether an error has occurred. Use TEST_ERROR to obtain a printable message. • Zero - No errors or warnings. • Positive – Warning. • Negative – Error.
Precondition	Resource is not open before executing this command.
Postcondition	Resource will be open and ready for further access if no error was encountered.
See Also	READ, WRITE, CLOSE, TEST_ERROR

Example	#include <HALResources.h>
	…
	RESULT result;
	HANDLE resourceHandle;
	RESOURCE_NAME resourceName = "FPGA";
	result = OPEN(&resourceHandle, resourceName)
	if (result < 0) {
	cout << "Error: " << TEST_ERROR(result) << endl;
	} else if (result > 0) {
	cout << "Warning: " << TEST_ERROR(result) << endl;
	}

7.93 STRS-93 OE Uses HAL (Deleted)

Rationale	This requirement was deleted because it was the same as STRS-11. STRS-11 stated the same thing, with only a few words different, so STRS-93 was redundant. STRS-93 previously stated: The STRS infrastructure shall use the HAL APIs to communicate with the specialized hardware via the physical interface defined by the STRS platform provider.

7.94 STRS-94 External Commands

Requirement	An STRS platform shall accept, validate, and respond to external commands.
Rationale	To adapt to changing circumstances, an STRS radio has to accept external commands from a ground station, another satellite, or another system on the same satellite. The external commands have to be validated as required by the

	mission. There has to be a way to determine whether or not the command worked and, for some commands, the resulting values.
Category	Adaptability
Traced-from	4.5, 5.4, 5.15
Use Case	Waveform Upload, STRS OE Upload, Waveform Instantiation, Waveform Start, Processor Resource Sharing with Flight Computer, Set Waveform Parameter, Get Waveform Parameter, Transmit a Packet, Receive a Packet, Waveform Stop, Waveform Deallocation, Waveform Abort, Waveform Remove, Built-In-Test
Applicable to	OE developer: usually platform provider
Notes	None
Verification Method	Observation of radio operation.

7.95 STRS-95 Use STRS APIs

Requirement	An STRS platform shall execute external application control commands using the standardized STRS APIs.
Rationale	To promote portability and adaptability, the use of the standard STRS APIs is required. One waveform should be able to control another waveform or device in a portable manner.
Category	Portability, Adaptability
Traced-from	4.1, 4.2, 4.3, 4.5, 4.9, 5.1, 5.2, 5.4, 5.6, 5.7
Use Case	Waveform Upload, STRS OE Upload, Waveform Instantiation, Waveform Start, Processor Resource Sharing with Flight Computer, Set Waveform Parameter, Get Waveform Parameter, Transmit a Packet, Receive a Packet, Waveform Stop, Waveform Deallocation, Waveform Abort, Waveform Remove, Built-In Test
Applicable to	OE developer: usually platform provider
Notes	None
Verification Method	Inspection using STRS compliance tool.

7.96 STRS-96 Use STRS_Query

Requirement	The STRS infrastructure shall use the STRS_Query method to service external system requests for information from an STRS application.
Rationale	The only way to request information from an application is by means of data values returned when an application method is invoked. The STRS_Query/APP_Query methods are designed for this purpose. Although STRS_RunTest/APP_RunTest could be used to request data values, they are designed for testing.
Category	Adaptability, Extensibility
Traced-from	4.4, 4.5, 5.1, 5.2

Use Case	Get Waveform Parameter
Applicable to	OE developer: usually platform provider
Notes	None
Verification Method	Inspection

7.97 STRS-97 Use STRS_Log (Deleted)

Rationale	This requirement was deleted because it was the same as STRS-57. STRS-97 previously stated: An STRS application shall use the STRS_Log and STRS_Write methods to send STRS telemetry set information to the external system. STRS-97 was less clear than STRS-57 because STRS-97 involved the external interface, not just the application to infrastructure call. Also, in some implementations, the application telemetry may not be sent directly to the external interface but may be sent to a file from which the telemetry may be downloaded as necessary.

7.98 STRS-98 Document Platform for XML (Project Option)

Requirement	(Optional, OE-specific) The STRS platform provider shall document the necessary platform information (including a sample file) to develop a predeployed application configuration file in XML 1.0.
Rationale	When functions, interfaces, components, and/or design rules are defined and published, the architecture is open. Open architectures facilitate interoperability among commercial and government developers and minimize the operational impact of upgrading hardware and software components. Leveraging the existing XML standard may reduce NASA's costs and risks by increasing reliability.
Category	Reconfigurability, Adaptability, Extensibility
Traced-from	4.2, 4.4, 4.5, 4.7, 5.2, 5.4
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	None
Verification Method	Inspection of document and sample file.

7.99 STRS-99 Document WF for XML (Deleted)

Rationale	This requirement was deleted because it is redundant. This requirement is replaced by STRS-12 (3). STRS-99 previously stated: The STRS application developer shall document the necessary application information to develop a predeployed application configuration file in XML 1.0.

7.100 STRS-100 Provide XML File (Project Option)

Requirement	(Optional, OE-specific) The STRS integrator shall provide a predeployed application configuration file in XML 1.0.
Rationale	A waveform (STRS application) configuration file contains specific information that (1) allows STRS to instantiate the application; (2) provides default configuration values; and (3) provides connection references to ports and services needed by the application. The format of the configuration files has to be defined in XML using an XML schema. The XML should be preprocessed to optimize space in the STRS radio memory while keeping the equivalent content. Examples include platform configuration files, STRS infrastructure configuration files as a XML schema, and waveform configuration files that contain specific information that allows STRS to instantiate the application, provide default configuration values, and provide connection references to ports and services needed by the application. Leveraging the existing XML standard may reduce NASA's costs and risks by increasing reliability.
Category	Reconfigurability
Traced-from	4.7, 5.3, 5.4
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	None
Verification Method	Inspection using XMLSpy.

7.101 STRS-101 XML Content (Project Option)

Requirement	(Optional, OE-specific) The predeployed STRS application configuration file shall identify the following application attributes and default values: (1) Identification. A. Class name. (2) State after processing the configuration file or equivalent. (3) Any resources to be loaded separately. A. Filename of loadable image. B. Target on which to put loadable image file. C. Target memory in bytes, number of gates, or logic elements. (4) Initial or default values for all distinct operationally configurable parameters.
Rationale	A waveform (STRS application) configuration file contains specific information that (1) allows STRS to instantiate the application; (2) provides default configuration values; and (3) provides connection references to ports and services needed by the application. The format of the configuration files has to be defined in XML using an XML schema. The XML should be

	preprocessed to optimize space in the STRS radio memory while keeping the equivalent content. Examples include platform configuration files, STRS infrastructure configuration files as a XML schema, and waveform configuration files that contain specific information that allows STRS to instantiate the application, provide default configuration values, and provide connection references to ports and services needed by the application.
Category	Reconfigurability
Traced-from	5.4
Use Case	None
Applicable to	Application developer
Notes	None
Verification Method	Inspection of delivered files and documentation.

7.102 STRS-102 Provide XML Schema (Project Option)

Requirement	(Optional, OE-specific) The STRS platform provider shall provide an XML 1.0 schema definition (XSD) file to validate the format and data for predeployed STRS application configuration files, including the order of the tags, the number of occurrences of each tag, and the values or attributes.
Rationale	A waveform (STRS application) configuration file contains specific information that (1) allows STRS to instantiate the application; (2) provides default configuration values; and (3) provides connection references to ports and services needed by the application. The format of the configuration files has to be defined in XML using an XML schema. Since the term XML schema was variously interpreted to mean either a description or a file, the requirement was clarified to specify that an XML schema definition (XSD) file is required. The XML should be preprocessed to optimize space in the STRS radio memory while keeping the equivalent content. Examples include platform configuration files, STRS infrastructure configuration files as a XML schema, and waveform configuration files that contain specific information that allows STRS to instantiate the application, provide default configuration values, and provide connection references to ports and services needed by the application. Leveraging the existing XML standard may reduce NASA's costs and risks by increasing reliability.
Category	Reliability
Traced-from	4.3, 4.7, 5.3, 5.4
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	None
Verification Method	Inspection using XMLSpy.

7.103 STRS-103 Provide XML Transformation Tool (Project Option)

Requirement	(Optional, OE-specific) The STRS platform provider shall document the transformation (if any) from a predeployed application configuration file in XML into a deployed application configuration file and provide the tools to perform such transformation.
Rationale	A waveform (STRS application) configuration file contains specific information that (1) allows STRS to instantiate the application; (2) provides default configuration values; and (3) provides connection references to ports and services needed by the application. The format of the configuration files has to be defined in XML using an XML schema. The XML should be preprocessed to optimize space in the STRS radio memory while keeping the equivalent content. Examples include platform configuration files, STRS infrastructure configuration files as an XML schema, and waveform configuration files that contain specific information that allows STRS to instantiate the application, provide default configuration values, and provide connection references to ports and services needed by the application.
Category	Reconfigurability, Adaptability, Extensibility
Traced-from	4.4, 4.5, 5.2, 5.3, 5.4
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	None
Verification Method	Inspection of document and tools.

7.104 STRS-104 Provide XML Transformed (Project Option)

Requirement	(Optional, OE-specific) The STRS integrator shall provide a deployed STRS application configuration file for the STRS infrastructure to place the STRS application in the specified state.
Rationale	A waveform (STRS application) configuration file contains specific information that (1) allows STRS to instantiate the application; (2) provides default configuration values; and (3) provides connection references to ports and services needed by the application. The format of the configuration files has to be defined in XML using an XML schema. The XML should be preprocessed to optimize space in the STRS radio memory while keeping the equivalent content. Examples include platform configuration files, STRS infrastructure configuration files as an XML schema, and waveform configuration files that contain specific information that allows STRS to instantiate the application, provide default configuration values, and provide connection references to ports and services needed by the application.
Category	Reconfigurability
Traced-from	5.4
Use Case	None
Applicable to	OE developer: usually platform provider

Notes	None
Verification Method	Inspection of delivered files and documentation.

7.105 STRS-105 OE Provides API in C

Requirement	The STRS infrastructure APIs shall have an ISO/IEC C language compatible interface.
Rationale	Because portability is a basic goal but middleware is not required, a totally language-independent solution was not available. The lowest common denominator turns out to be a C language interface. Using a standard ISO/IEC 9899 C or ISO/IEC 14882 C++ aids portability. The year is not included in the requirement, so that obsolete compilers are not mandated.
Category	Portability
Traced-from	4.1, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	None
Verification Method	Inspection using STRS compliance tool.

7.106 STRS-106 Use STRS.h

Requirement	An STRS application shall use the appropriate constant, typedef, or struct defined in table 74, STRS Predefined Data, when the data are used to interact with the STRS APIs.
Rationale	For portability, standard names are defined for various constants and data types.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	Application developer
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection

7.107 STRS-107 Document External Commands

Requirement	An STRS platform provider shall document the external commands describing their format, function, and any STRS methods invoked.
Rationale	To adapt to changing circumstances, an STRS radio has to accept external commands from a ground station, another satellite, or another system on the same satellite. The external commands have to be validated as required by the mission. There has to be a way to determine whether the command worked

	and, for some commands, the resulting values. To promote portability and adaptability, the use of the standard STRS APIs is required.
Category	Adaptability
Traced-from	4.5, 5.1, 5.4, 5.5, 5.6, 5.7, 5.15
Use Case	Waveform Upload, STRS OE Upload, Waveform Instantiation, Waveform Start, Processor Resource Sharing with Flight Computer, Set Waveform Parameter, Get Waveform Parameter, Transmit a Packet, Receive a Packet, Waveform Stop, Waveform Deallocation, Waveform Abort, Waveform Remove, Built-In-Test
Applicable to	OE developer: usually platform provider
Notes	None
Verification Method	Inspection of documentation.

7.108 STRS-108 Document Thermal and Power Limits

Requirement	The STRS 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.
Rationale	The power consumption and resulting heat generation of a reprogrammable FPGA will vary according to the amount of logic used and the clock frequency(s). The power consumption may not be constant for each possible waveform that can be loaded on the platform. The STRS 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.
Category	Reliability, Adaptability
Traced-from	4.3, 4.7, 5.9, 5.10, 5.15, 5.16
Use Case	None
Applicable to	Platform provider
Notes	See also STRS-9
Verification Method	Inspection of HID document.

7.109 STRS-109 Provide General-Purpose Processing Module

Requirement	An STRS platform shall have a GPM that contains and executes the STRS OE and the control portions of the STRS applications and services software.
Rationale	The GPM contains and executes the STRS OE, including POSIX®, STRS interface code, and configuration file parsing, to support the corresponding requirements. A layered hardware architecture augments the layered software

	architecture by providing the ability to change portions without affecting other portions to support extensibility, adaptability, and portability.
Category	Portability, Adaptability
Traced-from	4.3, 4.4, 4.5, 4.6, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	Platform provider
Notes	None
Verification Method	Inspection

7.110 STRS-110 Provide STRS_APIs.h

Requirement	The STRS platform provider shall provide an “STRS_APIs.h” that contains the method prototypes for the infrastructure APIs.
Rationale	For portability, standard names are defined for various constants, data types, and method prototypes in the API.
Category	Portability
Traced-from	4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	None
Verification Method	Inspection

7.111 STRS-111 Include STRS_DeviceControl.h

Requirement	Each STRS Device shall contain: #include "STRS_DeviceControl.h".
Rationale	For portability, standard names are defined for various constants, data types, and method prototypes in the API.
Category	Portability
Traced-from	4.9, 5.1, 5.2
Use Case	None
Applicable to	Application developer
Notes	None
Verification Method	Inspection using STRS compliance tool.

7.112 STRS-112 Provide STRS_DeviceControl.h

Requirement	The STRS platform provider shall provide an “STRS_DeviceControl.h” that contains the method prototypes for each STRS Device and, for C++, the class

	definition for the base class STRS_DeviceControl, which inherits from the base class STRS_ApplicationControl.
Rationale	For portability, standard names are defined for various constants, data types, and method prototypes in the API.
Category	Portability
Traced-from	4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	None
Verification Method	Inspection
Example	None

7.113 STRS-113 STRS_DeviceControl Base Class

Requirement	If the STRS Device-provided Device Control API is implemented in C++, the STRS Device class shall be derived from the STRS_DeviceControl base class.
Rationale	For portability, standard names are defined for various constants, data types, and method prototypes in the API.
Category	Portability
Traced-from	4.9, 5.1, 5.2
Use Case	None
Applicable to	Device
Notes	None
Verification Method	Inspection

Example	For example, the MyDevice.h file should contain a class definition of the form:
	`class MyDevice: public STRS_DeviceControl`
	`[, public STRS_Source]`
	`[, public STRS_Sink]`
	`{…};`
	`Note: [] indicates optional.`

7.114 STRS-114 APP_Destroy

Requirement	Each STRS application shall contain a callable APP_Destroy method as described in table 6, APP_Destroy().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability

Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	Application developer
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	APP_Instance

Example (C)

```
/* Not a class method */
/* Called in OE as:   */
ClassName_APP_Destroy(thisPtr);

/* Implemented in application as: */
void ClassName_APP_Destroy(
         STRS_Instance *thisPtr)
{
   free thisPtr;  /* Non-standard */
}
```

Example (C++)

```
// Not a class method
void ClassName::APP_Destroy (
          STRS_Instance *Obj)
{
    delete static_cast<ClassName *>(Obj);
}
```

7.115 STRS-115 APP_GetHandleID

Requirement	The STRS infrastructure shall define a callable APP_GetHandleID method in each application as described in table 7, APP_GetHandleID().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.

Example

```
if ( ! STRS_IsOK(rtn)) {
    int nb = sprint(buffer,
            “I found the error in %s at %s:%d\0”,
            APP_GetHandleName(), __FILE__,
            __LINE__ );
    STRS_Log(APP_GetHandleID(),
            STRS_GetErrorQueue(rtn), buffer, nb);
    return rtn;
}
```

7.116 STRS-116 APP_GetHandleName

Requirement	The STRS infrastructure shall define a callable APP_GetHandleName method in each application as described in table 8, APP_GetHandleName().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.

Example

```
if ( ! STRS_IsOK(rtn)) _
    int nb = sprint(buffer,
            “I found the error in %s at %s:%d\0”,
            APP_GetHandleName(), __FILE__,
            __LINE__ );
    STRS_Log(APP_GetHandleID(),
            STRS_GetErrorQueue(rtn), buffer, nb);
    return rtn;
}
```

7.117 STRS-117 STRS_GetHandleName

Requirement	The STRS infrastructure shall contain a callable STRS_GetHandleName method as described in table 29, STRS_GetHandleName().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider

Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	STRS_HandleRequest
Example	

```
char toResourceName[STRS_MAX_HANDLE_NAME_SIZE+1];
STRS_HandleID fromWF = APP_GetHandleID();
STRS_Result rtn = STRS_GetHandleName(fromWF,
                                     toID,
                                     toResourceName);
If ( ! STRS_IsOK(rtn)) {
   STRS_HandleID errQ =
           STRS_GetErrorQueue(rtn);
   STRS_Buffer_Size nb = strlen(
           "Did not find handle name.");
   STRS_Log(fromWF,errQ,
           "Did not find handle name.", nb);
} else {
     std::cout << "Found Handle name "
               << toResourceName << " for ID "
               << toID << std::endl;
}
```

7.118 STRS-118 STRS_ValidateHandleID

Requirement	The STRS infrastructure shall contain a callable STRS_ValidateHandleID method as described in table 34, STRS_ValidateHandleID ().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	STRS_HandleRequest, STRS_FileOpen, STRS_InstantiateApp, STRS_MessageQueueCreate, STRS_PubSubCreate, STRS_IsOK

Example

```
STRS_HandleID tstID =
      STRS_HandleRequest(fromID, otherWF);
STRS_Result  rtn = STRS_ValidateHandleID(tstID);
If ( ! STRS_IsOK( rtn )) {
   STRS_Message msg = “Handle ID error.”;
   STRS_HandleID errQ = STRS_GetErrorQueue(rtn);
   STRS_Log (fromID, errQ, msg, (STRS_Buffer_Size)
            sizeof(msg));
} else {
   cout << “OK.” << endl;
}
```

7.119 STRS-119 STRS_ValidateSize

Requirement	The STRS infrastructure shall contain a callable STRS_ValidateSize method as described in table 35, STRS_ValidateSize ().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	STRS_FileGetFreeSpace, STRS_FileGetSize, STRS_IsOK

Example

```
STRS_File_Size tstSize =
       STRS_FileGetFreeSpace(fromID, NULL);
STRS_Result  rtn = STRS_ValidateSize(tstSize);
If ( ! STRS_IsOK( rtn )) {
   STRS_Message msg = “File size error.”;
   STRS_HandleID errQ = STRS_GetErrorQueue(rtn);
   STRS_Log (fromID, errQ, msg, (STRS_Buffer_Size)
       sizeof(msg));
} else {
   cout << “OK.” << endl;
}
```

7.120 STRS-120 DEV_Close

Requirement	If the hardware is to be loaded by the STRS Device, the STRS Device shall contain a callable DEV_Close method as described in table 45, DEV_Close().

Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	Device
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using compliance tool.
See Also	STRS_DeviceClose

Example	`STRS_Result DEV_Close() {`
	`  STRS_Result rtn = STRS_OK;`
	`  int ok = HAL_Close(myDev);`
	`  if ( ! HAL_Successful(ok)) {`
	`  STRS_HandleID fromWF = APP_GetHandleID();`
	`  STRS_Buffer_Size nb = strlen(`
	`  "DEV_Close fails.");`
	`  STRS_Log(fromWF, STRS_ERROR_QUEUE,`
	`  "DEV_Close fails.", nb);`
	`  rtn = STRS_ERROR;`
	`  }`
	`  return rtn;`
	`}`

7.121 STRS-121 DEV_Flush

Requirement	If the hardware is to be flushed by the STRS Device, the STRS Device shall contain a callable DEV_Flush method as described in table 46, DEV_Flush().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	Device
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	STRS_DeviceFlush

Example

```
STRS_Result DEV_Flush() {
   STRS_Result rtn = STRS_OK;
   int ok = HAL_Flush(myDev);
   if ( ! Hal_Successful(ok)){
      STRS_HandleID fromWF = APP_GetHandleID();
      STRS_Buffer_Size nb = strlen(
           "DEV_Flush fails.");
      STRS_Log(fromWF, STRS_ERROR_QUEUE,
           "DEV_Flush fails.", nb);
     rtn = STRS_ERROR;
   }
   return rtn;
}
```

7.122 STRS-122 DEV_Load

Requirement	If the hardware is to be loaded by the STRS Device, the STRS Device shall contain a callable DEV_Load method as described in table 47, DEV_Load().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	Device
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using compliance tool.
See Also	STRS_DeviceLoad

Example

```
STRS_Result DEV_Load(char *fileName ) {
   STRS_Result rtn = STRS_OK;
   int ok = HAL_Load(myDev,fileName);
   if ( ! Hal_Successful(ok)){
      STRS_HandleID fromWF = APP_GetHandleID();
      STRS_Buffer_Size nb = strlen(
           "DEV_Load fails.");
      STRS_Log(fromWF, STRS_ERROR_QUEUE,
           "DEV_Load fails.", nb);
      rtn = STRS_ERROR;
   }
   return rtn;
}
```

7.123 STRS-123 DEV_Open

Requirement	If the hardware is to be loaded by the STRS Device, the STRS Device shall contain a callable DEV_Open method as described in table 48, DEV_Open().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	Device
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	STRS_DeviceOpen

Example	`STRS_Result DEV_Open() {`
	`  STRS_Result rtn = STRS_OK;`
	`  myDev = HAL_Open("/dev/fpga1", O_RDWR);`
	`  if (myDev != NULL) {`
	`  STRS_HandleID fromWF = APP_GetHandleID();`
	`  STRS_Buffer_Size nb = strlen(`
	`  "DEV_Open fails.");`
	`  STRS_Log(fromWF, STRS_ERROR_QUEUE,`
	`  "DEV_Open fails.", nb);`
	`  rtn = STRS_ERROR;`
	`  }`
	`  return rtn;`
	`}`

7.124 STRS-124 DEV_Reset

Requirement	If the hardware is to be reset by the STRS Device, the STRS Device shall contain a callable DEV_Reset method as described in table 49, DEV_Reset().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	Device
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	STRS_DeviceReset

Example

```
STRS_Result DEV_Reset() {
   STRS_Result rtn = STRS_OK;
   int ok = HAL_Reset(myDev);
   if ( ! Hal_Successful(ok)){
      STRS_HandleID fromWF =APP_GetHandleID();
      STRS_Buffer_Size nb = strlen(
           "DEV_Reset fails.");
      STRS_Log(fromWF, STRS_ERROR_QUEUE,
           "DEV_Reset fails.", nb);
     rtn = STRS_ERROR;
   }
   return rtn;
}
```

7.125 STRS-125 DEV_Unload

Requirement	If the hardware is to be loaded by the STRS Device, the STRS Device shall contain a callable DEV_Unload method as described in table 50, DEV_Unload().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	Waveform Deallocation
Applicable to	Device
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	STRS_DeviceUnload

Example

```
STRS_Result DEV_Unload() {
   STRS_Result rtn = STRS_OK;
   int ok = HAL_Unload(myDev);
   if ( ! Hal_Successful(ok)){
      STRS_HandleID fromWF =APP_GetHandleID();
      STRS_Buffer_Size nb = strlen(
          "DEV_Unload fails.");
      STRS_Log(fromWF, STRS_ERROR_QUEUE,
          "DEV_Unload fails.", nb);
      rtn = STRS_ERROR;
   }
   return rtn;
}
```

7.126 STRS-126 STRS_MessageQueueCreate

Requirement	The STRS infrastructure shall contain a callable STRS_MessageQueueCreate method as described in table 58, STRS_ MessageQueueCreate().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
Example	

```
STRS_HandleID fromWF = APP_GetHandleID();
STRS_Buffer_Size nbmax =
   STRS_MAX_LOG_MESSAGE_SIZE+1;
STRS_Queue_Max_Messages nmsgmax = 20;
if (nmsgmax > STRS_MAX_QUEUE_MESSAGES)
    nmsgmax = STRS_MAX_QUEUE_MESSAGES;
STRS_HandleID qX = STRS_MessageQueueCreate(fromWF,
 "QX", nbmax, nmsgmax);
STRS_Result rtn = STRS_ValidateHandleID(qX);
if ( ! STRS_IsOK(rtn)) {
    STRS_Buffer_Size nb = strlen(
           "Can't create queue.");
    STRS_Log(fromWF,STRS_ERROR_QUEUE,
           "Can't create queue.", nb).
    return STRS_ERROR;

}
```

7.127 STRS-127 STRS_MessageQueueDelete

Requirement	The STRS infrastructure shall contain a callable STRS_MessageQueueDelete method as described in table 59, STRS_ MessageQueueDelete().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.

Verification Method	Inspection using STRS compliance tool.

Example	`STRS_HandleID fromWF = APP_GetHandleID();`
	`STRS_Result rtn =`
	`  STRS_MessageQueueDelete(fromWF,qX);`
	`if (! STRS_IsOK(rtn)) {`
	`  STRS_Buffer_Size nb = strlen(`
	`  "Can't delete queue.");`
	`  STRS_Log(fromWF,STRS_ERROR_QUEUE,`
	`  "Can’t delete queue.", nb);`
	`  return STRS_ERROR;`
	`}`

7.128 STRS-128 STRS_PubSubCreate

Requirement	The STRS infrastructure shall contain a callable STRS_PubSubCreate method as described in table 60, STRS_PubSubCreate().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	STRS_Register

Example	`STRS_HandleID psX;`
	`STRS_HandleID fromWF = APP_GetHandleID();`
	`psX = STRS_PubSubCreate(fromWF, "PSX");`
	`STRS_Result rtn = STRS_ValidateHandleID(psX);`
	`if ( ! STRS_IsOK(rtn)) {`
	`  STRS_Buffer_Size nb = strlen(`
	`  "Can’t create Pub/Sub.");`
	`  STRS_Log(fromWF,STRS_ERROR_QUEUE,`
	`  "Can't create Pub/Sub.", nb).`
	`  return STRS_ERROR;`
	`}`

7.129 STRS-129 STRS_PubSubDelete

Requirement	The STRS infrastructure shall contain a callable STRS_PubSubDelete method as described in table 61, STRS_PubSubDelete().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.

Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.

Example	`STRS_HandleID fromWF = APP_GetHandleID();`
	`STRS_Result rtn = STRS_PubSubDelete(fromWF,psX);`
	`if (! STRS_IsOK(rtn)) {`
	`  STRS_Buffer_Size nb = strlen(`
	`  "Can't delete Pub/Sub.");`
	`  STRS_Log(fromWF,STRS_ERROR_QUEUE,`
	`  "Can’t delete Pub/Sub.", nb);`
	`}`

7.130 STRS-130 Document STRS Clock/Timer

Requirement	The implementer of an STRS clock/timer software component for use with STRS_GetTime shall document it to include handle name, kind, epoch, resolution, use of leap seconds, and whether it should match a time somewhere else, as further described in table 64, Document STRS Clock/Timer.
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	STRS clock/timer developer, which may be platform provider or application developer.
Notes	None
Verification Method	Inspection

Example — Sample documentation:

Clock/timer Name	Monotonic from Power On
Clock/timer Handle name	sysTime
Clock/timer Base	Same as kind = 0
Purpose	Be synch-able to flight computer.
Epoch	Power On
Resolution	100 microseconds
Leap Seconds	None
Matches	None
Clock/timer kind	1
Purpose	Keep up with other computers in the system. Timestamp.
Epoch	1/1/1970 GMT
Resolution	Same
Leap Seconds	None
Matches	System time

7.131 STRS-131 STRS_GetTimeAdjust

Requirement — The STRS infrastructure shall contain a callable STRS_GetTimeAdjust method as described in table 68, STRS_GetTimeAdjust().

Rationale — For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.

Category — Portability

Traced-from — 4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2

Use Case — None

Applicable to — OE developer: usually platform provider

Notes — The table in the requirement is in NASA-STD-4009A.

Verification Method — Inspection using STRS compliance tool.

See Also — STRS_SetTimeAdjust

Example —

```
STRS_HandleID fromWF = APP_GetHandleID();
STRS_HandleID clkDev = STRS_HandleRequest(fromWF,
  "Name of clock observed");
STRS_TimeAdjust tRate;
tRate = STRS_GetTimeAdjust(fromWF, clkDev);
STRS_TimeWarp drift = tRate / FEEDBACK_COEFFICIENT;
```

7.132 STRS-132 STRS_SetTimeAdjust

Requirement — The STRS infrastructure shall contain a callable STRS_SetTimeAdjust method as described in table 71, STRS_SetTimeAdjust().

Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Applicable to	OE developer: usually platform provider
Use Case	None
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	STRS_GetTimeAdjust
Example	

```
STRS_Result rtn;
STRS_HandleID fromWF = APP_GetHandleID();
STRS_HandleID refDev = STRS_HandleRequest(fromWF,
         ”Name of Reference clock”);
STRS_HandleID tgtDev = STRS_HandleRequest(fromWF,
         ”Name of drifting clock”);
STRS_HandleID defDev = STRS_HandleRequest(fromWF,
         STRS_DEFAULT_CLOCK_NAME);
STRS_Clock_Kind kRef = 1;
STRS_Clock_Kind kTgt = 1;
STRS_TimeWarp refbase, refkind, tgtbase, tgtkind;

rtn = STRS_GetTime(fromWF, refDev, refbase, kRef,
          refKind);
rtn = STRS_GetTime(fromWF, tgtDev, tgtbase, kTgt,
          tgtkind);
STRS_TimeWarp initial_difference = refkind – tgtkind;

while(TRUE)
{
    STRS_Sleep(fromWF, defDev, STRS_DEFAULT_CLOCK_KIND,
               POLL_INTERVAL, false);

    rtn = STRS_GetTime(fromWF, refDev, refbase,
             kRef, refkind);
    rtn = STRS_GetTime(fromWF, tgtDev, tgtbase,
             kTgt, tgtkind);
    STRS_TimeWarp drift = (refkind – tgtkind) –
             initial_difference;
    STRS_TimeAdjust tRate = drift *
FEEDBACK_COEFFICIENT;
    rtn = STRS_SetTimeAdjust(fromWF, tgtDev,tRate);
}
```

7.133 STRS-133 STRS_Sleep

Requirement	The STRS infrastructure shall contain a callable STRS_Sleep method as described in table 72, STRS_Sleep().
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE developer: usually platform provider
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection using STRS compliance tool.
See Also	STRS_GetTimeWarp

Example	`STRS_TimeWarp delta;`
	`STRS_Seconds isec = 1;`
	`STRS_Nanoseconds nsec = 0;`
	`delta = STRS_GetTimeWarp(isec,nsec);`
	`STRS_Result ret = STRS_ERROR;`
	`ret = STRS_Sleep(fromID, clockID, 0, delta, false);`
	`// Let's try again with absolute time.`
	`STRS_TimeWarp nowT, nowK;`
	`ret = STRS_GetTime(fromID,clockID,nowt,0,nowK);`
	`isec += STRS_GetSeconds(nowT);`
	`nsec += STRS_GetNanoseconds(nowT);`
	`nowK = STRS_GetTimeWarp(isec,nsec);`
	`ret = STRS_Sleep(fromID, clockID, 0, nowK, true);`

7.134 STRS-134 STRS Platform Queryable Parameters

Requirement	The STRS infrastructure shall have the queryable parameter names in table 75 for which values may be obtained using STRS_Query with the handle ID corresponding to the handle name STRS_OE_HANDLE_NAME.
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	OE
Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection and Test.

Example

```
STRS_Result rtn;
STRS_HandleID fromWF = APP_GetHandleID();
char name1[] = "STRS_OE_VERSION";
char value1[STRS_MAX_PROPERTY_VALUE_SIZE+1];
char name2[] = "STRS_PLATFORM_PROVIDER ";
char value2[STRS_MAX_PROPERTY_VALUE_SIZE+1];
STRS_HandleID oeID =
    STRS_HandleRequest(fromWF,STRS_OE_HANDLE_NAME);
rtn = STRS_ValidateHandleID(oeID);
if (STRS_IsOK(rtn)) {
     rtn = STRS_Query(fromWF, oeID, name1, value1,
                   STRS_MAX_PROPERTY_VALUE_SIZE);
     if ( STRS_IsOK(rtn)) {
          print ("%s = %s\n", name1,value1);
     } else {
          print("STRS_Query OE error: %s\n",
                 name1);
     }
     rtn = STRS_Query(fromWF, oeID, name2, value2,
                   STRS_MAX_PROPERTY_VALUE_SIZE);
     if ( STRS_IsOK(rtn)) {
          print ("OK: %s = %s\n", name2, value2);
     } else {
          print("STRS_Query OE error: %s\n",
                 name2);
     }

} else {
     // ERROR
      print(
        "STRS_HandleRequest did not work
         for %s.\n",STRS_OE_HANDLE_NAME);
     return STRS_ERROR;

}
```

7.135 STRS-135 STRS Application Queryable Parameters

Requirement	An STRS application shall have the queryable parameter names in table 76 for which values may be obtained using STRS_Query with the handle ID of the application.
Rationale	For an open architecture to support portability, the architecture has to be standardized across platforms and implementations. In particular, waveform applications and services have to use standard interfaces across all platforms.
Category	Portability
Traced-from	4.1, 4.2, 4.3, 4.4, 4.5, 4.8, 4.9, 5.1, 5.2
Use Case	None
Applicable to	Application developer

Notes	The table in the requirement is in NASA-STD-4009A.
Verification Method	Inspection and Test.
Example	

```
STRS_Result rtn;
STRS_HandleID fromWF = APP_GetHandleID();
char name1[] = "STRS_APP_VERSION";
char value1[STRS_MAX_PROPERTY_VALUE_SIZE+1];
char name2[] = " STRS_APP_DEVELOPER";
char value2[STRS_MAX_PROPERTY_VALUE_SIZE+1];
char name3[] = “STRS_APP_STATE”;
char value3[STRS_MAX_PROPERTY_VALUE_SIZE+1];
STRS_HandleID wfID =
    STRS_HandleRequest(fromWF,”WF1”);
rtn = STRS_ValidateHandleID(wfID);
if (STRS_IsOK(rtn)) {
      rtn = STRS_Query(fromWF, wfID, name1, value1,
                  STRS_MAX_PROPERTY_VALUE_SIZE);
      if ( STRS_IsOK(rtn)) {
            print ("OK: %s = %s\n", name1,value1);
      } else {
            print("STRS_Query WF error: %s\n",
                  name1);
      }
      rtn = STRS_Query(fromWF, wfID, name2, value2,
                  STRS_MAX_PROPERTY_VALUE_SIZE);
      if ( STRS_IsOK(rtn)) {
            print (OK: "%s = %s\n", name2, value2);
      } else {
            print("STRS_Query WF error: %s\n",
                  name2);
      }
      rtn = STRS_Query(fromWF, wfID, name3, value3,
                  STRS_MAX_PROPERTY_VALUE_SIZE);
      if ( STRS_IsOK(rtn)) {
            print ("OK: %s = %s\n", name3, value3);
      } else {
            print("STRS_Query WF error: %s\n",
                  name3);
      }

} else {
      // ERROR
      print(
        "STRS_HandleRequest did not work for %s.\n",
        “WF1”);
      return STRS_ERROR;
}
```

APPENDIX A

EXAMPLE CONFIGURATION FILES

A.1 STRS Platform Configuration File Hardware Example

Appendix A introduces examples of platform and application configuration files, necessary for application execution and platform initialization. Appendix A also describes example configuration file formats. STRS configuration files contain platform- and application-specific information for the customization of installed applications. These examples are not required formats. They are intended to illustrate some considerations that STRS platform providers and STRS application developers should take into account when designing their configuration file formats.

An example of the format of the portion of an STRS platform configuration file that deals with hardware is implemented in an XML schema. This format is shown in figure 14, Example of Hardware Portion of STRS Platform Configuration File.

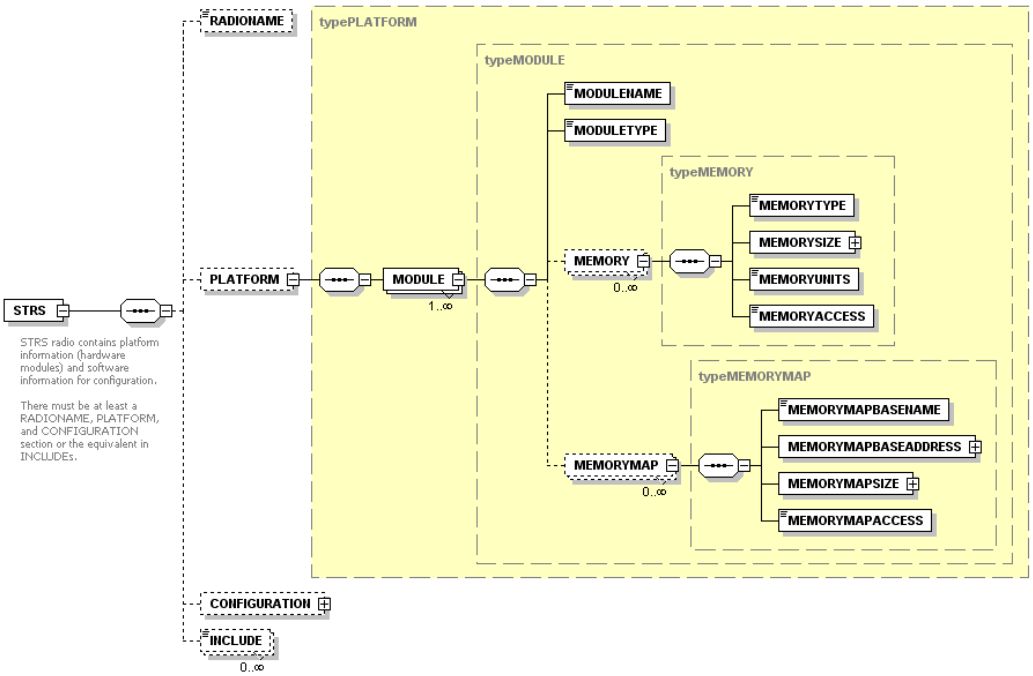

Figure 14—Example of Hardware Portion of STRS Platform Configuration File

For any GPP, the memory size and memory location should be specified in bytes. *Rationale for International Standard—Programming Languages—C states the following:*

(1) “All objects in C must be representable as a contiguous sequence of bytes, each of which is at least 8 bits wide.”

(2) “Any object can be treated as an array of characters, the size of which is given by the sizeof operator with that object’s type as its operand.”

(3) “It is fundamental to the correct usage of functions, such as malloc and fread that sizeof(char) be exactly one.”

Therefore, for consistency across C and C++ implementations, bytes are used.

MODULE list	A list of hardware modules having memory able to contain data and executable software.
• MODULENAME	The unique name for each hardware module accessible from the current GPP. The current GPP is denoted by SELF.
• MODULETYPE	The name of the hardware type. The hardware module types may be GPP, RF, FPGA, DSP, ASIC, and so forth.
• MEMORY list	A list of memory areas of various types. See below for further information.
o MEMORYTYPE	Memory type may be RAM, EEPROM, etc.
o MEMORYSIZE	The number of memory units.
o MEMORYUNITS	Memory units may be BYTES or GATES. For any GPP, the size is to be in BYTES.
o MEMORYACCESS	Memory access for the memory. Access may be READ, WRITE, or BOTH.
• MEMORYMAP list	This list provides the base addresses and memory size of regions of the current GPP RAM (SELF) that are memory mapped to the module: that is, memory mapped to an external device. There may be more than one item in the list when different parts of memory are either not contiguous or are used for different purposes. See section A.2, under DEVICE list, in ATTRIBUTE list, for memory offsets specific to the device associated with a name.
o MEMORYBASENAME	A unique identifier for the portion of memory mapped to the module.
o MEMORYBASEADDRESS	The starting byte address reserved for memory mapping.
o MEMORYSIZE	Number of bytes starting at the base address reserved for memory mapping.
o MEMORYACCESS	Memory access for the portion of memory mapped to the module. Access may be READ, WRITE, or BOTH. The access defined here may be different from the memory access defined in the previous section when part of the memory is used for one purpose and another part is used for a different purpose.

A.2 STRS Platform Configuration File Software Example

An example of the format of the portion of an STRS platform configuration file that deals with software is implemented in an XML schema. This format is shown in figure 15, Example of Software Portion of STRS Platform Configuration File.

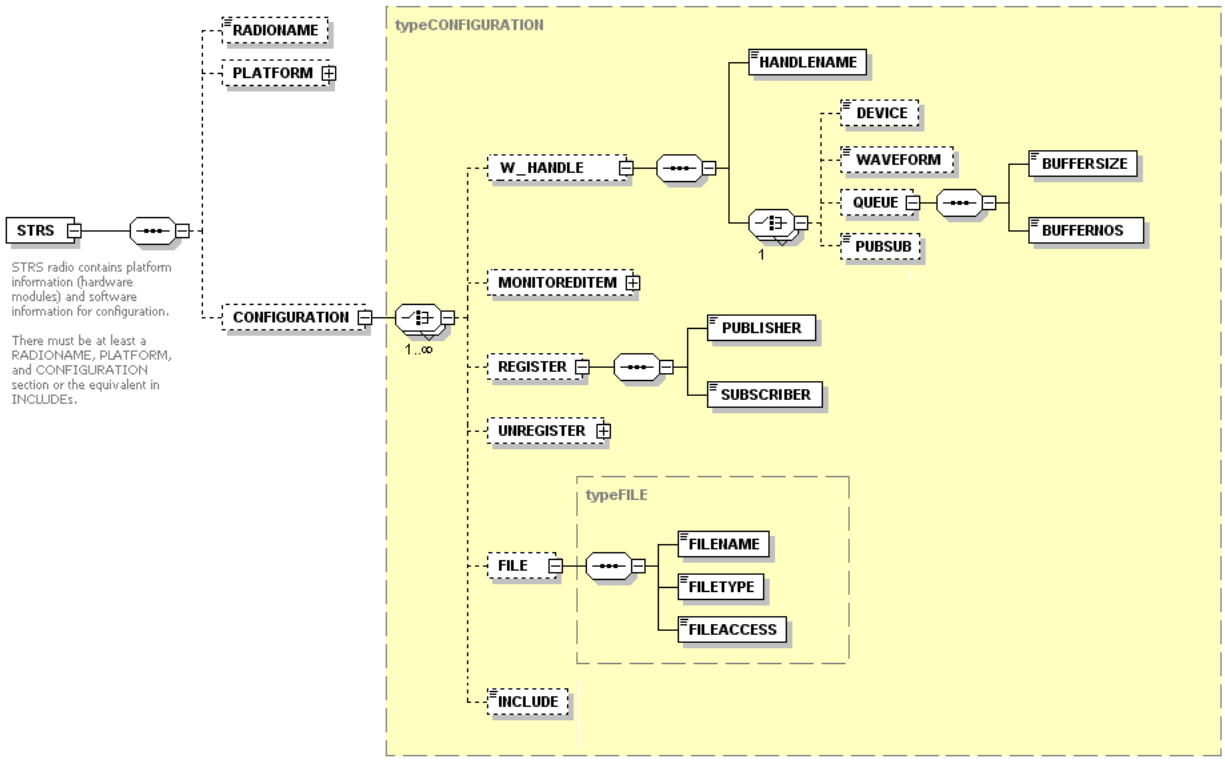

Figure 15—Example of Software Portion of STRS Platform Configuration File

FILE list	**A list of files to read, write, both, or append from multiple locations using a handle ID.**
• **FILENAME**	Storage area name or fully qualified file name.
• **FILETYPE**	The file type may be TEXT or BINARY.
• **FILEACCESS**	The file access may be READ, WRITE, BOTH, or APPEND. BOTH implies update: that is, READ and WRITE.
W_HANDLE list	This is a list of unique STRS handle names and corresponding objects.
• **HANDLENAME**	A unique shortened form of the Device, Waveform, Queue, Pub/Sub name to use as a character identifier to be used in messages and for obtaining the handle ID.
• **DEVICE**	OE-specific name to instantiate the device. A device is software that acts as a proxy for some hardware connection.
• **WAVEFORM**	OE-specific name to instantiate the application.
• **QUEUE list**	The information necessary to create a FIFO queue.
o BUFFERSIZE	The maximum size of messages in queue.
o BUFFERNOS	The maximum number of messages in queue.
• **PUBSUB list**	OE-specific name for the Pub/Sub..

REGISTER list	The correspondences between queues and subscribers. This couples publishers and subscribers. UNREGISTER is the same as REGISTER except it decouples publishers from subscribers.
• **PUBLISHER**	The name of the queue that the publisher uses to send data to the subscribers. Used in messages and for obtaining the handle ID.
• **SUBSCRIBER**	A handle name for a subscriber. Used in messages and for obtaining the handle ID.
MONITOREDITEM list	A list of monitored items that are tested to indicate the health of the system.
• **ATTRIBUTENAME**	The name of the property whose value is to be tested in a monitored component.
• **HANDLENAME**	The handle name defines the monitored component from which to obtain the value corresponding to the attributeName.
• **DELAY**	A positive value represents the nominal time delay between successive automated tests of the monitored component. A nonpositive value indicates that the test is to be requested.
• **TESTTYPE**	The type of test to apply to the property to ascertain whether the value indicates the monitored component is healthy. Examples include testing for exact values, within ranges, or by use of operations in Reverse Polish Notation (RPN).
o **EXACT**	Monitored value is to be one of the values in the value list.
o **EXCLUDE**	Monitored value is not to be in the value list.
o **BETWEENII**	Monitored value is to be between the pairs of values in the value list including both end points.
o **BETWEENIX**	Monitored value is to be between the pairs of values in the value list including the low end point and excluding the high end point.
o **BETWEENXI**	Monitored value is to be between the pairs of values in the value list excluding the low end point and including the high end point.
o **BETWEENXX**	Monitored value is to be between the pairs of values in the value list excluding both end points.
o **RPN**	The attributeName, values to be tested, and operators are to appear in the value list using RPN. RPN uses sequences of one or two arguments followed by an operator. The result of applying the operator replaces the original sequence used, and the process is repeated until there are no more operators. The attributeName for the monitored value is replaced, in the RPN formula, by the corresponding property value. For example, the sequence of data and operators in the VALUE list for testing the property named D in RPN—0;D;LT;D;500;LE;AND—is equivalent to (0<D && D≤500) The current set of operators includes the following: AND, OR, XOR, NOT, EQ, NE, GT, GE, LT, LE, PLUS, MINUS, MULTIPLY, DIVIDE, MOD, MIN, MAX. If floating point is required or allowed, the set of operators could be augmented with the following: SIN, COS, TAN, ASIN, ACOS, ATAN1, ATAN2, SINH, COSH, TANH, ABS, EXP, LOG10, LN, SQRT, FLOOR, CEIL, ROUND, POW.
• **VALUE list**	A list of values and possibly operations used corresponding to the value of TESTTYPE. o For example, if TESTTYPE is EXACT, the VALUE list will contain {512,1024,2048,4096} if those are the allowed values. o If TESTTYPE is EXCLUDE and odd numbers between 1 and 10 are not allowed, the VALUE list will contain {1,3,5,7,9}. o If TESTTYPE is BETWEENII and the attribute D is allowed between 0 and 500, inclusive (0 ≤ D ≤ 500), the VALUE list will contain {0,500}. Because the TESTTYPE is BETWEENII, the lower limit, 0, is included and the upper limit, 500, is included. o If TESTTYPE is BETWEENIX and the attribute D is allowed between 0 and 500 (0 ≤ D < 500), the VALUE list will contain {0,500}. Because the

	TESTTYPE is BETWEENIX, the lower limit, 0, is included and the upper limit, 500, is excluded. o If TESTTYPE is BETWEENXI and the attribute D is allowed between 0 and 500 ($0 < D \leq 500$), the VALUE list will contain {0,500}. Because the TESTTYPE is BETWEENXI, the lower limit, 0, is excluded and the upper limit, 500, is included. o If TESTTYPE is BETWEENXX and the attribute D is allowed between 0 and 500, exclusive ($0 < D < 500$), the VALUE list will contain {0,500}. Because the TESTTYPE is BETWEENXX, the lower limit, 0, is excluded and the upper limit, 500, is excluded. o If TESTTYPE is RPN and the attribute D is allowed between 0 and 500 ($0 < D \leq 500$), the VALUE list will contain {0,D,LT,D,500,LE,AND}.

A.3 STRS Application Configuration File Example

An example of the format of an independent STRS application configuration file in XML is shown in Figure 16, Example of STRS Waveform Configuration File.

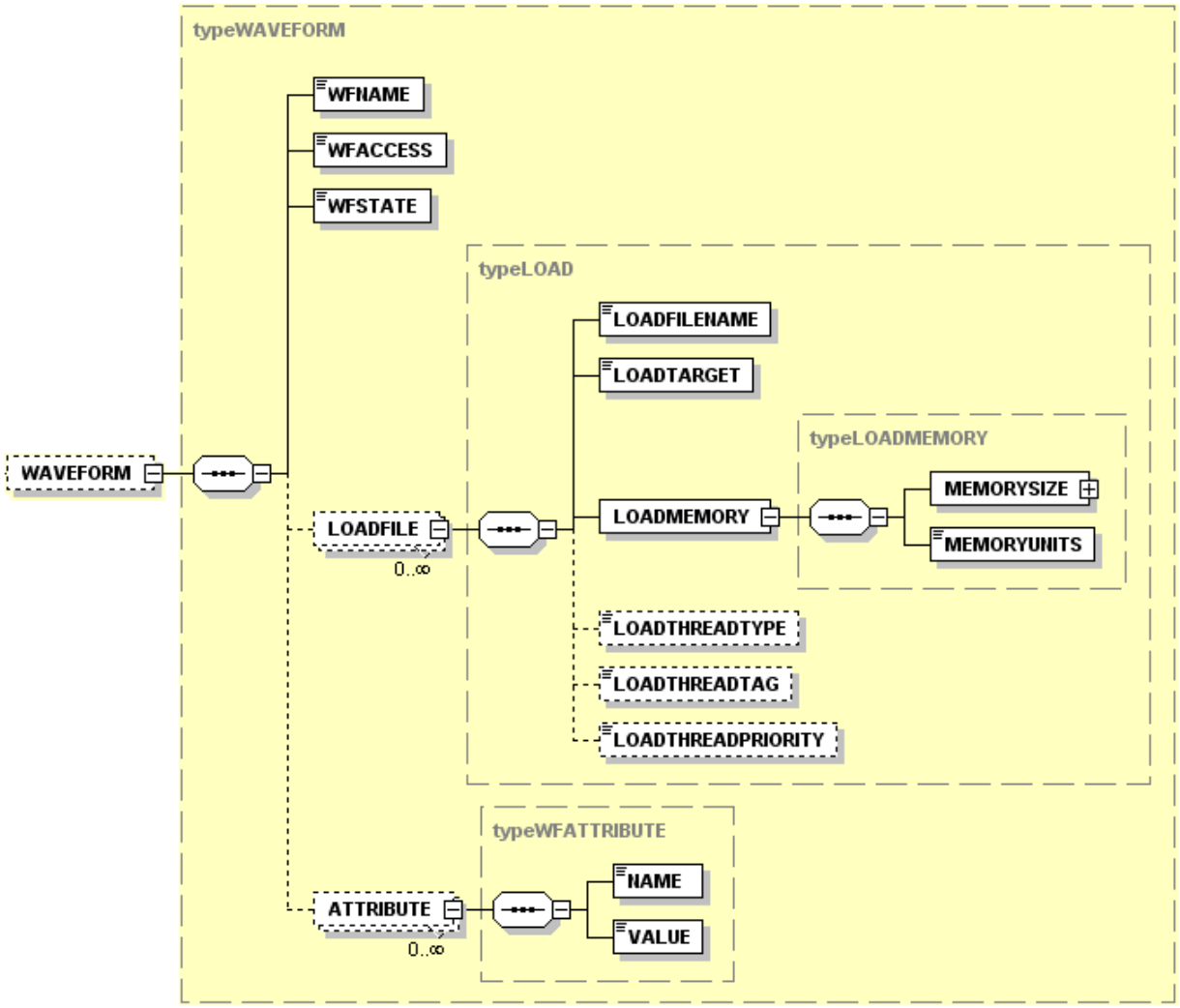

Figure 16—Example of STRS Waveform Configuration File

WAVEFORM list	
• WFNAME	If coded in C++, this is the application class name. If coded in C, this is an equivalent class name.
• WFACCESS	The access to the application may be specified as READ, WRITE, BOTH, or NONE. READ indicates that the application implements APP_Read(). WRITE indicates that the application implements APP_Write().
• WFSTATE	The state at which the application is left after processing the configuration file. The state may be INSTANTIATED, STOPPED, or RUNNING.
• LOADFILE list	A list of files to be loaded for execution if not already loaded. Usually, the software for the application on the current GPP (SELF) should be loaded before the configurable hardware design so that the software can load and configure the software or configurable hardware design as necessary.
o LOADFILENAME	Storage area name or fully qualified file name
o LOADTARGET	Module name for the device on which the file is instantiated. The load process is determined by the corresponding MODULE information (see A.1).
o LOADMEMORY	
• MEMORYSIZE	The number of memory units.
• MEMORYUNITS	Memory units may be BYTES or, GATES. For any GPP, the size is to be in BYTES.
o LOADTHREADTYPE	
o LOADTHREADTAG	
o LOADTHREADPRIORITY	
• ATTRIBUTE list	A list of properties set as default during initialization.
o NAME	Name of the attribute
o VALUE	Value of the attribute

The format of an independent STRS device configuration file in XML would be similar. An example of the format of an independent STRS application configuration file in XML is shown in Figure 17, Example of STRS Device Configuration File.

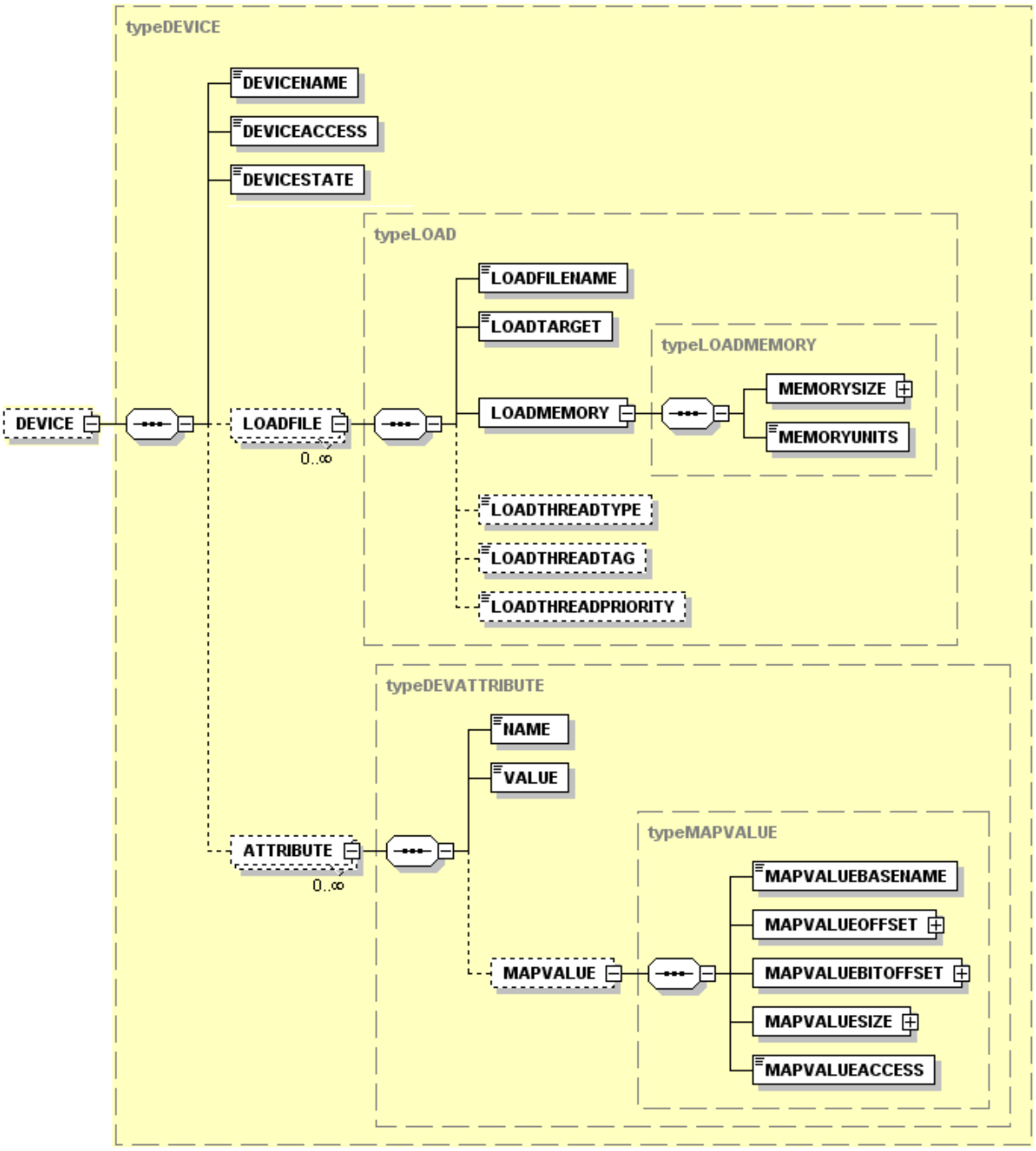

Figure 17—Example of STRS Device Configuration File

The difference being that the ATTRIBUTE tag could have additional parameters describing memory mapping.

• ATTRIBUTE list	A list of properties set as default during initialization.
o NAME	Name of the attribute.
o VALUE	Value of the attribute.
o MAPVALUE list	Location in memory of the attribute when memory mapped. A location is to be unique to the associated device.
▪ MAPVALUEBASENAME	A unique identifier for the portion of memory mapped to the module. This is to match a MEMORYBASENAME value defined in section A.1, under MODULE list in the MEMORYMAP list.
▪ MAPVALUEOFFSET	Offset from the address of baseName as defined in the module list's memory map list.
▪ MAPVALUEBITOFFSET	Bit offset from the high order position to begin.
▪ MAPVALUESIZE	Number of bits in which to store the value.
▪ MAPVALUEACCESS	Memory access may be READ, WRITE, or BOTH.

APPENDIX B

REFERENCES

B.1 Purpose and/or Scope

This Appendix provides reference documents that provide additional information relative to this NASA Technical Handbook.

B.2 References

	Rationale for International Standard—Programming Languages—C
XML 1.0, XML 1.1	W3C
DO-178	Software Considerations in Airborne Systems and Equipment Certification (RTCA, Incorporated)