



AEROSPACE STANDARD

AS5710**REV. A**

Issued	2008-12
Revised	2010-08
Reaffirmed	2015-04

Superseding AS5710

(R) JAUS Core Service Set

RATIONALE

AS5710A has been reaffirmed to comply with the SAE five-year review policy.

INTRODUCTION

This document, the JAUS Core Service Set (AS5710), defines a message-passing interface for capabilities and services commonly found in unmanned systems. These services represent the foundation on which more complex and domain-specific JAUS Service Sets can be developed.

The primary goal of the JAUS Core Service Set is logical interoperability between communicating elements in an unmanned system. To this end, each service defines the messages (vocabulary) and protocol (rules) for data exchange. This logical interoperability is independent of the physical transport, and it is expected that a Transport Standard, such as the JAUS Transport Specification [AS5669], is used in conjunction with this specification.

Each service in the JAUS Core Service Set can be described using the JAUS Service Interface Definition Language [JSIDL]. JSIDL creates a formal schema based on Relax NG Compact [rng] that allows for validation of each service definition described herein. Although knowledge of JSIDL is not required to understand or implement this Specification, it is highly recommended for supporting context. For convenience, the JAUS Core Service Set contains both a text based and XML based representation for each service.

The AS5710A Standard does not replace the latest JAUS Reference Architecture Version 3.3 [RA33P1, RA33P2, RA33P3], but is a direct evolution of that work. This Standard has been carefully designed to allow for the simplest possible migration of RA 3.3 implementations to a services-based framework. Even though the notion of services has come to define formal interfaces between components, the message sets in those services trace back to identical messages in the Reference Architecture.

This document uses a number of conventions to simplify the text. All names are given in Camel Case. Names start with upper case, while reference names start with a lower case.

SAE Technical Standards Board Rules provide that: "This report is published by SAE to advance the state of technical and engineering sciences. The use of this report is entirely voluntary, and its applicability and suitability for any particular use, including any patent infringement arising therefrom, is the sole responsibility of the user."

SAE reviews each technical report at least every five years at which time it may be revised, reaffirmed, stabilized, or cancelled. SAE invites your written comments and suggestions.

Copyright © 2015 SAE International

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of SAE.

TO PLACE A DOCUMENT ORDER: Tel: 877-606-7323 (inside USA and Canada)
Tel: +1 724-776-4970 (outside USA)
Fax: 724-776-0790
Email: CustomerService@sae.org
http://www.sae.org

SAE WEB ADDRESS:

Full document details: SAE AS5710A

SAE values your input. Provide feedback on this Technical Report, please visit <http://www.sae.org/technical/standards/AS5710A>

Documents
Document Sharing Hub

TABLE OF CONTENTS

1.	SCOPE.....	6
1.1	Purpose.....	6
1.2	Compliance.....	6
1.3	Document Organization.....	6
2.	REFERENCES.....	7
2.1	Applicable Documents.....	7
2.1.1	SAE Publications.....	7
2.1.2	JAUS Technical References.....	7
2.1.3	Other Publications.....	7
2.2	Definitions.....	7
2.2.1	Component.....	7
2.2.2	Service Definition.....	7
2.2.3	Service Identifier.....	8
2.2.4	JAUS Identifier.....	8
2.2.5	Transport Layer.....	8
2.2.6	Serialization.....	8
2.2.7	Message Code.....	8
2.2.8	Presence Vector.....	8
2.2.9	Little Endian.....	8
2.3	Constant values.....	8
2.4	List of Acronyms.....	8
3.	COMMON CONVENTIONS.....	9
3.1	Scaled Integers.....	9
3.2	Presence Vectors.....	9
3.3	Variable Length Arrays and Lists.....	10
3.4	Examples of List Manipulation.....	10
3.4.1	Case 1: Inserting into an Existing List.....	11
3.4.2	Case 2: Insert Failures.....	11
3.4.2	Case 3: Deleting from an Existing List.....	11
4.	UNDERSTANDING PROTOCOL DESCRIPTIONS.....	11
4.1	States.....	12
4.2	Transitions.....	12
5.	SERVICE DEFINITIONS.....	15
5.1	Transport.....	16
5.1.1	Description.....	16
5.1.2	Assumptions.....	17
5.1.3	Vocabulary.....	17
5.1.4	Encoding.....	18
5.1.5	Protocol Behavior.....	20
5.2	Events.....	21
5.2.1	Description.....	22
5.2.2	Assumptions.....	22
5.2.3	Vocabulary.....	22
5.2.4	Encoding.....	23
5.2.5	Protocol Behavior.....	29
5.3	AccessControl.....	31
5.3.1	Description.....	31
5.3.2	Assumptions.....	32
5.3.3	Vocabulary.....	32
5.3.4	Encoding.....	32

5.3.5	Protocol Behavior	37
5.4	Management	39
5.4.1	Description	40
5.4.2	Assumptions.....	40
5.4.3	Vocabulary	40
5.4.4	Encoding	41
5.4.5	Protocol Behavior	44
5.5	Time	46
5.5.1	Description	46
5.5.2	Assumptions.....	46
5.5.3	Vocabulary	46
5.5.4	Encoding	47
5.5.5	Protocol Behavior	48
5.6	Liveness	49
5.6.1	Description	49
5.6.2	Assumptions.....	49
5.6.3	Vocabulary	50
5.6.4	Encoding	50
5.6.5	Protocol Behavior	50
5.7	Discovery	51
5.7.1	Description	51
5.7.2	Assumptions.....	52
5.7.3	Vocabulary	52
5.7.4	Encoding	52
5.7.5	Protocol Behavior	60
5.8	List Manager Service	62
5.8.1	Description	62
5.8.2	Assumptions.....	63
5.8.3	Vocabulary	63
5.8.4	Encoding	63
5.8.5	Protocol Behavior	68
6.	NOTES	70
6.1	Change bar (l) and (R) symbols	70
APPENDIX A	XML FOR SERVICE DEFINITIONS	71
APPENDIX B	XML FOR DECLARED TYPE SETS.....	102
FIGURE 1	EXAMPLE SCENARIO FOR LIST MANIPULATION.....	10
FIGURE 2	EXAMPLE SCENARIO FOR INSERTING A NEW ELEMENT	10
FIGURE 3	EXAMPLE SCENARIO FOR CIRCULAR LIST SEGMENTS	11
FIGURE 4	FAILURE CASE LEAVING AN EXISTING ELEMENT ORPHANED.....	11
FIGURE 5	EXAMPLE SCENARIO FOR DELETING AN EXISTING ELEMENT	11
FIGURE 6	PROTOCOL STATE MACHINE.....	12
FIGURE 7	NESTED STATES WITH SIMPLE AND INTERNAL TRANSITIONS	14
FIGURE 8	PUSH AND POP TRANSITIONS.....	14
FIGURE 9	CONCURRENT FINITE STATE MACHINES	15
FIGURE 10	THE SYSTEM TOPOLOGY	16
FIGURE 11	TRANSPORT SERVICE ACTING AS AN INTERFACE TO THE TRANSPORT LAYER	17
FIGURE 12	TRANSPORT SERVICE PROTOCOL BEHAVIOR.....	21
FIGURE 13	EVENTS SERVICE	22
FIGURE 14	EVENTS SERVICE PROTOCOL BEHAVIOR.....	29
FIGURE 15	ACCESS CONTROL SERVICE.....	31
FIGURE 16	ACCESS CONTROL SERVICE PROTOCOL BEHAVIOR.....	37
FIGURE 17	MANAGEMENT SERVICE	40
FIGURE 18	MANAGEMENT SERVICE PROTOCOL BEHAVIOR	44
FIGURE 19	TIME SERVICE.....	46
FIGURE 20	TIME SERVICE PROTOCOL BEHAVIOR.....	48

FIGURE 21	TIME SERVICE TRANSITION TABLE	48
FIGURE 22	LIVENESS SERVICE	49
FIGURE 23	LIVENESS SERVICE PROTOCOL BEHAVIOR	50
FIGURE 24	DISCOVERY SERVICE	51
FIGURE 25	DISCOVERY SERVICE PROTOCOL BEHAVIOR	60
FIGURE 26	SEQUENCE DIAGRAM OF SERVICE DISCOVERY	61
FIGURE 27	LIST MANAGER SERVICE	62
FIGURE 28	LIST MANAGER SERVICE PROTOCOL BEHAVIOR	68
TABLE 1	TRANSITION TABLE	13
TABLE 2	CONDITION TABLE	13
TABLE 3	ACTION TABLE	13
TABLE 4	TYPES OF TRANSITIONS FOR WHICH ENTRY/EXIT ACTIONS ARE EXECUTED	15
TABLE 5	TRANSPORT SERVICE INTERNAL EVENTS SET	18
TABLE 6	RECEIVE EVENT ENCODING	18
TABLE 7	SEND EVENT ENCODING	19
TABLE 8	BROADCAST LOCAL EVENT ENCODING	20
TABLE 9	BROADCAST GLOBAL EVENT ENCODING	20
TABLE 10	SEND ERROR EVENT ENCODING	20
TABLE 11	TRANSPORT SERVICE STATE TRANSITION TABLE	21
TABLE 12	TRANSPORT SERVICE CONDITIONS	21
TABLE 13	TRANSPORT SERVICE TRANSITION ACTIONS	21
TABLE 14	EVENTS SERVICE VOCABULARY	22
TABLE 15	EVENTS SERVICE INTERNAL EVENTS SET	23
TABLE 16	CREATE EVENT MESSAGE ENCODING	23
TABLE 17	UPDATE EVENT MESSAGE ENCODING	24
TABLE 18	CANCEL EVENT MESSAGE ENCODING	24
TABLE 19	CREATE EVENT MESSAGE ENCODING	25
TABLE 20	QUERY EVENTS MESSAGE ENCODING	25
TABLE 21	QUERY EVENT TIMEOUT MESSAGE ENCODING	26
TABLE 22	CONFIRM EVENT REQUEST MESSAGE ENCODING	26
TABLE 23	REJECT EVENT REQUEST MESSAGE ENCODING	27
TABLE 24	REPORT EVENTS MESSAGE ENCODING	27
TABLE 25	EVENT MESSAGE ENCODING	28
TABLE 26	REPORT EVENT TIMOUT MESSAGE ENCODING	28
TABLE 27	COMMAND EVENT MESSAGE ENCODING	28
TABLE 28	EVENTS SERVICE STATE TRANSITION TABLE	30
TABLE 29	EVENTS SERVICE CONDITIONS	30
TABLE 30	EVENTS SERVICE TRANSITION ACTIONS	31
TABLE 31	ACCESS CONTROL SERVICE VOCABULARY	32
TABLE 32	ACCESS CONTROL SERVICE INTERNAL EVENTS SET	32
TABLE 33	REQUEST CONTROL MESSAGE ENCODING	33
TABLE 34	RELEASE CONTROL MESSAGE ENCODING	33
TABLE 35	QUERY CONTROL MESSAGE ENCODING	33
TABLE 36	QUERY AUTHORITY MESSAGE ENCODING	33
TABLE 37	SET AUTHORITY MESSAGE ENCODING	34
TABLE 38	QUERY TIMEOUT MESSAGE ENCODING	34
TABLE 39	REPORT CONTROL MESSAGE ENCODING	34
TABLE 40	REJECT CONTROL MESSAGE ENCODING	35
TABLE 41	CONFIRM CONTROL MESSAGE ENCODING	35
TABLE 42	REPORT AUTHORITY MESSAGE ENCODING	35
TABLE 43	REPORT TIMEOUT MESSAGE ENCODING	36
TABLE 44	ACCESS CONTROL SERVICE STATE TRANSITIONS	38
TABLE 45	ACCESS CONTROL SERVICE CONDITIONS	39
TABLE 46	ACCESS CONTROL SERVICE TRANSITION ACTIONS	39
TABLE 47	MANAGEMENT SERVICE VOCABULARY	40
TABLE 48	MANAGEMENT SERVICE INTERNAL EVENTS SET	41
TABLE 49	SHUTDOWN MESSAGE ENCODING	41
TABLE 50	STANDBY MESSAGE ENCODING	41

TABLE 51	RESUME MESSAGE ENCODING	41
TABLE 52	RESET MESSAGE ENCODING	42
TABLE 53	SET EMERGENCY MESSAGE ENCODING	42
TABLE 54	CLEAR EMERGENCY MESSAGE ENCODING	42
TABLE 55	QUERY STATUS MESSAGE ENCODING	43
TABLE 56	REPORT STATUS MESSAGE ENCODING	43
TABLE 57	MANAGEMENT SERVICE TRANSITION TABLE	45
TABLE 58	MANAGEMENT SERVICE CONDITIONS	45
TABLE 59	MANAGEMENT SERVICE TRANSITION ACTIONS	45
TABLE 60	TIME SERVICE VOCABULARY	46
TABLE 61	SET TIME MESSAGE ENCODING	47
TABLE 62	QUERY TIME MESSAGE ENCODING	47
TABLE 63	REPORT TIME MESSAGE ENCODING	48
TABLE 64	TIME SERVICE CONDITIONS TABLE	49
TABLE 65	TIME SERVICE TRANSITION ACTIONS	49
TABLE 66	LIVENESS SERVICE VOCABULARY	50
TABLE 67	QUERY HEARTBEAT PULSE MESSAGE ENCODING	50
TABLE 68	REPORT HEARTBEAT PULSE MESSAGE ENCODING	50
TABLE 69	LIVENESS SERVICE STATE TRANSITIONS	51
TABLE 70	LIVENESS SERVICE TRANSITION ACTIONS	51
TABLE 71	DISCOVERY SERVICE VOCABULARY	52
TABLE 72	REGISTER SERVICES MESSAGE ENCODING	52
TABLE 73	QUERY IDENTIFICATION MESSAGE ENCODING	53
TABLE 74	QUERY CONFIGURATION MESSAGE ENCODING	53
TABLE 75	QUERY SUBSYSTEM LIST MESSAGE ENCODING	53
TABLE 76	QUERY SERVICES MESSAGE ENCODING	54
TABLE 77	QUERY SERVICE LIST MESSAGE ENCODING	55
TABLE 78	REPORT IDENTIFICATION MESSAGE ENCODING	56
TABLE 79	REPORT CONFIGURATION MESSAGE ENCODING	57
TABLE 80	REPORT SUBSYSTEM LIST MESSAGE ENCODING	57
TABLE 81	REPORT SERVICES MESSAGE ENCODING	58
TABLE 82	REPORT SERVICE LIST MESSAGE ENCODING	59
TABLE 83	DISCOVERY SERVICE STATE TRANSITIONS	60
TABLE 84	DISCOVERY SERVICE TRANSITION ACTIONS	60
TABLE 85	LIST MANAGER SERVICE VOCABULARY	63
TABLE 86	SET ELEMENT MESSAGE ENCODING	64
TABLE 87	DELETE ELEMENT MESSAGE ENCODING	65
TABLE 88	QUERY ELEMENT MESSAGE ENCODING	65
TABLE 89	QUERY ELEMENT LIST MESSAGE ENCODING	66
TABLE 90	QUERY ELEMENT COUNT MESSAGE ENCODING	66
TABLE 91	CONFIRM ELEMENT REQUEST MESSAGE ENCODING	66
TABLE 92	REJECT ELEMENT REQUEST MESSAGE ENCODING	67
TABLE 93	REPORT ELEMENT MESSAGE ENCODING	67
TABLE 94	REPORT ELEMENT LIST MESSAGE ENCODING	67
TABLE 95	REPORT ELEMENT COUNT MESSAGE ENCODING	68
TABLE 96	LIST MANAGER SERVICE TRANSITION TABLE	69
TABLE 97	LIST MANAGER SERVICE STATE CONDITIONS TABLE	69
TABLE 98	LIST MANAGER SERVICE TRANSITION ACTIONS	70

1. SCOPE

This document defines a set of standard application layer¹ interfaces called *JAUS Core Services*. JAUS Services provide the means for software entities in an unmanned system or system of unmanned systems to communicate and coordinate their activities. The Core Services represent the infrastructure commonly found across all domains and types of unmanned systems. At present, eight services are defined in this document:

- Transport Service: Abstracts the functionality of the underlying communication transport layer
- Events Service: Establishes a publish/subscribe mechanism for automatic messaging
- Access Control: Manages preemptable exclusive control for safety critical operations
- Management: Defines component life-cycle management
- Time: Allows clients to query and set the system time for the component
- Liveness: Provides a means to maintain connection liveness between communicating components
- Discovery: Governs automatic discovery of remote entities and their capabilities
- List Manager: Encompasses behavior common to doubly linked lists

Each service is described by a JAUS Service Definition (JSD) which specifies the message set and protocol required for compliance. Each JSD is fully compliant with the JAUS Service Interface Definition Language [JSIDL].

1.1 Purpose

The purpose of this document is to facilitate interoperation of unmanned vehicle systems, subsystems, and payloads by standardization of the message set and associated protocol.

1.2 Compliance

The JAUS Core Service Set must support compliance assessment. To do so, this specification must be sufficiently precise to enable the “compliant”/“not compliant” distinction to be made independently of the underlying transport mechanism. An implementation is compliant if it strictly adheres to the vocabulary and protocol of the JAUS Service Definition. Sequence diagrams, use cases, and text descriptions are informational only.

It is important to note that implementations are considered compliant to individual Service Definitions within this Specification; it is not necessary that a single entity realize each Service to be considered compliant.

1.3 Document Organization

The layout of this document is as follows. [Section 2](#) lists external references, definitions of common terms and a list of acronyms used throughout the specification. [Section 3](#) specifies aspects of message encoding that are common across all services. Many of these encoding rules are more thoroughly discussed in [JSIDL]. [Section 4](#) provides context for interpreting the protocol portion of a Service Definition. [Section 5](#) specifies the JAUS Service Definition for each of the core services, with particular emphasis on the description, assumptions, message set, and protocol behavior. [Section 6](#) is reserved for general notes. Finally, [Appendix A](#) and [Appendix B](#) contain the complete JSIDL representation for each service and their associated message set.

¹ For a complete description of the two layer approach used by SAE JAUS, please consult [JSIDL](#).



2. REFERENCES

2.1 Applicable Documents

The following publications form a part of this document to the extent specified herein. The latest issue of SAE publications shall apply. The applicable issue of other publications shall be the issue in effect on the date of the purchase order. In the event of conflict between the text of this document and references cited herein, the text of this document takes precedence. Nothing in this document, however, supersedes applicable laws and regulations unless a specific exemption has been obtained.

2.1.1 SAE Publications

Available from SAE International, 400 Commonwealth Drive, Warrendale, PA 15096-0001, Tel: 877-606-7323 (inside USA and Canada) or 724-776-4970 (outside USA), Web address: www.sae.org.

AIR5665 Architecture Framework for Unmanned Systems

AS5669 JAUS/SDP Transport Specification

AS5684A JAUS Service Interface Definition Language

2.1.2 JAUS Technical References

RA33P1 JAUS Reference Architecture Specification, Volume II, Part 1, Architecture Framework, Version 3.3, June 22, 2007

RA33P2 JAUS Reference Architecture Specification, Volume II, Part 2, Message Definition, Version 3.3, June 22, 2007

RA33P3 JAUS Reference Architecture Specification, Volume II, Part 3, Message Set, Version 3.3, June 22, 2007

2.1.3 Other Publications

[att] Decimal Expansion of PI [<http://www.research.att.com/~njas/sequences/A000796>]

[bnt] Barry N. Taylor. The International System of Units (SI), National Institute of Standards and Technology Special Publication 330, 1991 Edition. [<http://physics.nist.gov/Document/sp330.pdf>]

[rng] Relax NG: [<http://www.oasis-open.org/committees/relax-ng/spec-20011203.html>] Standard lightweight XML schema language.

2.2 Definitions

2.2.1 Component

A component is a software element in a JAUS system. A component that provides one or more *services* is called a server. A component that uses one or more services is called a client.

While a component may support multiple services, two instances of the same service cannot co-exist within the same component. If two services that reside within a component inherit from the same base service, it is assumed that only one instance of that base service is present. Inheritance of service interfaces is described in [\[AS5684A\]](#).

2.2.2 Service Definition

A Service Definition is a textual and/or XML representation of a service interface. Any Service Definition shall conform to the JAUS Service Interface Definition Language Schema defined in [\[AS5684A\]](#). Each Service Definition contains a service identifier, version, message set, protocol, and associated information.

2.2.3 Service Identifier

A service identifier is a globally unique string that identifies a specific *Service Definition*. Since a Service Definition mandates a message set and associated protocol, the service identifier and version number are sufficient to uniquely identify the service interface. Service Identifiers are based on a unique URI, and are specified for each of the core services defined herein.

2.2.4 JAUS Identifier

The JAUS Identifier is a 4-byte unsigned integer that corresponds to a communication end point. Messages are sent to, and received from, a JAUS Identifier. A JAUS Identifier may represent a component that hosts multiple services; the Discovery Service therefore permits the run-time determination of this one-to-many mapping between JAUS Identifiers and Service Identifiers. Refer to the [Transport Service](#) for details on the JAUS identifier.

2.2.5 Transport Layer

The JAUS Transport Layer is responsible for providing resources and mechanisms for the routing and delivery of messages over a variety of available transport domains. While a Service Definition is independent of the underlying transport, it is designed for integration with [\[AS5669\]](#). Other transport layers are possible, provided they meet the requirements described in the [Transport Service](#).

2.2.6 Serialization

Serialization (sometimes called encoding or marshalling) is the process by which a message defined by a Service Definition is converted into a well-defined sequence of bytes that will be visible in a given Transport Domain, "on the wire". The reverse process is known as deserialization. Interoperability requires that both sender and receiver follow the same syntax for serialization and deserialization or data corruption may occur.

2.2.7 Message Code

The Message Code is an identifier globally unique to each message. This code, along with the associated service version, allows receiving entities to know the type, intent, and structure of an incoming message. The Message Code is serialized in the same position within each message.

2.2.8 Presence Vector

JAUS provides for variable length messages. These messages either have repeating data or have a mixture of required and optional data fields. The Presence Vector is used to indicate which of the optional data fields are included. In the Service Message tables, optional fields are mapped in order to each bit in the presence vector, starting with bit 0.

2.2.9 Little Endian

A byte-encoding scheme where the least significant byte of a multi-byte data item comes first in the byte sequence.

2.3 Constant values

In the context of this Standard, PI is defined as 3.14159265358979323846 [att].

2.4 List of Acronyms

ANSI	American National Standards Institute
ID	Identifier
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange

BLOB	Binary Large Object
DOM	Document Object Model
IEEE	Institute of Electrical and Electronics Engineers
JAUS	Joint Architecture for Unmanned Systems
JSD	JAUS Service Definition
JSIDL	JAUS Service (Interface) Definition Language
NIST	National Institute of Standards and Technology
RA	(JAUS) Reference Architecture
SMC	State Machine Compiler
UML	Unified Modeling Language
URL	Uniform Resource Locator
URN	Uniform Resource Name
URI	Uniform Resource Identifier
UUID	Universally Unique Identifier
XML	Extensible Markup Language

3. COMMON CONVENTIONS

Any implementation of a JAUS Service Definition must meet the requirements established in [\[AS5684A\]](#) for on-the-wire message serialization:

- All multi-byte data shall be encoded in little endian where the low byte comes first.
- All messages shall be encoded with a 2-byte header that contains the Message Code. This header must precede the message body.

In addition to the above requirements for all messages, JAUS Service Definitions may make use of the following additional serialization rules to improve messaging efficiency.

3.1 Scaled Integers

The use of Scaled Integers reduces the impact of transmitting floating point and double precision values that are valid only in a finite range. Scaled integers are integer types that represent real values. The real values are first bounded by upper and lower limits and are then mapped onto the integer range of the integer field type used for the scaled integer. Each scaled integer is associated with a *bias* and *scale factor*. For additional information on computing the bias and scale factor for scaled integers, consult [\[AS5684A\]](#).

3.2 Presence Vectors

Messages are allowed to have a mixture of required and optional data fields. A presence vector bit is mapped to each optional data field, or data group. Presence vectors can be specified as byte, unsigned short integer, or unsigned integer. The maximum number of bits for a presence vector is 32, which corresponds to an unsigned integer type.

The use of optional data fields is indicated by the “Optional” field for each element in a message definition. The bits of the presence vector map to optional data fields or groups of data fields that follow in the message data specification. This mapping is implicit, such that the least significant bit of the presence vector maps to the first optional field, with each additional optional field mapping to the next available least significant bit. If the optional data field/group is present in the message, the bit corresponding to that field/group in the presence vector is enabled. Otherwise, the corresponding bit shall be set to zero (0).

Unused bits in the presence vector shall be set to zero (0).

3.3 Variable Length Arrays and Lists

Many messages make use of variable length fields, such as arrays and lists. Such fields must be serialized with the count field preceeding the corresponding data. The data size of the count field is specified in each message definition.

3.4 Examples of List Manipulation

The List Manager Service introduces the concept of a single sequence of connected elements, similar to a doubly-linked list. Each element within the list includes references to the previous (parent) and next (child) element. The *SetElement* and *DeleteElement* messages allow elements to be inserted, modified, or removed from the list. This section provides examples of permitted actions commonly performed by clients, along with the expected server response. These examples are provided for reference only, and are not intended to an exhaustive case study of all permitted operations.

3.4.1 Case 1: Inserting into an Existing List

Suppose a list manager service currently contains 5 elements in a sequenced list such that Element 1 points to Element 2 as its NextUID identifier, while Element 2 points to Element 3 as its NextUID, and so on. Element 1 is considered the head element, and contains a PreviousUID of zero (0). Element 5 is considered the tail element, and contains a NextUID of zero (0). Graphically, the elements can be represented as the ordered sequence shown in Figure 1.

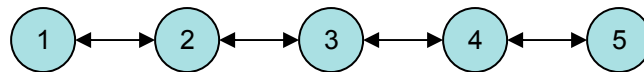


FIGURE 1. EXAMPLE SCENARIO FOR LIST MANIPULATION

To insert an element, the client must send the *SetElement* message with a new Element UID, and the appropriate Previous UID and Next UID to mark the previous (parent) and next (child) elements, respectively. It is the responsibility of the service to update the Next UID of the previous element (if it exists) and the Previous UID of the next element (if it exists) such that a single, contiguous list is maintained. For example, suppose the client sends a *SetElement* message with Element UID=6, Previous UID=2 and Next UID=3. Upon receipt, the List Manager updates the Next UID of the existing Element 2 and the Previous UID of the existing Element 3 such that the new element is inserted between them as shown in Figure 2.

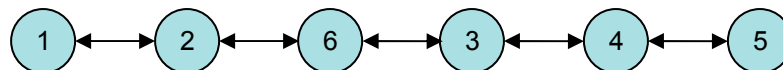


FIGURE 2. EXAMPLE SCENARIO FOR INSERTING A NEW ELEMENT

Similarly, a new head element can be inserted by specifying a Previous UID of zero (0) and a Next UID that points to the existing head. A new tail element can be inserted (appended) by specifying a Previous UID that points to the existing tail and a Next UID of zero (0). Circular or repeated patterns can be created by appending a new tail with a non-zero Next UID. For example, consider the list from 0 where the client sends a new *SetElement* message with Element UID=7, Previous UID=5, and Next UID=2. The resulting list is shown in Figure 3.

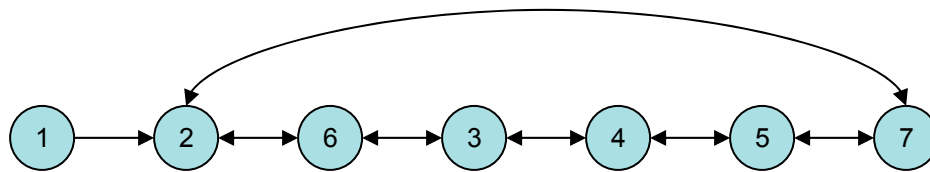


FIGURE 3. EXAMPLE SCENARIO FOR CIRCULAR LIST SEGMENTS

3.4.2 Case 2: Insert Failures

Since the list must contain a single sequence of connected elements, insert requests which orphan existing elements or reference non-existing elements must be rejected by the service. For example, suppose a list manager service currently contains 5 elements in a sequenced list as shown in Figure 1. If the client sends a SetElement message with Element UID=6, Previous UID=3, and Next UID=5, Element 4 would be orphaned as shown in Figure 4. In this case, the list should be left unmodified and the service shall return a RejectElementRequest message.

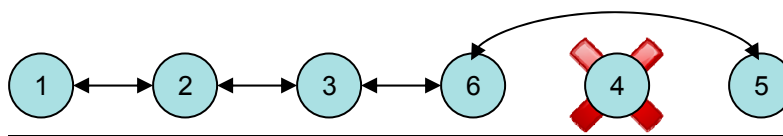


FIGURE 4. FAILURE CASE LEAVING AN EXISTING ELEMENT ORPHANED

To successfully perform such a change to the list, the client should use the DeleteElement message to remove Element 4 before sending the SetElement message to insert Element 6.

3.4.3 Case 3: Deleting from an Existing List

Suppose a list manager service currently contains 5 elements in a sequenced list as shown in Figure 1. To delete an element, the client must send the DeleteElement message with the UID of the element to be deleted. It is the responsibility of the service to update the Next UID of the previous element (if it exists) and the Previous UID of the next element (if it exists) such that a single, contiguous list is maintained. For example, Figure 5 shows the resulting list if Element 2 was deleted.

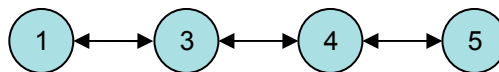


FIGURE 5. EXAMPLE SCENARIO FOR DELETING AN EXISTING ELEMENT

If the specified element does not exist in the list, the service shall leave the list unmodified and return a RejectElementRequest message.

4. UNDERSTANDING PROTOCOL DESCRIPTIONS

The protocol behavior for each component is described in the form of concurrent and hierarchical finite state machines². These finite state machines describe the state of the component and its behavior in response to an input message (external stimuli) or internal event (internal stimuli). The state machines are described diagrammatically as shown in Figure 6 below, and consist of two main elements:

States, described as rectangles with rounded corners, represent the situations during the life of a component. At any given time, a component is said to be in a state that satisfies some condition, performs some activity, or waits for some stimulus.

² For a more detailed description of protocol behavior specification, refer [AS5684A].

Transitions, described by labeled and directed arrows, represent state changes from a single source state to a single destination state.

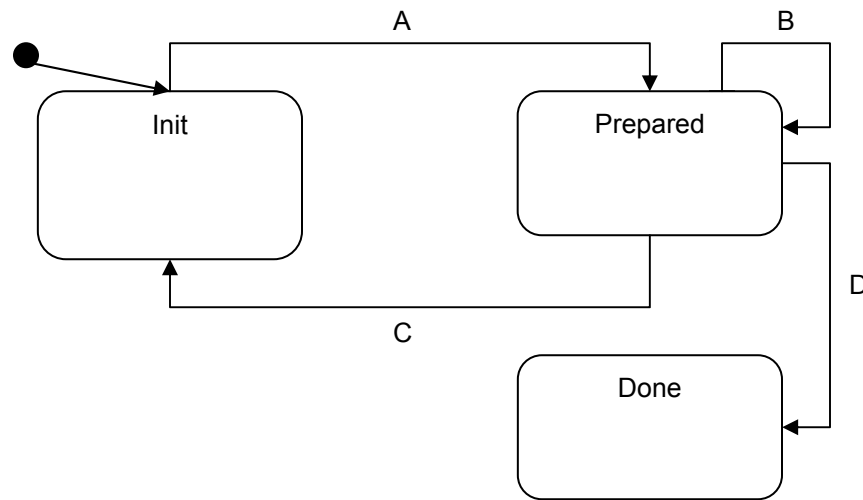


FIGURE 6 - PROTOCOL STATE MACHINE

4.1 States

Each state is given a name by which it can be uniquely identified. A special state called the pseudo-start state is described as a small filled circle. This state contains an arrow that points to the initial state, the state that the component is in when it first comes to life. Pseudo states are also used to point to a nested state that is automatically entered when their parent states are entered.

Each state may correspond to some ongoing activity that is described using a set of actions called entry and exit actions. These actions are performed when a state is entered and when it is exited, and in the order in which they are listed. Their execution is also conditional on the type of the transition that causes the state to be entered or exited. This is explained in the next sub-section on transitions. The entry action usually starts the activity on entry of the state and the exit action often terminates that activity on exit from the state. But in general, the entry and exit actions may not be related and can be used to perform protocol related functions like sending a message, resetting a timeout or performing operations on stored data. If the entry and exist actions perform operations that are local to the component, that is, they do not send out messages, then the actions are represented in italics. Entry and exit are considered to be atomic and non-interruptible. This kind of activity may also be specified in a fine grained manner with the use of nested states, each of which may have their respective entry and exit actions that are executed in the order in which the states are nested.

4.2 Transitions

Transitions cause a state machine to change its state from the source state of the transition to its destination state, and to perform a set of actions called transition actions before transitioning into the destination state. Like entry and exit actions of states, transition actions are also atomic and non-interruptible and represented in italics if they only perform local operations. The signature of a transition consists of an input message or internal event that triggers the transition, an optional guard condition that must evaluate to true for the transition to be executable, and an ordered sequence of zero or more transition actions. These signatures are listed in tabular form as shown in Table 1 to 3 below.

If the transition is triggered by an internal event, then the trigger for that transition is represented in italics. The transition with trigger Failure is an example of a transition that is triggered by an internal event. Since the evaluation of a guard condition is an operation that is always local to the component, all guard conditions are also represented in italics. Transitions are executed when they are found to be executable. Once a state is entered, its transitions are evaluated for executability in the order in which they are defined. If two or more transitions are executable at the same time, the transition that is defined highest in a top-down order among the ones that are executable is the transition that is chosen to be executed.

TABLE 1 - TRANSITION TABLE

Label	Trigger	Condition	Actions
A	RequestToPrepare		sendPrepared
B	SetData	<i>dataWritable</i>	<i>logData</i> , sendSuccess(LOGGED)
	SetData	NOT <i>dataWritable</i>	sendAbort(NOWRITE)
C	RollBack		<i>clearLog</i> , sendSuccess(ROLLEDBACK)
	Commit	NOT <i>isCommit</i>	sendAbort(NOCOMMIT)
	<i>Failure</i>		sendAbort(FAILURE)
D	Commit	<i>isCommit</i>	sendSuccess(COMMITTED)

TABLE 2 - CONDITION TABLE

Condition	Interpretation
<i>dataWritable</i>	True if data can be written
<i>isCommit</i>	True if commit was successful

TABLE 3 - ACTION TABLE

Action	Interpretation
sendPrepared	Send the prepared message
<i>logData</i>	Write the data to a log
sendAbort(code)	Send the abort message with indicated code
<i>clearLog</i>	Empty the log
sendSuccess(code)	Send the success message with indicated code

When a state machine is nested, a transition to the enclosing state (or super state) also represents a transition to a nested state only if a nested pseudo start state is used to point to the nested state. In Figure 7, the transition labeled A has Prepared.Standby as its destination state due to the presence of the nested pseudo start state in the super state called Prepared. Alternately, a transition may be directed specifically to a nested state as in transition B in Figure 7. A nested state may also be a source state to a transition (transition G in Figure 7, for example). A transition that has the enclosing state as the source state is equivalent to a transition that has any of the nested states as its source state.

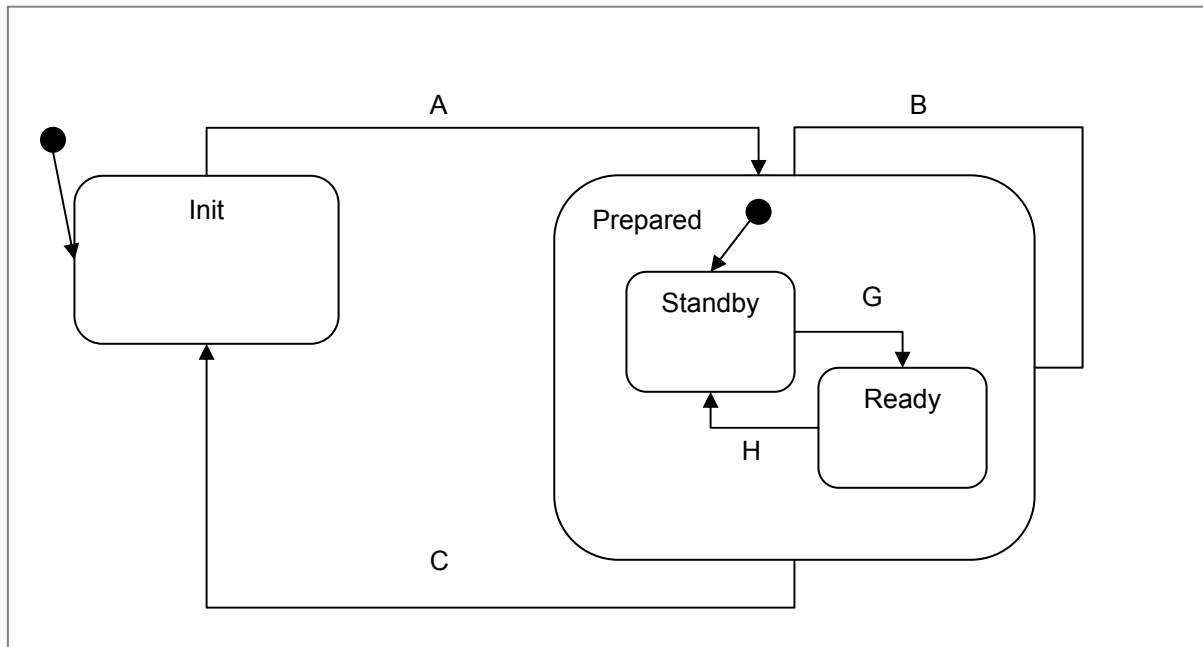


FIGURE 7 - NESTED STATES WITH SIMPLE AND INTERNAL TRANSITIONS

This standard uses four types of transitions: simple transitions, internal transitions, push transitions and pop transitions. As the name suggests, a simple transition changes the state of the finite state machine from the transition's source state to its destination state. An internal transition causes the finite state machine to self-transition to the source state of the transition without exiting and re-entering the source state. That is, the entry and exit actions are not invoked on an internal transition. In Figure 7 above, the transition labeled A is a Simple transition from Init to Prepared, and the transition labeled B represents an internal transition from Prepared to Prepared. Note that the internal transition arrow has no arrow head. A push transition causes the finite state machine to change its state to the destination state of the push transition in a manner that allows a subsequent pop transition to return the state machine back to the source state of the preceding push transition. This operation is similar to the last in first out (LIFO) operation of a stack. Push and Pop transitions are useful for describing subroutine type operations where control has to be returned to the caller of a subroutine that has completed its execution. Push and Pop transitions are described using unfilled circles marked either Push or Pop, as shown in Figure 8 below. In Figure 8, the transitions labeled A and B cause the finite state machine to "push" the Blocked state into a stack, while transition labeled C causes the state machine to pop the blocked state from the stack. For example, the sequence of transitions {A, B, C} results in the blocked state being stacked twice and unstacked once, leaving the state machine in a blocked state that is stacked once. The sequence {A, B, C, C} will return the state machine back to the running state.

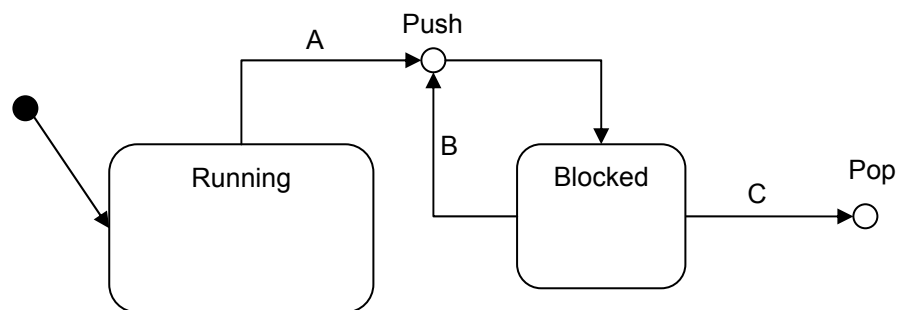


FIGURE 8 - PUSH AND POP TRANSITIONS

The execution of entry and exit actions of the source and destination states of a transition is conditional upon the type of the transition. The table below shows when these actions can and cannot be executed.

TABLE 4 - TYPES OF TRANSITIONS FOR WHICH ENTRY/EXIT ACTIONS ARE EXECUTED

Transition Type	Execute "from" state's Exit Actions?	Execute "to" state's Entry Actions?
Simple Transition	Yes	Yes
Loop back Transition	No	No
Push Transition	No	Yes
Pop Transition	Yes	No

Concurrent state machines are described graphically as separate state machines that are separated by a vertical or horizontal line as shown in Figure 9.

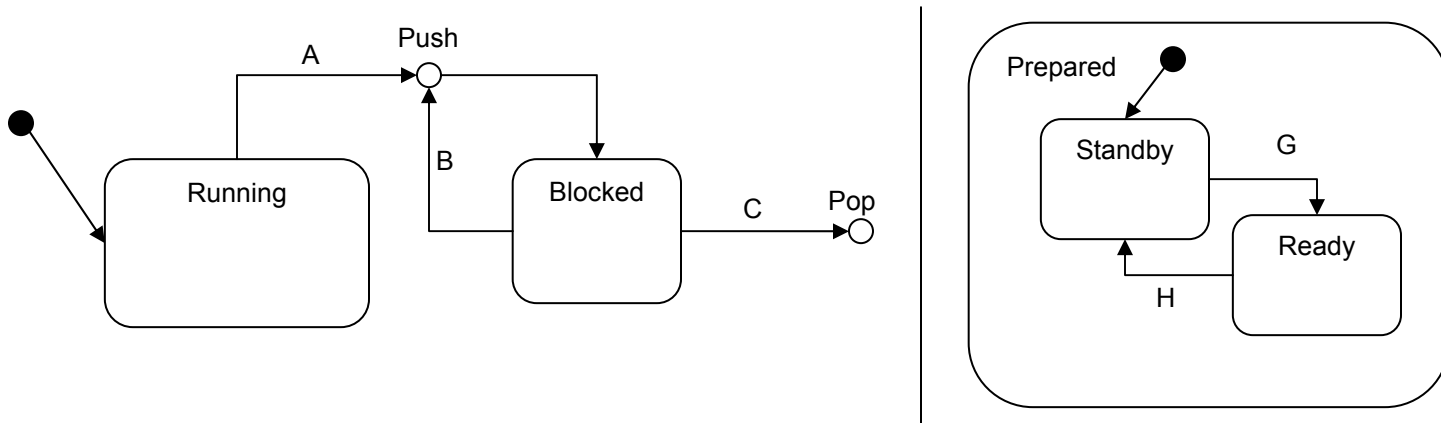


FIGURE 9 - CONCURRENT FINITE STATE MACHINES

A finite state machine can be extended by hierarchically adding to it, or its existing features using the inheritance relationship. This relationship allows only single inheritance and can be used to add new messages, nested states, new transitions and entire concurrent state machines to the service that is being extended. The relationship also allows for the overriding of transitions defined in the service that is being extended with the constraint that the overriding transitions do not alter the behavior of the transitions that are being overridden. Refer to JSIDL for a more detailed description of the inheritance relationship [AS5684A]

5. SERVICE DEFINITIONS

The following subsections provide a textual definition for each Service Definition in the JAUS Core Service Set. Corresponding JSIDL definitions are offered in the Appendix. Additional information on interpreting the service definition elements may be found in [JSIDL].

To facilitate the organization and reuse of services, a system is logically divided into subsystems, nodes, and components. A system contains one or more subsystems. A subsystem contains one or more nodes. A node contains one or more components. A component contains one or more services. Figure 10 depicts a graphical representation of this topology.

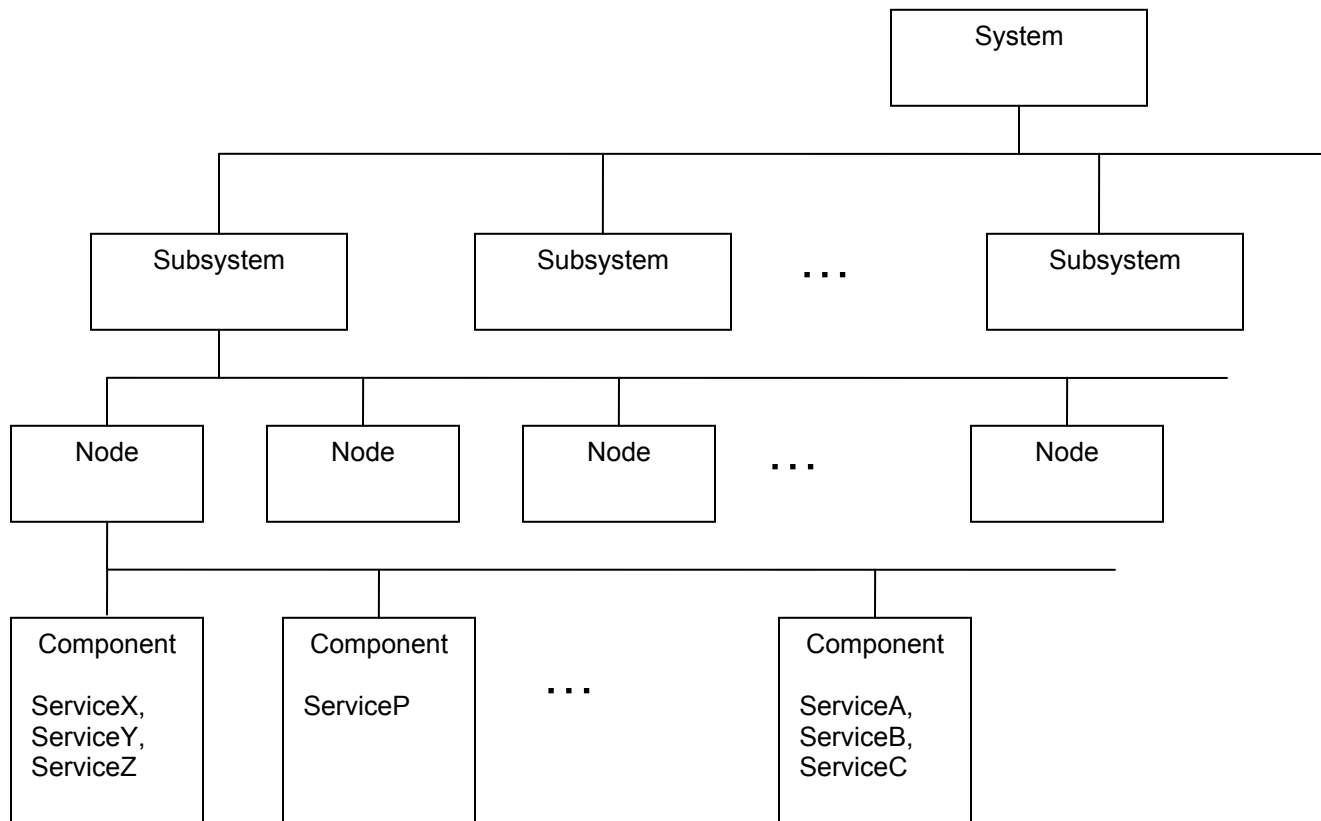


FIGURE 10 - THE SYSTEM TOPOLOGY

Within each component, multiple services may exist only if each service meets the following criteria:

- The service must have a message set that is disjoint from all other services within the same component. For example it is not possible to have two instances of the same service within a single component. If it is necessary to have multiple instances of the same service, each service instance must be in a separate component which can then be logically grouped with other components within a single node.
- The service must not cause an ambiguous state change in another service within the same component. For services that inherit a common service, this limitation means that the inherited service must contain only one state. For example, two services within the same component must not inherit from the AccessControl service (see Section 5.3) since a state transition in one service can cause an ambiguous state change in the other service through the implicit state change in the AccessControl service. Conversely, it is safe for two services to inherit from the Events service (see Section 5.2) since that service contains only one state.

5.1 Transport

name=Transport
version=1.1
id=urn:jaus:jss:core:Transport

5.1.1 Description

The transport service acts as an interface to the JAUS transport layer. It models an abstract bi-directional communication channel (input queue and output queue) whose primary function is to provide the capability of sending messages to a single destination endpoint or broadcasting messages to all endpoints in the system, and to receive a message from any source endpoint. It also provides the capability to prioritize the delivery of sent messages.



Documents

Document Sharing Hub

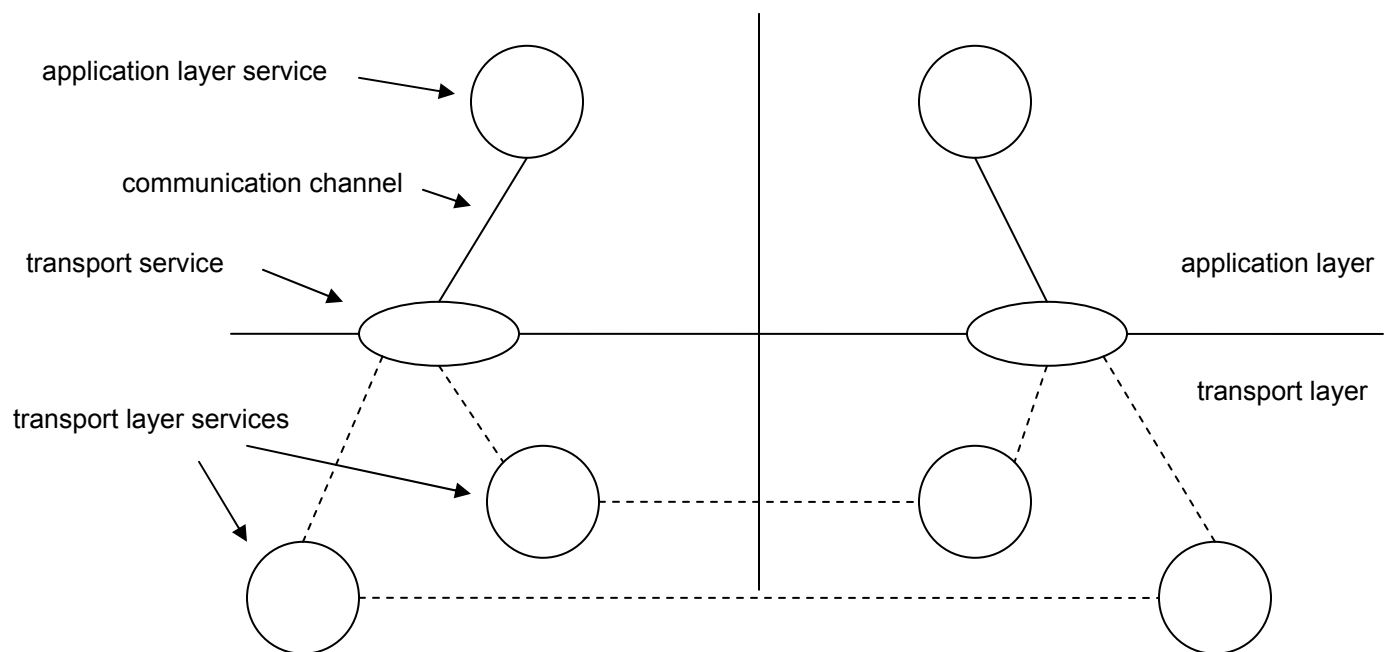


FIGURE 11 - TRANSPORT SERVICE ACTING AS AN INTERFACE TO THE TRANSPORT LAYER

This service establishes a communication endpoint whose address is defined by a triple {SubsystemID, NodeID, ComponentID} as specified by the Send and Receive internal events. Other services that need to utilize the communication channel provided by the transport service must inherit from the transport service.

5.1.2 Assumptions

To ensure proper delivery of messages, this service assumes that the remote transport layer services also implement a similar or compatible version of a transport service.

5.1.3 Vocabulary

The table below lists the vocabulary of the Transport Service. This service has no input and output messages. It contains only internal events that are triggered when a message is sent or received.

TABLE 5 - TRANSPORT SERVICE INTERNAL EVENTS SET

Name	Interpretation
<u>Receive</u>	The Receive event is used by derived services to hand over the data that was sent by another entity to an endpoint established by this service.
<u>Send</u>	The Send event is used by a derived service to hand over the payload data that it needs to send to a specified destination endpoint via the transport layer. Upon receipt, this service prepares the message for delivery (marshals the message) as specified by the transport layer specification [AS5669] and attempts to deliver the encapsulated payload data to the destination endpoint. The second field called ReliableDelivery must be used to specify the method of delivery. Unreliable delivery shall result in the message being sent only once with no guarantee of its arrival. If reliable delivery is selected but the Transport Layer is unable to deliver the message, a SendError is generated.
<u>BroadcastLocal</u>	The Broadcast Local event is the same as the Send event except that this service sends the payload to all endpoints within the subsystem. This is equivalent to a Send event where the destination is given as <local subsystem id>:255:255 where <local subsystem id> is the subsystem identifier of the sender.
<u>BroadcastGlobal</u>	The Broadcast Global event is the same as the Send event except that this service sends the payload to all endpoints on all subsystems. This is equivalent to a Send event where the destination is given 65535:255:255.
<i>SendError</i>	This error is raised when a message that was sent using a reliable method of delivery could not be sent to its destination endpoint after the delivery was attempted for a certain time interval.

5.1.4 Encoding

5.1.4.1 Internal Event Set

TABLE 6 - RECEIVE EVENT ENCODING

body └─ record name=ReceiveRec					
record name= ReceiveRec					
Field #	Name	Type	Units	Optional	Interpretation
1	<bit_field> SourceID	unsigned integer	one ³	false	Bits 0-7: Component ID, range 1...255 where 255 represents all components Bits 8-15: Node ID, range 1...255 where 255 represents all nodes Bits 16 – 31: Subsystem ID, range 1...65535 where 65535 represents all subsystems
2	<variable_length_field> Message Payload	count_field = unsigned long integer	format= J AUS _MESSAGE	false	Enclosed message payload

³ 'one' represents a dimensionless quantity in SI units [bnt]

TABLE 7 - SEND EVENT ENCODING

body record name=SendRec					
record name=SendRec					
Field #	Name	Type	Units	Optional	Interpretation
1	<presence_vector>	unsigned_byte	one ³	false	
2	<fixed_field> ReliableDelivery	unsigned_byte	one ³	false	Enumeration: 0 Unreliable (default) 1 Reliable
3	<bit_field> DestinationID	unsigned integer	one ³	false	Bits 0-7: Component ID, range 1...255 where 255 represents all components Bits 8-15: Node ID, range 1...255 where 255 represents all nodes Bits 16 – 31: Subsystem ID, range 1...65535 where 65535 represents all subsystems
4	<bit_field> SourceID	unsigned integer	one ³	true	Bits 0-7: Component ID, range 1...255 where 255 represents all components Bits 8-15: Node ID, range 1...255 where 255 represents all nodes Bits 16 – 31: Subsystem ID, range 1...65535 where 65535 represents all subsystems
5	<fixed_field> Priority	unsigned byte	one ³	true	Default priority is NORMAL value_enum 0 = LOW 1 = NORMAL 2 = HIGH 3 = SAFETY
6	<variable_length_field> Message Payload	count_field = unsigned long integer	format= JAUS_MESSAGE	false	Enclosed message payload

TABLE 8 - BROADCAST LOCAL EVENT ENCODING

body └─ record name=BroadcastRec					
record name=BroadcastRec					
Field #	Name	Type	Units	Optional	Interpretation
1	<presence_vector>	unsigned_byte	one ³	false	
2	<bit_field> DestinationID	unsigned integer	one ³	false	Bits 0-7: Component ID, range 1...255 where 255 represents all components Bits 8-15: Node ID, range 1...255 where 255 represents all nodes Bits 16 – 31: Subsystem ID, range 1...65535 where 65535 represents all subsystems
3	<bit_field> SourceID	unsigned integer	one ³	true	Bits 0-7: Component ID, range 1...255 where 255 represents all components Bits 8-15: Node ID, range 1...255 where 255 represents all nodes Bits 16 – 31: Subsystem ID, range 1...65535 where 65535 represents all subsystems
4	<fixed_field> Priority	unsigned byte	one ³	true	Default priority is NORMAL value_enum 0 = LOW 1 = NORMAL 2 = HIGH 3 = SAFETY
5	<variable_length_field> Message Payload	count_field = unsigned long integer	format= JAUS_MESSAGE	false	Enclosed message payload

TABLE 9 - BROADCAST GLOBAL EVENT ENCODING

body └─ record name=BroadcastRec
Encoding for the <i>BroadcastGlobal</i> event shall be the same as the <i>BroadcastLocal</i> event.

TABLE 10 – SEND ERROR EVENT ENCODING

body └─ record name=SendRec
Encoding for the <i>SendError</i> event shall be the same as the <i>Send</i> event.

5.1.5 Protocol Behavior

The protocol behavior of the transport service consists of two concurrent state machines. One is used for receiving messages and the other is used for sending messages out. State machines of derived services must inherit from the Receive state machine and override the Receive transition in order to specify the receipt of messages belonging to their vocabularies (see [AccessControl](#) service). Similarly, derived services may overload the transitions of the Send state machine if more specific behavior needs to be specified for those transitions.

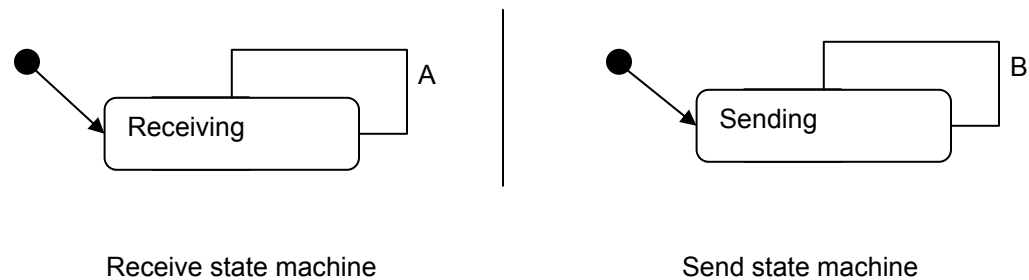


FIGURE 12 - TRANSPORT SERVICE PROTOCOL BEHAVIOR

TABLE 11 - TRANSPORT SERVICE STATE TRANSITION TABLE

Label	Trigger	Conditions	Actions
A	Receive	false	
	SendError	false	
B	Send		Enqueue
	BroadcastLocal		BroadcastLocalEnqueue
	BroadcastGlobal		BroadcastGlobalEnqueue

TABLE 12 - TRANSPORT SERVICE CONDITIONS

Condition	Interpretation
false	The strongest pre-condition that can only be weakened in derived services for transitions containing this guard to be executed.

TABLE 13 - TRANSPORT SERVICE TRANSITION ACTIONS

Action	Interpretation
Enqueue	Convert the destination address into an unsigned integer such that the ComponentID maps to the least significant byte, NodeID to the next least significant byte and SubsystemID maps onto the remaining two bytes of the integer. Package the message as specified by the transport layer specification and send it to its destination as per the specified priority.
BroadcastLocalEnqueue	Package the message as specified by the transport layer specification and send it to all endpoints in the local subsystem.
BroadcastGlobalEnqueue	Package the message as specified by the transport layer specification and send it to all endpoints on all subsystems.

5.2 Events

name=Events
version=1.1
id=urn:jaus:jss:core:Events

inherits-from Transport
name=transport
id= urn:jaus:jss:core:Transport
version=1.1

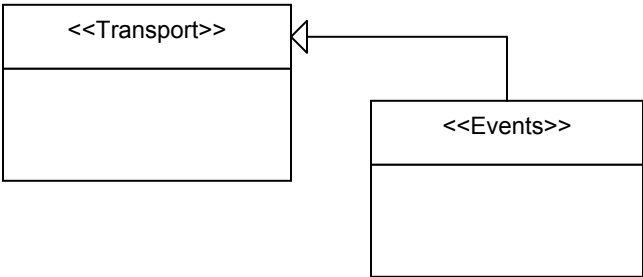


FIGURE 13 - EVENTS SERVICE

5.2.1 Description

This service allows clients to subscribe to changes in the information provided by report messages supported by services that derive from this service. When a client successfully creates an event, an InternalEvent occurs when the event creation criteria are met, and as a result the matching Event is sent to the subscribing client. A recurring event may be terminated by the server if not re-created or updated by the client within a certain *timeout* period. This mechanism may be used to prevent lost events within the system if the client fails to cancel the event properly. Selection of the appropriate length of time for the *timeout* is implementation specific.

All services which inherit from the Events Service shall be capable of generating an event for any supported Query/Report pair of messages.

In version 1.1, the concept of Command Events are introduced. This allows services which derive from the Events Service to establish a mechanism by which a command must be executed within a specified period of time. Failure to complete the command within the duration given results in a failure message.

5.2.2 Assumptions

Messages may be delayed, lost or reordered.

5.2.3 Vocabulary

The table below lists the vocabulary of the Events Service.

TABLE 14 - EVENTS SERVICE VOCABULARY

Message ID (hex)	Name	Command
Input Set		
01F0	CreateEvent	True
01F1	UpdateEvent	True
01F2	CancelEvent	True
01F6	CreateCommandEvent	True
21F0	Query Events	False
21F2	QueryEventTimeout	False
Output Set		
01F3	ConfirmEventRequest	False
01F4	RejectEventRequest	False
41F0	ReportEvents	False
41F1	Event	False
41F2	ReportEventTimeout	False
41F6	CommandEvent	False

TABLE 15 - EVENTS SERVICE INTERNAL EVENTS SET

Name	Interpretation
<i>EventOccured</i>	Must occur when an event occurs.
<i>EventError</i>	Must occur when an event error occurs.
<i>Timeout</i>	Must occur when a recurring event is not re-created or updated periodically
<i>CommandCompleted</i>	Must occur when a command event is completed
<i>CommandExpired</i>	Must occur when a command event does not complete within its specified period of time.

5.2.4 Encoding

5.2.4.1 Internal Event Set

All internal events contain no data fields.

5.2.4.2 Input Set

5.2.4.2.1 ID 01F0h: CreateEvent

This message is used to set up an event. Field 1 is a local request ID that the event provider returns in the Confirm or Reject message. Field 2 is the Event Type, which allows the requester to specify the type of event – Periodic specifies that the event is triggered at regular intervals in time, in which case field 3 (Requested periodic rate) must be set to a non-zero value. Event type of Every Change specifies that the corresponding Report message should be sent every time the data associated with that message changes.

Field 4 contains the size of the Query message that is to specify the contents of the Report. Field 5 contains the Query message (including its two byte header).

TABLE 16 - CREATE EVENT MESSAGE ENCODING

body └─ record name=CreateEventRec					
record name=CreateEventRec					
Field #	Name	Type	Units	Optional	Interpretation
1	<fixed_field> RequestID	unsigned byte	one ³	false	Local Request ID for use in confirm event
2	<fixed_field> EventType	unsigned byte	one ³	false	value_enum 0 Periodic 1 Every change
3	<fixed_field> Requested PeriodicRate	unsigned short integer	hertz	false	Must be specified for periodic event, and set to 0 for every change scale_range real_lower_limit=0 real_upper_limit=1092
4&5	<variable_length_field> QueryMessage	count_field = unsigned integer	one ³	false	The JAUS Query message to be used by the receiving component to generate the report message(s)

5.2.4.2.2 ID 01F1h: UpdateEvent

The Update Event message allows the requester to request a rate or change. The format is the same as the Create Event, only with the addition of Event ID field to specify the event that needs updating.

TABLE 17 - UPDATE EVENT MESSAGE ENCODING

body └─ record name=UpdateEventRec					
record name=UpdateEventRec					
Field #	Name	Type	Units	Optional	Interpretation
1	<fixed_field> RequestID	unsigned byte	one ³	false	Local Request ID for use in confirm event
2	<fixed_field> EventType	unsigned byte	one ³	false	value_enum 0 Periodic 1 Every change
3	<fixed_field> RequestedPeriodic Rate	unsigned short integer	hertz	false	Must be specified for periodic event, and set to 0 for every change scale_range real_lower_limit=0 real_upper_limit=1092
4	Event ID	unsigned byte	one ³	false	Unique identifier of existing event to update
5&6	<variable_length_field> QueryMessage	count_field = unsigned integer	one ³	false	The JAUS Query message to be used by the receiving component to generate the report message(s)

5.2.4.2.3 ID 01F2h: CancelEvent

The Cancel Event message is used by the requester to cancel and/or request deletion of the specified event.

TABLE 18 - CANCEL EVENT MESSAGE ENCODING

body └─ record name=CancelEventRec					
record name=CancelEventRec					
Field #	Name	Type	Units	Optional	Interpretation
2	<fixed_field> RequestID	unsigned byte	one ³	false	Local Request ID for use in confirm/reject message
4	<fixed_field> EventID	unsigned byte	one ³	false	Unique identifier of existing event to be removed

5.2.4.2.4 ID 01F6h: CreateCommandEvent

This message is used to set up a command event. Field 1 is a local request ID that the event provider returns in the Confirm or Reject message. Field 2 is the maximum allowed execution time; any command not completed within its specified duration is considered a failure. Field 3 contains the size of the Command message that is to specify the command to be executed. Field 4 contains the encoded command message (including its two byte header).

TABLE 19 - CREATE EVENT MESSAGE ENCODING

body └─ record name=CreateEventRec					
record name=CreateEventRec					
Field #	Name	Type	Units	Optional	Interpretation
1	<fixed_field> RequestID	unsigned byte	one ³	false	Local Request ID for use in confirm event
2	<fixed_field> MaximumAllowedDuration	unsigned integer	second	false	Any commands not executed within the maximum allowed time are considered a failure.
3&4	<variable_length_field> CommandMessage	count_field = unsigned integer	one ³	false	The JAUS Command message to be used by the receiving component to generate the command event.

5.2.4.2.5 ID 21F0h: QueryEvents

The Query Events message is used to request detail on events. Queries can be made by message ID, event type or Event ID. If no filter is specified, all events should be reported.

TABLE 20 - QUERY EVENTS MESSAGE ENCODING

body └─ variant name=QueryEventsVar (vtag_field = unsigned byte) record name=MessageIDRec record name=EventTypeRec record name=EventIDRec record name=AllEventsRec					
record name=MessageIDRec					
Field #	Name	Type	Units	Optional	Interpretation
1	<fixed_field> MessageID	unsigned short integer	one ³	false	Query Message ID of the Event message that the receiving component is inquiring about.
record name=EventTypeRec					
Field #	Name	Type	Units	Optional	Interpretation
1	<fixed_field> EventType	unsigned byte	one ³	false	value_enum 0 Periodic 1 Every change
record name=EventIDRec					
Field #	Name	Type	Units	Optional	Interpretation
1	<fixed_field> EventID	unsigned byte	one ³	false	Event ID returned by Confirm Event for details on specific event.
record name=AllEventsRec					
Field #	Name	Type	Units	Optional	Interpretation
1	<fixed_field> AllEvents	unsigned byte	one ³	false	All events should be reported. value_range(0,0)

5.2.4.2.6 ID 21F2h: QueryEventTimeout

This message is used by clients of this service to query the timeout period of the service.

TABLE 21 - QUERY EVENT TIMEOUT MESSAGE ENCODING

body
└ (empty)

5.2.4.3 Output Set

5.2.4.3.1 ID 01F3h: ConfirmEventRequest

The Confirm Event message is used to confirm an Event has been created/updated/or cancelled. Field 1 represents the Request ID from the Create, Update, or Cancel message that initiated this message. The Request ID’s scope is local to the requesting client only. Field 2, Event ID, is a globally unique ID that is established by the service when the event is created. Field 3 is used to specify the closest rate that the service can provide if it cannot match the requested rate.

TABLE 22 - CONFIRM EVENT REQUEST MESSAGE ENCODING

body					
└ record name=ConfirmEventRequestRec					
record name=ConfirmEventRequestRec					
Field #	Name	Type	Units	Optional	Interpretation
1	<fixed_field> RequestID	unsigned byte	one ³	false	ID of the event maintenance request (Create, Update, or Cancel)
2	<fixed_field> EventID	unsigned byte	one ³	false	The identifier of the specific event
3	<fixed_field> ConfirmedPeriodic Rate	unsigned short integer	hertz	false	Requested rate or closest to requested rate scale_range lower_limit=0 upper_limit=1092

5.2.4.3.2

ID 01F4h: RejectEventRequest

The Reject Event Request message is used to reject an Event creation, update or cancellation. Field 2 represents the Request ID from the Create, Update, or Cancel message that initiated this message. The Request ID's scope is local to the requesting client only.

TABLE 23 - REJECT EVENT REQUEST MESSAGE ENCODING

body record name=RejectEventRequestRec					
record name=RejectEventRequestRec					
Field #	Name	Type	Units	Optional	Interpretation
1	<presence_vector>	unsigned byte	one ³	false	
2	<fixed_field> RequestID	unsigned byte	one ³	false	ID of the event maintenance request (create, update, or cancel)
3	<fixed_field> ResponseCode	unsigned byte	one ³	false	value_enum 1 periodic events not supported 2 change based events not supported 3 connection refused 4 invalid event setup 5 message not supported 6 error, invalid event ID for update event request
4	<fixed_length_string> ErrorMessage	String of size 80 bytes	one ³	true	String for additional information

5.2.4.3.3 ID 41F0h: ReportEvents

This message is used to report the active event requests that match the requirements provided in the QueryEvents message.

TABLE 24 - REPORT EVENTS MESSAGE ENCODING

body list (count_field = unsigned byte) record name=ReportEventsRec					
record name=ReportEventsRec					
Field #	Name	Type	Units	Optional	Interpretation
1	<fixed_field> EventType	unsigned byte	one ³	false	value_enum 0 Periodic 1 Every change
2	<fixed_field> EventID	unsigned byte	one ³	false	Unique Identifier of existing event to update
3&4	<variable_length_field> QueryMessage	count_field = unsigned integer	one ³	false	The JAUS Query message to be used by the receiving component to generate the report message(s)

5.2.4.3.4 ID 41F1h: Event

The Event message is sent when an event is triggered. It includes the Event ID and a sequence number to allow the client to keep track of event processing.

TABLE 25 - EVENT MESSAGE ENCODING

body └─ record name=EventRec					
record name=EventRec					
Field #	Name	Type	Units	Optional	Interpretation
1	<fixed_field> Event ID	unsigned byte	N/A	False	Unique identifier of the enclosed event
2	<fixed_field> Sequence Number	unsigned byte	N/A	False	Sequential count of the number of times this event has been issued
3&4	<variable_length_field> ReportMessage	count_field = unsigned integer	N/A	False	The J AUS Report message including the message header

5.2.4.3.5 ID 41F2h: ReportEventTimeout

This message is used to report the timeout period of the service.

TABLE 26 - REPORT EVENT TIMEOUT MESSAGE ENCODING

body └─ record name=ReportEventTimeoutRec					
record name=ReportEventTimeoutRec					
Field #	Name	Type	Units	Optional	Interpretation
1	<fixed_field> Timeout	Unsigned byte	Minutes	False	Clients must re-create or update an event to prevent being rejected when the timeout expires. A value of zero indicates this feature is disabled.

5.2.4.3.6 ID 41F6h: CommandEvent

The CommandEvent message is sent when a command specified by a previous Create Command Event message has completed or expired. It includes the Event ID and command result.

TABLE 27 – COMMAND EVENT MESSAGE ENCODING

body └─ record name=CommandEventRec					
record name=CommandEventRec					
Field #	Name	Type	Units	Optional	Interpretation
1	<fixed_field> Event ID	unsigned byte	one ³	False	Unique identifier of the enclosed event
2	<fixed_field> CommandResult	Unsigned byte	one ³	False	Enumeration 0 SUCCESSFUL 1 UNSUCCESSFUL

5.2.5 Protocol Behavior

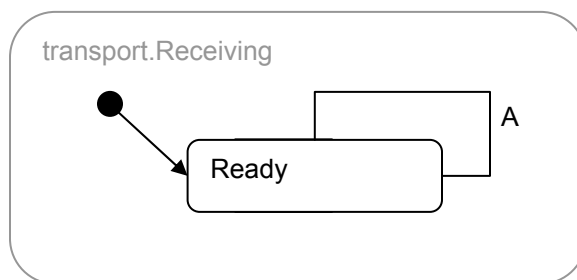


FIGURE 14 - EVENTS SERVICE PROTOCOL BEHAVIOR

TABLE 28 - EVENTS SERVICE STATE TRANSITION TABLE

Label	Trigger	Conditions	Actions
A	Query Events		sendReportEvents
	Query Event Timeout		sendReportEventTimeout
	Create Event	<i>isSupported</i> AND NOT <i>eventExists</i>	<i>createEvent</i> , sendConfirmEventRequest <i>resetEventTimer</i>
	Create Event	<i>isSupported</i> AND <i>eventExists</i>	<i>updateEvent</i> , sendConfirmEventRequest <i>resetEventTimer</i>
	Create Event	NOT <i>isSupported</i>	sendRejectEventRequest
	Create Command Event	<i>isCommandSupported</i> AND NOT <i>eventExists</i>	<i>createCommandEvent</i> , sendConfirmEventRequest, startCommandTimer
	Create Command Event	<i>isCommandSupported</i> AND <i>eventExists</i>	<i>updateCommandEvent</i> , sendConfirmEventRequest, startCommandTimer
	Create Command Event	NOT <i>isCommandSupported</i>	sendRejectEventRequest
	Update Event	<i>isSupported</i> AND <i>eventExists</i>	<i>updateEvent</i> , sendConfirmEventRequest <i>resetEventTimer</i>
	Update Event	NOT <i>isSupported</i> OR NOT <i>eventExists</i>	sendRejectEventRequest
	Cancel Event	<i>eventExists</i>	<i>cancelEvent</i> , sendConfirmEventRequest <i>stopEventTimer</i>
	Cancel Event	NOT <i>eventExists</i>	sendRejectEventRequest
	<i>Event Occurred</i>	<i>eventExists</i>	sendEvent
	<i>Event Error</i>	<i>eventExists</i>	sendRejectEventRequest
	<i>Timeout</i>		<i>cancelEvent</i> , sendConfirmEventRequest <i>stopEventTimer</i>
	<i>Command Completed</i>	<i>eventExists</i>	stopCommandTimer, sendCommandEvent(SUCCESSFUL)
	<i>Command Expired</i>		sendCommandEvent(UNSUCCESSFUL)

TABLE 29 - EVENTS SERVICE CONDITIONS

Condition	Interpretation
<i>isSupported</i>	True if parameters are supported. All messages that belong to services that derive from this service must be supported.
<i>eventExists</i>	True if the specified event exists.
<i>isCommandSupported</i>	False. This condition should be overridden by derived services to allow command events.

TABLE 30 - EVENTS SERVICE TRANSITION ACTIONS

Action	Interpretation
sendReportEvents	Send Report Events message to the component that sent the query
<i>createEvent</i>	Create the event
<i>createCommandEvent</i>	Add the encapsulated command message to the input message queue of this service so that it can be received and processed by the extensions of this service (derived services)
sendConfirmEventRequest	Send Confirm Event Request message
sendRejectEventRequest	Send Reject Event Request message
<i>updateEvent</i>	Update the event
<i>updateCommandEvent</i>	Update the command event
<i>cancelEvent</i>	Cancel the event
sendEvent	Send an Event notification
<i>resetEventTimer</i>	Reset the timeout period for the recurring event.
<i>stopEventTimer</i>	Stop evaluating the recurring event for timeouts.
sendCommandEvent	Send a CommandEvent notification
startCommandTimer	Start a timer for the specified CommandEvent
stopCommandTimer	Stop the timer for the specified CommandEvent

5.3 AccessControl

name=AccessControl

version=1.1

id= urn:jaus:jss:core:AccessControl

Inherits-from Events

name=events

id= urn:jaus:jss:core:Events

version=1.1

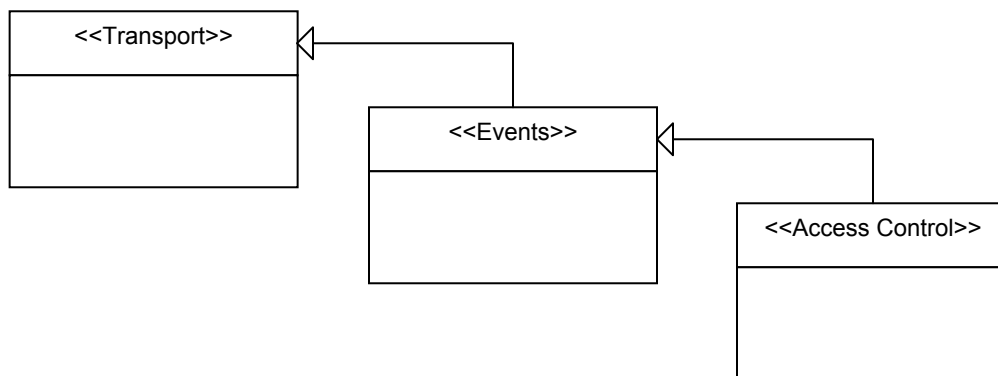


FIGURE 15 - ACCESS CONTROL SERVICE

5.3.1 Description

The Access Control service offers a basic interface for acquiring preemptable exclusive control to one or more related services that utilize this function. Once the exclusive control is established, the related services shall only execute commands originating from the controlling component. The authority code parameter of this service is used for preemption and is to be set equal to that of its controlling client. This service always grants control to the highest authority client that is requesting exclusive control. Commands from all other clients are ignored unless from a client with higher authority.

This service maintains two values, a default value and a current value of a field called authority code. The default value is the value that the service is pre-configured with. Access is provided to clients based on the value of their authority code in comparison to the current value of this service.

State transitions between the Available and NotAvailable nested states are behaviors that are deferred to service definitions that derive from this service.

5.3.2 Assumptions

Messages may be delayed, lost or reordered.

5.3.3 Vocabulary

The table below lists the vocabulary of the Access Control Service.

TABLE 31 - ACCESS CONTROL SERVICE VOCABULARY

Message ID (hex)	Name	Command
Input Set		
000D	RequestControl	True
000E	ReleaseControl	True
200D	QueryControl	False
2001	QueryAuthority	False
0001	SetAuthority (Deprecated)	True
2003	QueryTimeout	False
Output Set		
400D	ReportControl	False
0010	RejectControl	False
000F	ConfirmControl	False
4001	ReportAuthority	False
4003	ReportTimeout	False

TABLE 32 - ACCESS CONTROL SERVICE INTERNAL EVENTS SET

Name	Interpretation
<i>Timeout</i>	Occurs when access is not re-acquired periodically

5.3.4 Encoding

5.3.4.1 Internal Event Set

All internal events contain no data fields.

5.3.4.2 Input Set

5.3.4.2.1 ID 000Dh: RequestControl

This message is used to request interruptible control of the receiving component. Once control is established, the receiving component shall only execute commands from the sending component. The authority code parameter is to be set equal to that of the sending component. The receiving component must always accept the control of the highest authority component that is requesting uninterruptible control. Commands from all other components are ignored unless from a component with higher authority.

TABLE 33 - REQUEST CONTROL MESSAGE ENCODING

body └─ record name=RequestControlRec					
record name=RequestControlRec					
Field #	Name	Type	Units	Optional	Interpretation
1	<fixed_field> AuthorityCode	unsigned byte	one ³	false	range: 0-255

5.3.4.2.2 ID 000Eh: ReleaseControl

This message is used to relinquish uninterruptible control of the receiving component.

TABLE 34 - RELEASE CONTROL MESSAGE ENCODING

body └─ (empty)

5.3.4.2.3 ID 200Dh: QueryControl

This message is used by clients to query the current control state of this service.

TABLE 35 - QUERY CONTROL MESSAGE ENCODING

body └─ (empty)

5.3.4.2.4 ID 2001h: QueryAuthority

This message is used by clients to query the current value of the authority code of this service.

TABLE 36 - QUERY AUTHORITY MESSAGE ENCODING

body └─ (empty)

5.3.4.2.5 ID 0001h: SetAuthority (Deprecated)

This message is deprecated and scheduled to be removed during the next revision.

This message shall set the command authority of the receiving component. The authority bits range in value from 0 to 255 with 255 being the highest.

TABLE 37 - SET AUTHORITY MESSAGE ENCODING

body └─ record name=SetAuthorityRec					
record name=SetAuthorityRec					
Field #	Name	Type	Units	Optional	Interpretation
1	<fixed_field> AuthorityCode	unsigned byte	one ³	false	range: 0-255

5.3.4.2.6 ID 2003h: QueryTimeout

This message is used by clients of this service to query the timeout period of the service.

TABLE 38 - QUERY TIMEOUT MESSAGE ENCODING

body └─ (empty)					
--------------------	--	--	--	--	--

5.3.4.3 Output Set

5.3.4.3.1 ID 400Dh: ReportControl

This message is used to report the current state of control of this service. If the service is in the Controlled state, this message reports the ID of the controlling component. The ID fields shall be set to zero (0) if this service is in the NotControlled state..

TABLE 39 - REPORT CONTROL MESSAGE ENCODING

body └─ record name=ReportControlRec					
record name=ReportControlRec					
Field #	Name	Type	Units	Optional	Interpretation
1	<fixed_field> SubsystemID	unsigned short integer	one ³	false	value_range (0,65535)
2	<fixed_field> NodeID	unsigned byte	one ³	false	value_range (0,254)
3	<fixed_field> ComponentID	unsigned byte	one ³	false	value_range (0,254)
4	<fixed_field> AuthorityCode	unsigned byte	one ³	false	value_range (0,255)

5.3.4.3.2 ID 0010h: RejectControl

The Reject Control message is used to notify a component that control has been released (response code = 0), or a request to release control could not be processed (response code = 1).

TABLE 40 - REJECT CONTROL MESSAGE ENCODING

body └─ record name=RejectControlRec					
record name=RejectControlRec					
Field #	Name	Type	Units	Optional	Interpretation
1	<fixed_field> ResponseCode	unsigned byte	one ³	false	value_enum 0 CONTROL RELEASED value_enum 1 NOT AVAILABLE

5.3.4.3.3 ID 000Fh: ConfirmControl

The Confirm Control message is used to notify a component that it accepts control from that component (or not). When control has been granted, response code of 0, the component under control will only execute messages from the controlling component until control is released or interrupted. When the requesting component has lower authority than the current controlling entity, the response will be 2. For components not supporting interruptible control, or if the component is engaged in other operations that would prevent this service from performing its actions to grant control, the response code value of 1 can be used.

TABLE 41 - CONFIRM CONTROL MESSAGE ENCODING

body └─ record name=ConfirmControlRec					
record name=ConfirmControlRec					
Field #	Name	Type	Units	Optional	Interpretation
1	<fixed_field> ResponseCode	unsigned byte	one ³	false	value_enum 0 CONTROL ACCEPTED value_enum 1 NOT AVAILABLE value_enum 2 INSUFFICIENT AUTHORITY

5.3.4.3.4 ID 4001h: ReportAuthority

This message is used to report the current command authority and associated time-out value.

TABLE 42 - REPORT AUTHORITY MESSAGE ENCODING

body └─ record name=AuthorityRec					
record name=AuthorityRec					
Field #	Name	Type	Units	Optional	Interpretation
1	<declared_field> AuthorityCode	Unsigned byte	one ³	false	

5.3.4.3.5 ID 4003h: ReportTimeout

This message is used to report the timeout period of the service.

TABLE 43 - REPORT TIMEOUT MESSAGE ENCODING

body └─ record name=ReportTimeoutRec					
record name=ReportTimeoutRec					
Field #	Name	Type	Units	Optional	Interpretation
1	<fixed_field> Timeout	Unsigned byte	Seconds	False	Clients must re-request control to prevent being rejected when the timeout expires. A value of zero indicates this feature is disabled.

5.3.5 Protocol Behavior

This service consists of an 'availability' feature by which the access control behavior can be defined for the case in which this service or other related services are engaged in operations that would prevent this service from performing its actions.

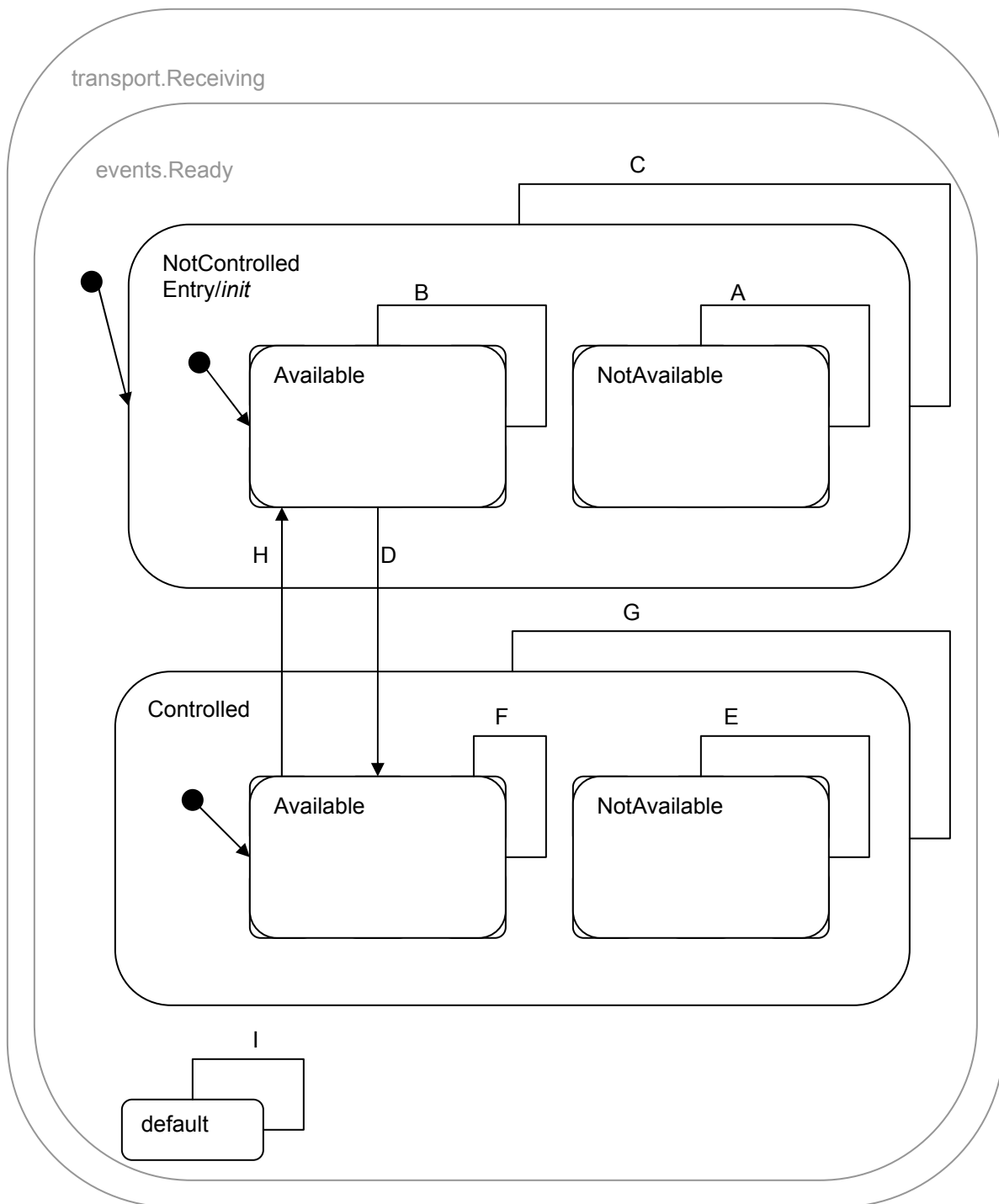


FIGURE 16 - ACCESS CONTROL SERVICE PROTOCOL BEHAVIOR

TABLE 44 - ACCESS CONTROL SERVICE STATE TRANSITIONS

Label	Trigger	Conditions	Actions
A	RequestControl		sendConfirmControl(NOT_AVAILABLE)
B	RequestControl	<i>isDefaultAuthorityGreater</i>	sendConfirmControl(INSUFFICIENT_AUTHORITY)
C	ReleaseControl		sendRejectControl (CONTROL_RELEASED)
D	RequestControl	NOT <i>isDefaultAuthorityGreater</i>	<i>storeAddress</i> , <i>setAuthority</i> , <i>resetTimer</i> , sendConfirmControl(CONTROL_ACCEPTED)
E	RequestControl		sendConfirmControl(NOT_AVAILABLE)
	ReleaseControl		sendRejectControl(NOT_AVAILABLE)
	<i>Timeout</i>		<i>resetTimer</i>
F	RequestControl	<i>isCurrentAuthorityLess</i> AND NOT <i>isControllingClient</i>	sendRejectControlToController(CONTROL_RELEASED) , <i>storeAddress</i> , <i>setAuthority</i> , <i>resetTimer</i> , sendConfirmControl(CONTROL_ACCEPTED)
	RequestControl	NOT <i>isCurrentAuthorityLess</i> AND NOT <i>isControllingClient</i>	sendConfirmControl(INSUFFICIENT_AUTHORITY)
	RequestControl	NOT <i>isDefaultAuthorityGreater</i> AND <i>isControllingClient</i>	<i>resetTimer</i> , <i>setAuthority</i> , sendConfirmControl(CONTROL_ACCEPTED)
G	SetAuthority	<i>isControllingClient</i> AND <i>isAuthorityValid</i>	<i>setAuthority</i>
H	ReleaseControl	<i>isControllingClient</i>	sendRejectControl(CONTROL_RELEASED)
	RequestControl	<i>isDefaultAuthorityGreater</i> AND <i>isControllingClient</i>	sendRejectControlToController(CONTROL_RELEASED)
	<i>Timeout</i>		sendRejectControlToController(CONTROL_RELEASED)
I	QueryAuthority		sendReportAuthority
	QueryTimeout		sendReportTimeout
	QueryControl		sendReportControl