IEEE Standard for a Smart Transducer Interface for Sensors and Actuators—Network Capable Application Processor (NCAP) Information Model

Sponsor

TC-9 Committee on Sensor Technology
of the
IEEE Instrumentation and Measurement Society

Approved 26 June 1999

IEEE-SA Standards Board

Abstract: This standard defines an object model with a network-neutral interface for connecting processors to communication networks, sensors, and actuators. The object model containing blocks, services, and components specifies interactions with sensors and actuators and forms the basis for implementing application code executing in the processor.
Keywords: actuators, communication network, object model, sensors

---

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

Copyright © 2000 by the Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 18 April 2000. Printed in the United States of America.

Print: ISBN 0-7381-1767-6 SH94767
PDF: ISBN 0-7381-1768-4 SS94767

No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE-SA Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
USA

Note: Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying patents for which a license may be required by an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

IEEE is the sole entity that may authorize the use of certification marks, trademarks, or other designations to indicate compliance with the materials set forth herein.

Authorization to photocopy portions of any individual standard for internal or personal use is granted by the Institute of Electrical and Electronics Engineers, Inc., provided that the appropriate fee is paid to Copyright Clearance Center. To arrange for payment of licensing fee, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; (978) 750-8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Introduction

[This introduction is not part of IEEE Std 1451.1-1999, IEEE Standard for a Smart Transducer Interface for Sensors and Actuators—Network Capable Application Processor (NCAP) Information Model.]

The objective of the IEEE/NIST Working Group on transducer interface standards is to utilize existing control networking technology and develop standardized connection methods for Smart Transducers to control networks. Little or no changes would be required to use different methods of analog-to-digital (A/D) conversion, different microprocessors, or different network protocols and transceivers.

This objective is achieved through the definition of a common object model for the components of a Networked Smart Transducer, together with interface specifications to these components.

The Networked Smart Transducer model shows two key views of a smart transducer:

— Physical view
— Logical view

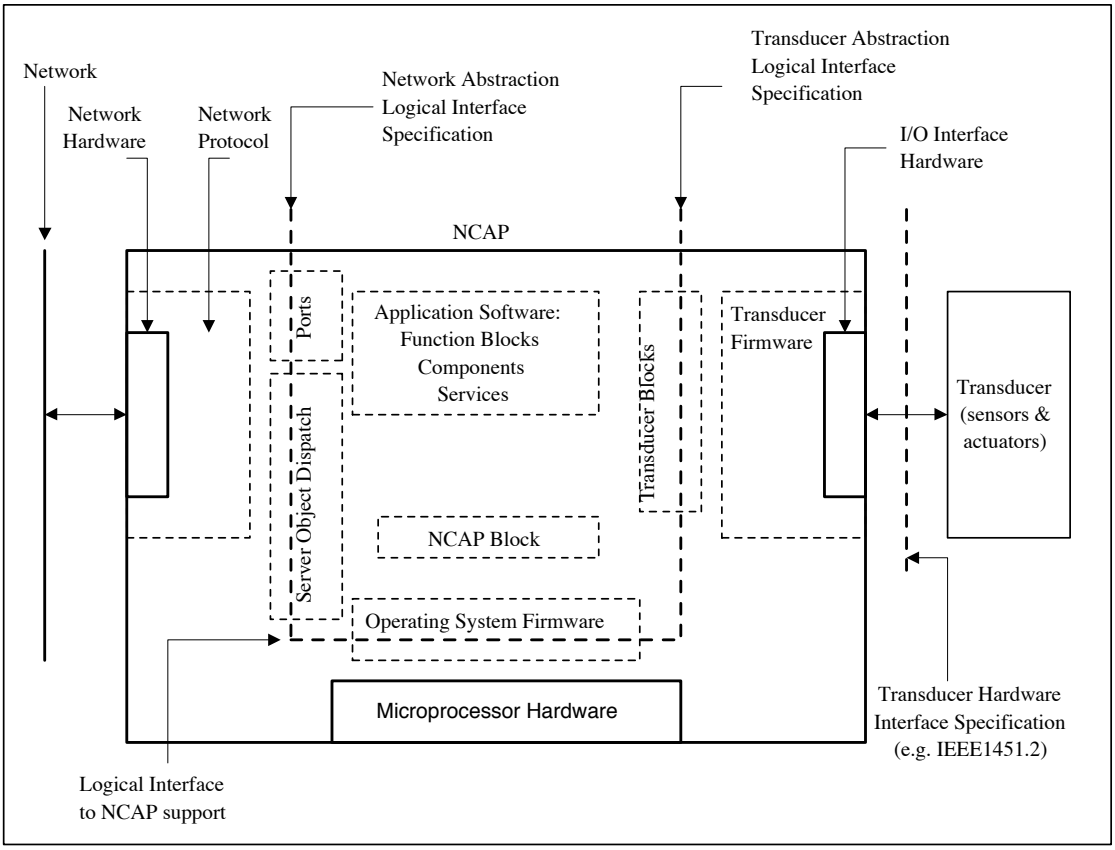

Figure 1—Networked Smart Transducer model

Physical view

The first view shows the physical components of the system. This view is indicated by components drawn in solid lines in Figure 1.

Copyright © 2000 IEEE. All rights reserved.

Figure 1 shows a model composed of sensors and actuators connected to form a transducer. The transducer is connected over an interface to a microprocessor or controller that is in turn interfaced to the network. The Hardware Interface Specification between the sensor/actuator and the rest of the device hardware, known as the network capable application processor (NCAP), is indicated by the rightmost thick, dashed line in the figure. A typical specification is described in the companion standard [IEEE Std 1451.2-1997].

The NCAP hardware consists of the microprocessor and its supporting circuitry as well as hardware implementing the physical layer of the attached network and the input/output (I/O) interface to the transducer, as shown in Figure 1.

Logical view

The second view is the logical view of the system and is indicated by components shown in dotted lines in Figure 1.

The logical components may be grouped into application and support components. The support components are the operating system, the network protocol, and transducer firmware components shown. The operating system provides an interface to applications, indicated by the dashed line labeled “Logical Interface to NCAP support.”

A second logical interface, labeled “Network Abstraction Logical Interface Specification,” consists of Port and Server Object Dispatch components defined in this standard. This interface provides an abstraction to hide communication details specific to a given network within a small set of communication methods. The details of this interface are defined by this standard.

The third logical interface, labeled “Transducer Abstraction Logical Interface Specification” performs the same abstraction function for the specifics of the transducer I/O hardware and firmware. In effect, this interface makes all such transducer interfaces look like I/O drivers. The details of this interface are defined by this standard.

Applications are modeled as Function Blocks in combination with Components and Services. The NCAP block provides application organization and support for the other blocks. All of these Blocks, Components, and Services are defined by this standard.

These interfaces are optional in the sense that not all must be exposed in an implementation.

NOTE—If support for interoperable transducers is not required, it is permissible to not use the IEEE1451.2 Interface Specification, or a similar transducer interface standard, but to still use the IEEE1451.1 object model. Similarly, if networking is not supported, or if the networking implementation is closed, it is not necessary to use IEEE1451.1 to still get the benefits of using IEEE1451.2 or a similar transducer interface standard.

Participants

At the time this draft standard was completed, the Information Model Working Group had the following membership:

Jay Warrior, *Chair*
David Bennett, *Secretary*
John C. Eidson, *Editor*

Dale Blackwell	Alec Dara-Abrams	David Wilson
Harold Brown	Rick Schneeman	George Woodmansee

Copyright © 2000 IEEE. All rights reserved.

The following members of the balloting committee voted on this standard:

Dale Blackwell
Richard Bortins
Mark A. Buckner
Wayne L. Catlin
Rodney Cummings
John Eidson
Randy Frank
Patrick S. Gonia
Martt Harding
Mike Hogan
David Howarth
Elin Klaseen
Kang Lee
Jacek Maitan
Charles McParland
Tremont Miao
John Michaloski
Ashley Molloy
Gerardo Pardo-Castellote
Mark I. Schiefer
David Tetzlaff
Jay Warrior
James Xu

The following persons contributed to earlier versions of this standard:

Larry D. Bashaw
Mike Clark
Edwin El-Kareh
Jill Fay
Mike Geipel
Kang B. Lee
Jacek Maitan
Hans Sitte
Bob Woolever
R. Wesley Wysor

When the IEEE-SA Standards Board approved this standard on 26 June 1999, it had the following membership:

Richard J. Holleman, *Chair* **Donald N. Heirman,** *Vice Chair*

Judith Gorman, *Secretary*

Satish K. Aggarwal
Dennis Bodson
Mark D. Bowman
James T. Carlo
Gary R. Engmann
Harold E. Epstein
Jay Forster*
Ruben D. Garzon
James H. Gurney
Lowell G. Johnson
Robert J. Kennelly
E. G. "Al" Kiener
Joseph L. Koepfinger*
L. Bruce McClung
Daleep C. Mohla
Robert F. Munzner
Louis-François Pau
Ronald C. Petersen
Gerald H. Peterson
John B. Posey
Gary S. Robinson
Akio Tojo
Hans E. Weinrich
Donald W. Zipse

*Member Emeritus

Also included is the following nonvoting IEEE-SA Standards Board liaison:

Robert E. Hebner

Jennifer McClain Longman
IEEE Standards Project Editor

CANopen is a registered trademark of CAN in Automation (CiA), Am Weichselgarten 26, D-91058 Erlangen, Germany.

CONTROLNET is a registered trademark of Allison Systems, Inc., W226 N781 Eastmound Dr., Waukesha, WI 53186.

DEVICENET is a registered trademark of Iris Technologies, Inc., Westmoreland Industrial Park R.R. 12–Box 36, Greensburg, PA 15601.

FOUNDATION fieldbus is a registered trademark of the Fieldbus Foundation, Suite 100, 9390 Research Blvd., Austin, TX 78759.

LonTalk is a registered trademark of Echelon Corporation, 4015 Miranda Avenue, Palo Alto, CA 94041.

Profibus is a registered trademark of Profibus International, Haid-und-Neu Strasse 7, D-76131 Karlsruhe, Germany.

Copyright © 2000 IEEE. All rights reserved.

Contents

1. Overview 1

1.1 Scope 2
1.2 Purpose 2
1.3 Benefits 2

2. References 3

2.1 General references 3
2.2 References pertaining to String representation 3
2.3 References pertaining to uncertainty 5

3. Definitions 5

4. Conventions 14

4.1 Class naming 14
4.2 Descriptive syntax 14
4.3 Word usage 15
4.4 Class definitions 16
4.5 Operation signature datatypes 17
4.6 Behavioral specification notation 18

5. Information model 19

5.1 Object model specifications 20
5.2 Data model specifications 27
5.3 Network communication models 28
5.4 IEEE 1451.1 users 32

6. Datatypes in an IEEE 1451.1 system 33

6.1 Primitive datatypes 33
6.2 Derived datatypes 37

7. Common properties 51

7.1 Class designator properties 51
7.2 Object operation properties 53
7.3 Block Cookie properties 66
7.4 Value uncertainty specification 68

8. Top-level class definitions 69

8.1 Root abstract class 69
8.2 Entity abstract class 70

9. Block classes 81

9.1 Block abstract class 81
9.2 NCAP Block class 90

Copyright © 2000 IEEE. All rights reserved.

9.3 Function Block class 108
9.4 Base Transducer Block abstract class 111
9.5 Transducer Block abstract class 114
9.6 Dot2 Transducer Block class 123
9.7 Dot3 Transducer Block class 123
9.8 Dot4 Transducer Block class 124

10. Component classes 124

10.1 Component abstract class 124
10.2 Parameter class 126
10.3 Parameter With Update class 128
10.4 Physical Parameter abstract class 133
10.5 Scalar Parameter class 147
10.6 Scalar Series Parameter class 149
10.7 Vector Parameter class 154
10.8 Vector Series Parameter class 159
10.9 Time Parameter class 164
10.10 Action class 168
10.11 File class 171
10.12 Partitioned File class 177
10.13 Component Group class 181

11. Service classes 187

11.1 Service abstract class 187
11.2 Base Port abstract class 189
11.3 Base Client Port abstract class 191
11.4 Client Port class 193
11.5 Asynchronous Client Port class 198
11.6 Base Publisher Port abstract class 208
11.7 Publisher Port class 211
11.8 Self Identifying Publisher Port class 213
11.9 Event Generator Publisher Port class 219
11.10 Subscriber Port class 225
11.11 Mutex Service class 229
11.12 Condition Variable Service class 233

12. Publication semantics 239

12.1 Overview 239
12.2 Publishing operational behavior 240
12.3 Structure of publications 241

13. Defined publications 246

13.1 Publication Timer_Tick specification 246
13.2 Publication Timer_Properties specification 247
13.3 Publication Request_Timer_Properties specification 249
13.4 Publication Physical_Parametric_Data specification 249
13.5 Publication Physical_Parametric_ Data_Publisher_Identifying specification 252

14. Encode and decode rules 253

Copyright © 2000 IEEE. All rights reserved.

14.1 Argument encoding and decoding 253
14.2 Client-server argument encoding and decoding 254
14.3 Publish-side argument encoding 256
14.4 Subscriber-side argument decoding 260

15. Memory management rules 263

15.1 Applicability of memory management ruler 263
15.2 In process memory management 263
15.3 Simple datatype memory management 263
15.4 Nonsimple datatype memory management 263

16. Conformance 266

16.1 Conformance objective 266
16.2 Definition of conformance terms 267
16.3 Conformance requirements 269

Annex A (informative) Using the object model 275

Annex B (informative) Client-server example 284

Annex C (informative) Publish-subscribe example 292

Annex D (informative) System configuration and operation examples 298

Annex E (informative) NCAP interoperation and network-independent code 310

Annex F (normative) IEEE 1451.1 String character set 312

Annex G (normative) Assignment of enumeration values to ISO 639 language codes 315

Annex H (normative) IEEE1451.2 Transducer Block definition 320

Annex I (informative) Bibliography 341

Copyright © 2000 IEEE. All rights reserved.

IEEE Standard for a Smart Transducer Interface for Sensors and Actuators—Network Capable Application Processor (NCAP) Information Model

1. Overview

This standard is divided into 16 clauses:

Clause	Purpose
1.	Provides the scope, purpose, and benefits of this standard
2.	Lists references to other standards that are referenced by this standard
3.	Provides definitions that are either not found in other standards or have been modified for use with this standard
4.	Provides conventions for the notation used in this standard
5.	Provides an overview of the information model specified by the standard
6.	Defines the datatypes used in this standard
7.	Defines object properties common to all objects specified in this standard
8.	Defines the top-level objects of an IEEE 1451.1 system
9.	Defines the Block classes
10.	Defines the Component classes
11.	Defines the Service classes
12.	Defines the properties of publications
13.	Defines standard publications
14.	Defines the encoding and decoding rules
15.	Defines the rules for memory management
16.	Defines requirements for conformance

Copyright © 2000 IEEE. All rights reserved.

Annexes are provided as follows:

Annex	Purpose
A	Provides an overview of the use of the object model
B	Provides a detailed explanation of client-server interactions
C	Provides a detailed explanation of publish-subscribe interactions
D	Provides detailed examples of the configuration of systems
E	Provides detailed considerations of interoperability
F	Defines a string character set required for certain strings
G	Defines a string language enumeration
H	Defines a Transducer Block for IEEE 1451.2 transducers
I	Bibliography

1.1 Scope

This standard defines an interface for connecting network-capable processors to control networks through the development of a common control network information object model for smart sensors and actuators.

The object model includes definitions of

— Transducer Blocks

— Function Blocks

— NCAP Blocks

This standard will not define individual device algorithms or specifics on what is implemented by using the model.

1.2 Purpose

Many control network implementations are currently available that allow transducers to be accessed over a network. The purpose of this standard is to provide a network-neutral application model that will reduce the effort in interfacing smart sensors and actuators to a network.

1.3 Benefits

A system designed and built in conformance to this standard is expected to provide the following benefits:

— A uniform design model for system implementation

— A uniform and network-independent set of operations for system configuration

— Defined network-independent models for communication

— Interoperability of all communications

— Defined network-independent models for implementing application functionality

— Portable application models

 Copyright © 2000 IEEE. All rights reserved.

— A network-independent abstraction layer and encode-decode rules that isolate applications from the details of network communications

— A uniform information model for representing physical parametric data

— Uniform models for managing and representing event data, parametric data, and bulk data

— Uniform models for managing and representing time

— Uniform models for intranode concurrency management and components to manage internode concurrency

— Uniform models for memory management

2. References

This standard shall be used in conjunction with the publications and standards listed in this clause. The notation [document-designator] in this standard is a reference to the document in this clause denoted [document-designator].

2.1 General references

[IEEE 754]: IEEE Std 754-1985 (Reaff 1990), IEEE Standard for Binary Floating-Point Arithmetic.[1]

[IEEE 1451.2]: IEEE Std 1451.2-1997, IEEE Standard for a Smart Transducer Interface for Sensors and Actuators—Transducer to Microprocessor Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats.

2.2 References pertaining to String representation

The following standards shall be used in conjunction with the `StringCharacterSet` and `StringLanguage` enumerations (see 6.1.2).

ANSI X3.4 1986, Coded Character Set—7-bit American National Standard Code for Information Interchange.[2]

CNS 11643-1992, (Taiwan) Standard Interchange Code for Generally Used Chinese Characters.[3]

FSS-UTF, File System Safe Universal Transformation Format (FSS_UTF). X/Open CAE Specification C501 ISBN 1-85912-082.[4]

GB 2312-80 (PRC), China State Bureau of Standards. Coded Chinese Graphic Character Set for Information Interchange.[5]

[ISO 639]: ISO 639:1988-04-01 (E/F), Code for the Representation of Names of languages. Later editions of this standard shall not be used.[6]

---

[1]IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA (http://www.standards.ieee.org/).

[2]ANSI publications are available from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA (http://www.ansi.org/).

[3]This document is available on the World Wide Web at the following site: http://www.imc.org/rfc1922.

[4]This document is available by contacting X/Open company, Ltd.—USA, 3141 Fairview Park Drive., Suite 670, Falls Church, VA 22042-4501, USA.

[5]CSBS documents are available from the China State Bureau of Standards, P.O. Box 8010, 42 Hi Chun Road, Haidian District, Beijing 100088, China.

Copyright © 2000 IEEE. All rights reserved.

ISO/IEC 646:1991, Information technology—ISO 7-bit coded character set for information interchange.

ISO/IEC 2022: 1994, Information technology—ISO 7-bit and 8-bit coded character sets code extension techniques.

ISO/IEC 6429: 1992, Information technology—Control functions for coded character sets.

ISO/IEC 8859-1: 1998, Information processing—8-bit single-byte coded graphic character set—Part 1: Latin alphabet No. 1.

ISO/IEC 8859-2: 1999, Information processing—8-bit single-byte coded graphic character set—Part 2: Latin alphabet No. 2.

ISO/IEC 8859-3: 1999, Information processing—8-bit single-byte coded graphic character set—Part 3: Latin alphabet No. 3.

ISO/IEC 8859-4: 1998, Information processing—8-bit single-byte coded graphic character set—Part 4: Latin alphabet No. 4.

ISO/IEC 8859-5: 1999, Information technology—8-bit single-byte coded graphic character sets—Part 5: Latin/Cyrillic alphabet.

ISO/IEC 8859-6: 1999, Information technology—8-bit single-byte coded graphic character sets—Part 6: Latin/Arabic alphabet.

ISO/IEC 8859-7: 1987, Information processing—8-bit single-byte coded graphic character set—Part 7: Latin/Greek alphabet.

ISO/IEC 8859-8: 1999, Information technology—8-bit single-byte coded graphic character sets—Part 8: Latin/Hebrew alphabet.

ISO/IEC 8859-9: 1999, Information technology—8-bit single-byte coded graphic character sets—Part 9: Latin alphabet No. 5.

ISO/IEC 8859-10: 1998, Information Technology—8-bit single-Byte coded graphic.

ISO/IEC DIS 10646-1: 1993, Information technology—Universal multiple-octet coded character set (UCS)—Part 1: Architecture and basic multilingual plane.

ISO/IEC DIS 10646-2: 1993, Information technology—Universal Multiple-Octet Coded Character Set (UCS)—Part 2: Ideographic Character Sets.

JIS X 0208:1997, Japanese Standards Association. Code of the Japanese Graphic Character Set for Information Interchange.[7]

JIS X 0212-1990, Japanese Standards Association. Code of Supplementary Japanese Graphic Character Set for Information Interchange.

KS X 1001:1992, Korea Bureau of Standards. Korean Graphic Character Set for Information Interchange.[8]

---

[6]ISO publications are available from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse (http://www.iso.ch/). ISO publications are also available in the United States from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA (http://www.ansi.org/).

[7]JIS documents are available from the Japanese Standards Association, 1-24, Akasaka 4-chome, Minato-Ka, Tokyo, Japan 107.

 Copyright © 2000 IEEE. All rights reserved.

KS X 1002:1991, Korea Bureau of Standards. Korean Graphic Character Set for Information Interchange. (supplemental character set).

TIS 620-2533 1990, Character Codes for Computers.[9]

2.3 References pertaining to uncertainty

[ISA uncertainty]: Measurement Uncertainty: Methods and Applications, Ronald H. Dieck (This document refers to and is compatible with [ISO uncertainty].) Instrument Society of America, ISBN 1-55617-628-7.[10]

[ISO uncertainty]: Guide to the expression of uncertainty in measurement, International Organization for Standardization, first edition 1995. ISBN 92-67-10188-9.

[NIST]: NIST Technical Note 1297, 1994 edition: Guidelines for evaluating and expressing the uncertainty of NIST measurement results, Barry N. Taylor and Chris E. Kuyatt. Produced by the Physics Laboratory, National Institute of Standards and Technology, United States Department of Commerce, Gaithersburg, MD 20899-0001, USA. (This NIST publication refers to and is compatible with [ISO uncertainty].)[11]

3. Definitions

Most terms in the text of this standard appearing with the first letter of each word capitalized will be found in the subclauses identified; see 4.2.4. Such terms shall have the IEEE 1451.1-specific interpretations given in this clause. Exceptions are some terms in examples or informative material that are defined where they are used or are obvious from the context. Terms in this clause in uncapitalized form represent common parlance terms with a specific interpretation to be used in this standard.

3.1 Abstract Class: A class that can not be instantiated but only subclassed as specified in 7.1.1.

3.2 Action: An instance of the class `IEEE1451_Action` or of a subclass thereof; see 10.10.

3.3 Active Concurrency: Concurrent Execution that appears to be independent as specified in 7.2.4.2.2.

3.4 actuator: A component that provides a physical output in response to a stimulating variable or signal.

3.5 Application System: The collection of components in a system built according to this standard as specified in 5.1.

3.6 argument: The usual mathematical meaning.

3.7 Argument: A value of type `Argument`; see 6.2.14.

3.8 Argument Array: A value of type `ArgumentArray`; see 6.2.14.

3.9 Array: Denotes the IEEE 1451.1 array datatype; see 6.2.1.

---

[8]KS documents are available from the Korean Bureau of Standards, 2 Chung-ang-dong Kwach'on-city, Kyonggi-do 171-11, Korea.

[9]TIS documents are available from the Thai Industrial Standards Institute, Rama 6 Street, Bangkok 10400, Thailand, (tel. 202-3348; fax 247-8739).

[10]ISA documents are available from the Instrument Society of America, 67 Alexander Drive, P.O. Box 12277, Research Triangle Park, NC 27709.

[11]This document is available from the Superintendent of Documents, United States Government Printing Office, Washington, DC 20402, USA.

Copyright © 2000 IEEE. All rights reserved.

3.10 Associated Block Cookie: A particular Block Cookie associated with an Object as specified in 7.3.1.

3.11 Asynchronous Client Port: An instance of the class `IEEE1451_AsynchronousClientPort` or of a subclass thereof; see 11.5.

3.12 asynchronous communication: In the IEEE 1451.1 client-server model, refers to a communication in which the client does not block. State is maintained to allow the client to determine whether the return has been received from the server and to permit the client to retrieve the return.

3.13 Base Client Port: An instance of a subclass of `IEEE1451_BaseClientPort`; see 11.3.

3.14 Base Port: An instance of a subclass of `IEEE1451_BasePort`; see 11.2.

3.15 Base Publisher Port: An instance of a subclass of `IEEE1451_BasePublisherPort`; see 11.6.

3.16 Base Transducer Block: An instance of a subclass of the class `IEEE1451_BaseTransducerBlock`; see 9.4.

3.17 behavior: A statement of the externally visible response and internal change of state of an object to invoked operations or internal events, given its current internal state.

3.18 Block: An instance of a subclass of `IEEE1451_Block`; see 9.1.

3.19 Block Cookie: The value of the cookie of a specific Block; see 9.1.3.3.

3.20 Block Major State: One of the states specified for a Block class in 9.1.3.2.

3.21 Block State Machine: The state machine specified for the referenced Block. The general Block State Machine for all Blocks is specified in 9.1.3.2.

3.22 boot-up: The process an NCAP and its operating system perform, usually on application of power, in preparation for executing operations related to the application and application visible components of the system. System level network visible actions may be accomplished as well, for example, the publication of `PSK_NCAPBLOCK_ANNOUNCEMENT`.

3.23 channel: A physical or logical communication link to a single transducer or to a group of transducers considered as a single transducer.

3.24 class: A collection of objects that share common characteristics and features.

3.25 Class ID: All classes have a descriptive header entry “Class ID,” the value of which is the Class ID (note capitalization); see 7.1.1. For any object of class `IEEE1451_Root`, the operation `GetClassID` returns a value, `class_id`, that has the same value as Class ID; see 8.1.1.2.

3.26 Class Name: All classes have a descriptive header entry “Class,” the value of which is the Class Name (note capitalization), see 7.1.1. For any Object, the operation `GetClassName` returns a value, `class_name`, that has the same value as Class Name; see 8.1.1.1.

3.27 Client Cached Block Cookie: A particular Block Cookie associated with a Base Client Port as specified in 11.3.2.3.

3.28 Client object: Any object that invokes operations on other objects. Also, the client in a client-server communication as modeled in this standard, in which case the Server operation is invoked via an `IEEE1451_BaseClientPort` Object.

 Copyright © 2000 IEEE. All rights reserved.

3.29 Client Port: An instance of the class `IEEE1451_ClientPort` or of a subclass thereof; see 11.4.

3.30 client-server communication: A communication pattern, where a specific object, the client, communicates in a one-to-one fashion with a specific server object, the server.

3.31 commissioning: The process of providing to the appropriate components, the information necessary for the designed communication between components.

3.32 Component: An instance of a subclass of `IEEE1451_Component`; see 10.1. The term component, uncapitalized, has its usual common parlance meaning.

3.33 Component Group: An instance of the class `IEEE1451_ComponentGroup` or of a subclass thereof; see 10.13.

3.34 Concurrent Execution: A form of simultaneous execution of operations as specified in 7.2.4.2.

3.35 Condition Variable Service: An instance of the class `IEEE1451_ConditionVariableService` or of a subclass thereof; see 11.12.

3.36 Configurable Startup Set: A Component Group Object owned by an NCAP Block Object for use in bringing the system to a known state as specified in 9.2.5.8.

3.37 configuration: A process often synonymous with commissioning. Often includes the selection of attributes of an object that change its appearance or performance characteristics as opposed to its communication properties.

3.38 configuration, collection: In a collection of objects, the configuration is the enumeration of the members of the collection, and the specification of the allowed communications between the members. To configure a collection means to make the necessary changes to the collection and its members to make real the defined enumeration of members and the specification of allowed communication between members**.**

3.39 configuration, object: In an object, the configuration is the specification of its allowed communications and of the internal state or organization of the object. To configure an object means to make the necessary changes to the object to make real these specifications.

3.40 cookie: A quantity used to indicate or signal to a recipient of data, significant changes in the state of the entity supplying the data.

3.41 data, analog: Data that represents a variable that is mathematically continuous in the domain of the application. For example, the measurements of time, velocity, pressure, are all continuous variables, excluding quantum effects**.**

3.42 data, digital: Data that does not admit to representations obeying the laws of arithmetic. Examples of digital data are digital state vectors, image bit maps, and proximity switch outputs.

3.43 data, discrete: Data that represents a variable that can be mathematically represented in integer form. For example, a count of whole items processed is a discrete variable.

3.44 dimension: The number of degrees of freedom of a physical quantity. For example, an electric field, which is a spatial vector quantity, has dimension 3.

3.45 Direct Ownership: A relationship between a Block and an object as specified in 5.1.3.

Copyright © 2000 IEEE. All rights reserved.

3.46 domain: When used in this standard in contexts other than publish-subscribe communications, the term domain, uncapitalized, has its usual mathematical interpretation.

3.47 Domain: In IEEE 1451.1 publish-subscribe communications, the Domain is a publication-specific scoping mechanism that defines the set of entities that can potentially communicate via the publish-subscribe model; see 12.3.5. Specifically, the term Domain shall denote a value having type `PubSubDomain`.

3.48 Dot2 Transducer Block: Denotes an instance of the class `IEEE1451_Dot2TransducerBlock` or of a subclass thereof; see 9.6.

3.49 Dot3 Transducer Block: An instance of the class `IEEE1451_Dot3TransducerBlock` or of a subclass thereof; see 9.7.

3.50 Dot4 Transducer Block: An instance of the class `IEEE1451_Dot4TransducerBlock` or of a subclass thereof; see 9.8.

3.51 DotX Transducer Block: A Transducer Block implementing one of the IEEE 1451.X family of standards, see 16.3.6.

3.52 encode/decode: Encoding is the mapping of typed information from its internal datatype format into the types allowed by the signatures of the `Perform-`, `Execute-`, and `Publish-`like operations. Decoding is the mapping from the types allowed by the signatures of the `Perform-`, `Execute-`, and subscription callback-like operations into the datatypes used internally.

3.53 entity: Signifies the hardware/software embodiment of an object.

3.54 Entity: An instance of a subclass of `IEEE1451_Entity`; see 8.2.

3.55 epoch: The reference time defining the origin of the time scale used in a particular measurement.

3.56 event: An abstraction of the mechanism by which asynchronously generated signals or conditions are generated and represented.

3.57 Event Generator Publisher Port: An instance of the class `IEEE1451_EventGeneratorPublisherPort` or of a subclass thereof; see 11.9.

3.58 file: An abstraction of the mechanism for the allocation, deallocation, initialization, and use of memory resources in a device.

3.59 File: An instance of the class `IEEE1451_File` or of a subclass thereof; see 10.11.

3.60 Full Implementation: Refers to the implementation of an operation on an Object. A Full Implementation means that all the referenced visible functionality of the operation is implemented as specified for the operation; see 7.2.1.2.2.

3.61 Function Block: An instance of a subclass of `IEEE1451_FunctionBlock`, see 9.3.

3.62 Get Semantics: Denotes retrieval semantics as specified in 7.2.4.1.1.

3.63 Human-Machine Interface (HMI): Includes keyboards, displays, keypads, touch screens, and similar devices to allow human interaction with a system.

 Copyright © 2000 IEEE. All rights reserved.

3.64 hot swap: A disconnection of a subcomponent and a reconnection of the same or a replacement component without first turning off the power at the connection interface.

3.65 IEEE 1451.1 language: Consists of the `StringLanguage`, `StringCharacterCode`, and `StringCharacterSet` values defined in Annex F. This language is used in certain Strings specified in this standard.

3.66 Indirect Ownership: A relationship between a Block and an object as specified in 5.1.3.

3.67 initialization: A process of setting initial values of variables, constants, state, and other artifacts to establish the startup conditions for an object.

3.68 instance: The specific object that results from allocating resources to implement a single consistent set of internal variables required for the operations and behavior defined for a class definition.

3.69 Interface Definition Language (IDL): A programming language-independent method of specifying operation syntax.

3.70 Interface Only Implementation: Refers to the implementation of an operation on an Object. This means that only the interface properties of the operation are implemented. The operation will return an appropriate error code but otherwise have no other effect on the Object; see 7.2.1.2.1.

3.71 Local: As an adjective shall indicate that the modified term's extent is a single process space. Thus, a Local operation may be invoked only from within the process space of the object; see 5.1.2.3.

3.72 Local NCAP Block: The unique network capable application processor (NCAP) Block within a process as specified in 5.1.2.2.

3.73 Major Field: A field of a return code as specified in 7.2.3.1.1.

3.74 mapping onto network protocol: The embodiment of hardware and software as it relates to supporting IEEE 1451.1 communications on a specific bus standard.

3.75 marshaling/demarshaling: The mapping of information typed as given in the signatures of `Perform-`, `Execute-`, and `Publish`-like operations into the network-specific, on-the-wire formats, including any required endian issues. Demarshaling is the reverse process.

3.76 Minor Field: A field of a return code as specified in 7.2.3.1.1.

3.77 mutex: A mechanism for implementing mutual exclusion.

3.78 Mutex Service: An instance of the class `IEEE1451_MutexService` or of a subclass thereof; see 11.11.

3.79 NCAP Block: An instance of the class `IEEE1451_NCAPBlock` or of a subclass thereof; see 9.2.

3.80 Network Capable Application Processor (NCAP or IEEE1451_NCAP): A device that supports a network interface, application functionality, and generally access to the physical world via one or more transducers. An IEEE1451_NCAP is an NCAP conformant to this standard**.**

3.81 network: A communication mechanism for interconnecting multiple Smart Transducers.

3.82 network infrastructure: The network and associated software used to communicate between network capable application processors (NCAPs) as specified in 5.3.

Copyright © 2000 IEEE. All rights reserved.

3.83 Network Visible: Refers to objects or operations that can be accessed or invoked from a different process than the process in which the object or operation executes, in particular if the processes are in different network capable application processors (NCAPs) communicating over a network. In IEEE 1451.1, Network Visible objects or operations are also locally visible; see 5.1.2.3.

3.84 object: A collection of data and operations.

3.85 Object: An instance of the class `IEEE1451_Entity` or of a subclass thereof; see 8.2.

3.86 Object Dispatch Address: A network-specific identifier for an endpoint of a client-server communication. Specifically, a value having datatype `ObjectDispatchAddress`; see 6.1.3.

3.87 Object ID: A unique identifier for an instance of an Object as specified in 8.2.3.5.

3.88 object model: A definition of data structures and operations organized in a formal specification. An object model provides applications with a common view and a common way of interfacing to an element of functionality.

3.89 Object Name: A nonconfigurable name for an instance of an Object used to convey the purpose or function of the Object instance. For any Object, the operation `GetObjectName` returns a value, `object_name`, that has the same value as Object Name; see 8.2.1.4.

3.90 Object Tag: A configurable identifier for the endpoints of client-server communications. Specifically a value having datatype `ObjectTag`; see 6.2.3. For any Object, the operation `GetObjectTag` returns a value, `object_tag`, that has the same value as Object Tag; see 8.2.1.1.

3.91 Octet Array: A value of type `OctetArray`; see 6.2.1.

3.92 operation: At the specification level, an operation is a service that a class knows how to carry out. In an implementation of the IEEE 1451.1 standard, an operation on a specified class is mapped to a public method on the corresponding realized class.

3.93 Operation ID: A reference to an operation as specified in 7.2.2.4.

3.94 Operation Name: A name given to an operation defined for an IEEE 1451.1 class; see 7.2.2.3.

3.95 orientation: The positioning of the dimensions of a spatial vector. For example, spatial vectors may be represented by a cartesian coordinate system oriented in space in some application-specific manner.

3.96 Own: A relationship between objects as specified in 5.1.3.

3.97 Owning Block: A relationship between Block and other objects as specified in 5.1.3.1.

3.98 parameter: An attribute usually representing some property subject to change, for example, a configuration parameter.

3.99 Parameter: An instance of the class `IEEE1451_Parameter` or of a subclass thereof; see 10.2. A Parameter is a representation of a variable. Parameters may selectively be manipulated across a network, and can be an externally visible representation for data.

3.100 Parameter With Update: An instance of the class `IEEE1451_ParameterWithUpdate` or of a subclass thereof; see 10.3.

 Copyright © 2000 IEEE. All rights reserved.

3.101 parametric data: Data representing an internal state variable of an object that is represented by a Parameter. More specifically, physical parametric data generally represents some aspect of the physical world.

3.102 partition: A subdivision of a file.

3.103 Partition: An addressable portion of a Partitioned File; see 10.12.

3.104 Partitioned File: An instance of the class `IEEE1451_PartitionedFile` or of a subclass thereof; see 10.12.

3.105 Permanent Startup Set: A Component Group Object owned by an NCAP Block Object for use in bringing the system to a known state as specified in 9.2.5.8.

3.106 Physical Parameter: An instance of a subclass of `IEEE1451_PhysicalParameter`; see 10.4.

3.107 Physical Parameter Type: The syntax and interpretation of the Physical Parameter's data and metadata. The value of Physical Parameter Type for a given Physical Parameter is determined as specified in 10.4.1.1.

3.108 Physical Units: The designator of the units of a variable of datatype `Units` using the conventions of [IEEE 1451.2] as specified in H.4.2.4.

3.109 port: An abstraction of an access point to network communications.

3.110 Port: A Port Object. Context may indicate that the Port Object is of a specific class.

3.111 Port Object: Any Object whose class is `IEEE1451_SubscriberPort` or a subclass thereof or a subclass of `IEEE1451_BasePort`.

3.112 Precedence Concurrency: Concurrent Execution that may appear to preempt other activity as specified in 7.2.4.2.2.

3.113 process: Consists of all execution within a single distinct address space supported by the operating system of a computer.

3.114 Public Transducer: A Parameter or other class instance, that is the public interface or abstraction of one or more physical transducers supported by an NCAP; see 9.5.

3.115 Publication Change ID: A publisher-defined identifier of the semantics of the publication as specified in 11.8.3.4.2.

3.116 Publication Contents: The publisher-defined contents of a publication. The syntax and other properties of Publication Contents shall be as specified in 11.7 and 11.8 for Publisher Ports and for individual publications where they are defined. Publication Contents is always of type `ArgumentArray`.

3.117 Publication Domain: A Domain for a specific publication. See also: Domain.

3.118 Publication Key: A publisher-defined identifier specifying the form and contents of a publication in a publish-subscribe communication. Specifically, a value having the datatype and properties specified in 12.3.2. For a Publisher Port, the operation `GetPublicationKey` returns a value, `publication_key,` that has the same value as Publication Key; see 11.6.1.3.

Copyright © 2000 IEEE. All rights reserved.

3.119 Publication Topic: A configurable identifier for the name of a publication in a publish-subscribe communication. Specifically a value having datatype `PublicationTopic`; see 6.2.8. For a Publisher port, the operation `GetPublicationTopic` returns a value, `publication_topic`, that has the same value as Publication Topic; see 11.6.1.1.

3.120 Publisher object: Any object that posts publications onto the network via an Object of class `IEEE1451_BasePublisherPort` or a subclass thereof.

3.121 Publisher Port: An instance of the class `IEEE1451_PublisherPort` or of a subclass thereof; see 11.7.

3.122 publish-subscribe communication: A communication pattern where one or more objects, publishers, communicate information on a specific topic to one or more subscriber objects interested in that topic without the necessity of any of these objects knowing the identity of any other object. In an IEEE 1451.1 system, the pattern is established via the Publication Topic, Publication Key, and Publication Domain of the publication.

3.123 return code: A value, returned to the caller of an operation, providing information about the completion status of the operation. This standard defines two forms of return codes; see 7.2.3.1.

3.124 Root: An instance of a subclass of `IEEE1451_Root`; see 8.1.

3.125 Scalar Parameter: An instance of the class `IEEE1451_ScalarParameter` or of a subclass thereof; see 10.5.

3.126 Scalar Series Parameter: An instance of the class `IEEE1451_ScalarSeriesParameter` or of a subclass thereof; see 10.6.

3.127 Self Identifying Publisher Port: An instance of the class `IEEE1451_SelfIdentifyingPublisherPort` or of a subclass thereof; see 11.8.

3.128 sensor: A component providing a useful output in response to a physical, chemical, or biological phenomenon. This component may already have some signal conditioning associated with it. Examples: platinum resistance temperature detector, humidity sensor with voltage output, light sensor with frequency output, pH probe, and piezoresistive bridge.

3.129 Series Parameter: A Scalar Series Parameter or a Vector Series Parameter.

3.130 Server Object: Any Object that executes one or more of its operations in response to a request from a Client object. A Server Object is the server in a client-server communication as modeled in this standard.

3.131 Server Object Tag: An attribute of a Client Port that identifies the Object Tag of the Server Object with which the Port communicates in client-server communications; see 11.3.1.1.1.

3.132 Service: An instance of a subclass of `IEEE1451_Service`; see 11.1.

3.133 Set Semantics: Update semantics as specified in 7.2.4.1.2.

3.134 Smart Transducer: A transducer that provides functions over and above that necessary for generating a correct representation of a sensed or controlled physical quantity. This functionality typically simplifies the integration of the transducer into applications in a networked environment.

3.135 Smart Transducer object model: An object model for a Smart Transducer. The model includes an interface to a transducer object and a transducer bus.

 Copyright © 2000 IEEE. All rights reserved.

3.136 state transition diagram: A graphical means of expressing the allowed states of an object and the allowed transitions from one state to another; see 4.6.

3.137 Smart Transducer Interface Module (STIM): The supporting electronics on the transducer side of the hardware interface to the NCAP. In IEEE 1451.1, an STIM and an NCAP combined form a Networked Smart Transducer. In the various IEEE 1451.X standards, for example IEEE 1451.2, the term STIM may have a more precise meaning within the scope of the particular standard.

3.138 String: The IEEE 1451.1 representation of a sequence of human readable characters; see 6.1.2.

3.139 Subscriber object: Any object that receives publications from the network via an Object of class `IEEE1451_SubscriberPort`.

3.140 Subscriber Port: An instance of the class `IEEE1451_SubscriberPort` or of a subclass thereof; see 11.10.

3.141 Subscription Domain: A Domain identified by a subscriber for a specific publication. See also: Domain.

3.142 Subscription Key: The Subscriber defined analog of the Publication Key used to help define the selection of received publications in a publish-subscribe communication. Specifically, a value having the datatype and properties specified in 12.3.2. For a Subscriber Port, the operation `GetSubscriptionKey` returns a value, `subscription_key,` that has the same value as the Port's Subscription Key; see 11.10.1.3.

3.143 Subscription Qualifier: A configurable identifier used to help define the selection of received publications on the basis of publication's Publication Topic in a publish-subscribe communication. Specifically, a value having datatype `SubscriptionQualifier`; see 6.2.8. For a Subscriber Port, the operation `GetSubscriptionQualifier` returns a value, `subscription_qualifier`, that has the same value as Port's Subscription Qualifier; see 11.10.1.1.

3.144 synchronous communication: In the IEEE 1451.1 client-server model, refers to a stateless communication in which the client blocks until the return is received from the server. That is, the client can perform no activities until unblocked by the return of the result.

3.145 Transducer Electronic Data Sheet (TEDS): Several of the IEEE 1451.X standards use TEDS to provide a machine-readable specification of the characteristics of the transducer interface.

3.146 Time Parameter: An instance of the class `IEEE1451_TimeParameter` or of a subclass thereof; see 10.9.

3.147 timeout: A mechanism for terminating requested activity that, at least from the requester's perspective, does not complete within the time specified by the timeout's "value."

3.148 transducer: A device converting energy from one domain into another, calibrated to minimize the errors in the conversion process. A sensor or an actuator.

3.149 Transducer Block: An instance of a subclass of `IEEE1451_TransducerBlock`; see 9.5

3.150 units: The units of a measured value of a physical variable define the standard quantity of the measure of that variable used to express the value. The representation of the units of an Object is the datatype `Units`; see 6.2.9.

3.151 unrepresented channel: A channel that is not represented by a Public Transducer.

Copyright © 2000 IEEE. All rights reserved.

3.152 Vector Parameter: An instance of the class `IEEE1451_VectorParameter` or of a subclass thereof; see 10.7.

3.153 Vector Series Parameter: An instance of the class `IEEE1451_VectorSeriesParameter` or of a subclass thereof; see 10.8.

3.154 visibility, Local: Operations or objects are locally visible if they can not be accessed or invoked except from within the process in which the object or operation executes. In IEEE 1451.1, objects not specifically designated as Network Visible are only Locally visible.

4. Conventions

The specifications of this clause shall apply to all objects defined in this standard, and where appropriate, to any Objects defined by subclassing class definitions found in this standard.

4.1 Class naming

The names of all classes specific to this standard shall have a prefix "`IEEE1451_`". The purpose is to allow specifications or source code implementing this standard to be easily searched for Class Names associated with this standard.

4.2 Descriptive syntax

The syntax conventions specified in 4.2.1, 4.2.2, 4.2.3, and 4.2.4 are used in this standard.

4.2.1 Lexical form syntax

A lexical form refers to

a) An operation signature

b) Element of a signature

c) A datatype

When a lexical form appears in the text of the standard, it will be in the special computer font and will obey the conventions stated below. The computer font is Courier whereas normal font is Times. In particular, when an IEEE 1451.1-defined datatype is referred to in text, it will always appear in computer font. Likewise, the formal name or the formal name standing for the value of an argument of an IEEE 1451.1-defined operation will always appear in computer font.

In addition, the conventions illustrated in the following list regarding lexical forms shall be used:

— Type names: e.g., `TemperatureSensor` (no word separation, initial letter of each word capitalized, computer font).

— Class Names: e.g., `IEEE1451_FunctionBlock` (no word separation, initial letter of each word capitalized, prefixed by `IEEE1451_` if specified by this standard, computer font).

— Class operations: e.g., `OpenValve` (no word separation, initial letter of each word capitalized, computer font).

— Enumeration members and global constants: e.g., `HIGH, TEMPERATURE_SETPOINT` (underscore word separation, all letters capitalized, computer font).

 Copyright © 2000 IEEE. All rights reserved.

— Fields of structures: e.g., `setPoint, element, specialElement` (no word separation, initial word not capitalized, initial letter capitalization on subsequent words, computer font).

— All other variables: `read, read_status` (underscore word separation, no letters capitalized, computer font).

— `<local name for something>:` text enclosed in angle brackets, < >, is used where the standard needs to refer to something whose syntax or lexical form is dependent on the local implementation and language. In text, these will appear in computer font.

Example: “The value returned by the operation `GetClassID` for this class shall be 1.2.” “The datatype of `member_properties` is `ObjectProperties`.”

When a lexical form appears in text, as opposed to in a signature, a type, or a format definition, the form is to be interpreted as singular, plural, or possessive as appropriate to the context of the text. This is to avoid computer font terms with plural or possessive endings in normal font.

4.2.2 Reference to a collection of specific IEEE 1451 Objects

When a lexical form in the standard refers to one or a collection of specific IEEE 1451.1-specified Objects or their values, the form will be in normal font with the first letter capitalization on each word. A definition and clause reference for all such lexical forms appears in Clause 3.

Example: “All Function Blocks will have a state machine,” where Function Blocks refers to instances of the class `IEEE1451_FunctionBlock`.

4.2.3 General concepts

A general concept, that is the common parlance or mathematical meaning of a term, will not be capitalized, and will appear in normal font.

Example: “Applications are constructed using objects.”

4.2.4 Terms

Terms for concepts that have been given a specific meaning in this standard will appear in normal font with the first letter capitalized on each word. A definition and clause reference for all such lexical forms appears in Clause 3.

Example: “All Network Visible Objects are registered.”

4.3 Word usage

4.3.1 Shall

The word shall, equivalent to “is required to,” is used to indicate mandatory requirements, strictly to be followed in order to conform to the standard and from which no deviation is permitted.

4.3.2 Recommended

The word recommended is used to indicate flexibility of choice with a strong preference alternative.

Copyright © 2000 IEEE. All rights reserved.

4.3.3 Must

The word must is NOT used. The use of the word must is deprecated and shall not be used when stating mandatory requirements. The word must is used only to describe unavoidable situations.

4.3.4 Should

The word should, equivalent to “is recommended that,” is used to indicate

— Among several possibilities one is recommended as particularly suitable, without mentioning or excluding others.

— That a certain course of action is preferred but not necessarily required.

— That (in the negative form) a certain course of action is deprecated but not prohibited.

4.3.5 May

The word may, equivalent to “is permitted,” is used to indicate a course of action permissible within the limits of the standard.

4.3.6 Can

The word can, equivalent to “is able to,” is used to indicate possibility and capability, whether material or physical.

4.4 Class definitions

Class definitions are described by using schema. Each schema consists of one or more of the following elements as needed:

— The name of the class

— The name of the class’s parent

— The class’s ID

— A description of the class

— A list of the class’s Network Visible operations

— A list of non-Network Visible, that is Local only, operations of the class

— A list of publications and subscriptions for the class

— State machine diagrams and other material defining the class’s behavior

An operation is the term standardized in the Unified Modeling Language (UML) [B5], and is used for all IEEE 1451.1 class descriptions. It refers to what some programming languages call a method or a member function.

An implementation of an IEEE 1451.1 class may have other, unlisted operations that are not visible over the network. However, unless otherwise noted, these operations are not in the scope of this standard and are not included in the class definitions.

Non-Network Visible operations that are specified in the class definition are indicated in the schema by preceding the operation designation by the word “Local.”

 Copyright © 2000 IEEE. All rights reserved.

Operation signatures, enumerations, structures, and other IEEE 1451.1-specified data types are presented in a form of IDL [B3].

All IDL specifications in this document are preceded by the characters “IDL:” in bold type. If an electronic copy of this document is parsed with an appropriate script, the IDL may be extracted for use in code development.

4.5 Operation signature datatypes

The datatypes appearing in the signatures of operations shall define the datatype used to represent the actual values of the referenced information. These datatypes shall not be construed as limiting implementations to “pass by value” or any other technique. It is recommended that vendors provide documentation defining the mapping of the signatures of all IEEE 1451.1 operations to the vendor’s chosen language and implementation.

The only data types that shall occur in the IDL signature of an IEEE 1451.1-specified operation are those data types that are defined in the IEEE 1451.1 specification. IDL has no concept of a pointer or a reference to a datum. This does not imply that all Network Visible operation arguments and return values are “pass by value.” Rather, the actual types of these arguments are IEEE 1451.1 implementation and implementation language dependent.

For example:

IF: The IDL specification of the signature of the operation is:

```
IDL: OpReturnCode Write(
           in UInteger16 transaction_id,
           in UInteger32 requested_number_of_octets,
           in OctetArray data,
           out UInteger32
                     actual_number_of_octets_written);
```

THEN

For a C implementation: The header for the `Write` operation might be:

```
OpReturnCode Write(
           /*in*/ UInteger16 transaction_id,
           /*in*/ UInteger32
                     requested_number_of_octets,
           /*in*/ OctetArray *data,
           /*out*/ UInteger32
                     *actual_number_of_octets_written);
```

OR

For a C++ implementation: The header for the `Write` operation might be:

```
OpReturnCode Write(
           /*in*/ UInteger16 transaction_id,
           /*in*/ UInteger32
                     requested_number_of_octets,
           /*in*/ OctetArray &data,
```

Copyright © 2000 IEEE. All rights reserved.

```
            /*out*/ UInteger32 &actual_number_of_octets_written);
```

OR

For a Java implementation: The signature of the `Write` operation might be:

```
OpReturnCode Write(
            /*in*/ UInteger16 transaction_id,
            /*in*/ UInteger32 requested_number_of_octets,
            /*in*/ OctetArray data,
            /*out*/ Long actual_number_of_octets_written);
```

where `actual_number_of_octets_written` and `data` are declared as object references, for example:

```
            Long actual_number_of_octets_written = new Long (0);
            OctetArray data = new OctetArray( );
```

4.6 Behavioral specification notation

State transition diagrams are used to specify the behavioral characteristics of the object model as illustrated in Figure 2. Each state transition diagram is composed of the following components:

— Named boxes, representing states

— Directed arrows, indicating transitions from one state to the next

Each transition is labeled with

— The enabling event or predicate label for a transition

— The transition action label for a transition

The notation describes state transition diagrams using the Mealy style, where actions are associated with the transition from one state to another.

Events, e.g., event_1, event_2, and event_3, identify the inputs to the state machine. They can be operation requests and responses, or internal occurrences such as timer expirations.

Predicates, e.g., event_1 OR event_2, identify enabling conditions for transitions. The first predicate encountered that is `TRUE` selects the transition to execute and therefore the next state.

Transition actions, e.g., result_1, are the actions that are executed before transitioning to the next state.

The next state identifies the state for the state machine after the selected transition action completes. As the transition to the next state occurs, the value of the current state changes.

A bold line for a state box indicates that the box represents multiple states. Any transition shown that begins and terminates in such a state box indicates that there has been no change in state.

Transitions, e.g., the transition resulting in result_3, that have no indicated enabling conditions, occur via unspecified mechanisms. Unless otherwise stated, in IEEE 1451.1 these mechanisms are implementation-specific and outside the scope of the standard.

 Copyright © 2000 IEEE. All rights reserved.

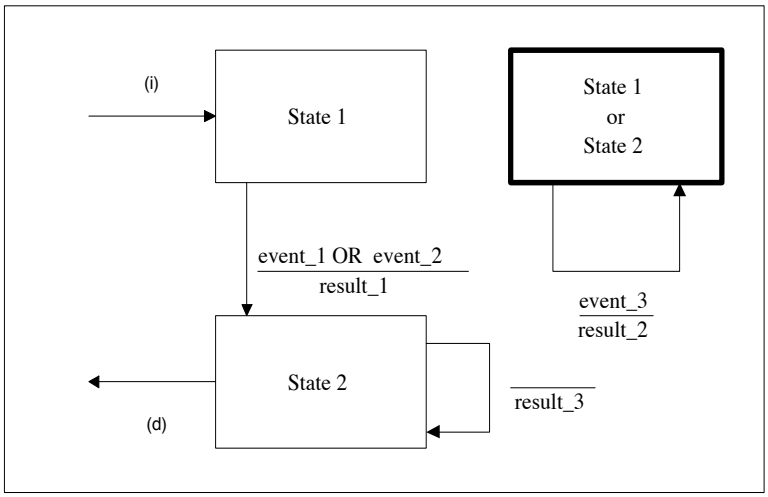

Figure 2—Mealy state transition diagram

A transition into a state machine, for example, (i), is indicated by a transition arrow that has no source state. A transition out of a state machine, for example, (d), is indicated by a transition arrow with no destination state.

Example: As a result of either event_1 or event_2 becoming TRUE, State 1 is replaced with the value of the next state. In this example, the next state is State 2, which is specified as the name of the state box that is the target of the transition arrow. Before this transition, result_1 occurs. event_3 can occur in either State 1 or State 2. The state is unchanged but an action, result_2, occurs as the result of event_3.

5. Information model

The IEEE 1451.1 standard specifies a software architecture. This architecture is applicable to distributed systems consisting of one or more Network Capable Application Processors (NCAPs), communicating over a network. The NCAPs may interact with the physical world via attached transducers.

The standard provides

— A network abstraction layer

— A transducer abstraction layer

The standard specifies

— The software interfaces between application functions on an NCAP and a communication network in a manner independent of any specific network.

— The software interfaces between application functions on an NCAP and transducers attached to that NCAP in a manner independent of any specific transducer driver interface.

Systems implemented according to IEEE 1451.1 standard will achieve a high degree of interoperability regardless of the underlying network or transducer technologies.

The IEEE 1451.1 software architecture is defined via three types of models

— An object model (for the software components of IEEE 1451.1 systems)

Copyright © 2000 IEEE. All rights reserved.

— A data model (for the information communicated across the specified object interfaces)
— Two network communications models

NOTES

1—The IEEE 1451.1 software architecture specification is presented in this document from an object-oriented perspective. However, there is no requirement that actual implementations use object-oriented implementation techniques. For example, a C language implementation is not required to have run time dispatch to handle calls to inherited operations. It may instead directly implement all operations, including inherited operations for an "object" as specific function calls in a code module representing that object's functionality.

2—The standard does not specify any application functions. Application functions are to be constructed by using the standard's models rather than be specified by the models.

3—More detailed descriptions of the use of the software architecture specified by this standard are given in Annex A, Annex B, Annex C, and Annex D.

5.1 Object model specifications

The IEEE 1451.1 object model specifies the software component types used to design and implement Application Systems. The object model provides the software "building blocks" for Application Systems. The standard specifies each object class in the model by defining

— The object class interface (expressed as the set of signatures of the object's operations)
— The object class behavior (described via text and, where appropriate, state machine diagrams)

The IEEE 1451.1 object class hierarchy is shown in Figure 3. The figure gives the abbreviated Class Name (the name with the IEEE 1451 prefix dropped) and the Class ID for each class. The subclass relation is denoted by the indentation levels.

An object shall be termed an Object if, and only if, its class is a subclass of `IEEE1451_Entity`.

The class of any Object occurring in an IEEE 1451.1 system shall be either

— One of the classes in Figure 3 listed in bold type
— A specialization (subclass) of one of those classes

The IEEE 1451.1 classes in Figure 3 listed in italic type are abstract classes, and they shall not be instantiated. An instantiated Object of class A is also an instance of all classes that are parents of A recursively up to and including `IEEE1451_Root`.

The software in an Application System consists of a collection of Objects of class `IEEE1451_Block` and their associated objects.

The IEEE 1451-defined Object types associated with the Block types are Service Objects and Component Objects. The association shall be that of Block Ownership, which is a special form of containment; see 5.1.3.

5.1.1 Object classes

There are four object classes found in an Application System

— Block classes
— Component classes
— Service classes
— Non-IEEE 1451.1 object classes

 Copyright © 2000 IEEE. All rights reserved.

Root 1
- *Entity* 1.1
  - *Block* 1.1.1
    - **NCAP Block** 1.1.1.1
    - *Function Block* 1.1.1.2
    - *Base Transducer Block* 1.1.1.3
      - *Transducer Block* 1.1.1.3.1
        - **Dot2 Transducer Block** 1.1.1.3.1.1
        - **Dot3 Transducer Block** 1.1.1.3.1.2
        - **Dot4 Transducer Block** 1.1.1.3.1.3
  - *Component* 1.1.2
    - **Parameter** 1.1.2.1
      - **Parameter With Update** 1.1.2.1.1
        - *Physical Parameter* 1.1.2.1.1.1
          - **Scalar Parameter** 1.1.2.1.1.1.1
            - **Scalar Series Parameter** 1.1.2.1.1.1.1.1
          - **Vector Parameter** 1.1.2.1.1.1.2
            - **Vector Series Parameter** 1.1.2.1.1.1.2.1
      - **Time Parameter** 1.1.2.1.2
    - **Action** 1.1.2.2
    - **File** 1.1.2.3
      - **Partitioned File** 1.1.2.3.1
    - **Component Group** 1.1.2.4
  - *Service* 1.1.3
    - *Base Port* 1.1.3.1
      - *Base Client Port* 1.1.3.1.1
        - **Client Port** 1.1.3.1.1.1
        - **Asynchronous Client Port** 1.1.3.1.1.2
      - *Base Publisher Port* 1.1.3.1.2
        - **Publisher Port** 1.1.3.1.2.1
        - **Self Identifying Publisher Port** 1.1.3.1.2.2
          - **Event Generator Publisher Port** 1.1.3.1.2.2.1
    - **Subscriber Port** 1.1.3.2
    - **Mutex Service** 1.1.3.3
    - **Condition Variable Service** 1.1.3.4

Figure 3—IEEE 1451.1 class hierarchy

5.1.1.1 Block classes

Three base Block classes are specified by the standard

— `IEEE1451_NCAPBlock` class (provides standard software interfaces for supporting network communications and system configuration)

— `IEEE1451_BaseTransducerBlock` classes (provide standard software interfaces between transducers and application functions)

— `IEEE1451_FunctionBlock` classes (encapsulate application-specific functionality)

5.1.1.2 Component classes

Objects of class Component provide common application constructs

— Structured information, for example, measurements and files

— Collections of related application-specific objects

— Actions with state where the action takes place over a relatively long period of time

Copyright © 2000 IEEE. All rights reserved.

5.1.1.3 Service classes

Service class Objects support

— Communication between objects on distinct NCAPs

— Systemwide synchronization

5.1.1.4 Non-IEEE 1451.1 objects

Nothing in this standard precludes an Application System from making use of objects that are not Objects, that is, of a subclass of `IEEE1451_Entity`. The conformance requirements, see Clause 16, place some restrictions on the areas of applicability of non-IEEE 1451.1 objects. Unless otherwise stated, the specification and behavior of such objects are outside the scope of this standard.

5.1.2 Object relations

There are several binary and unary relations specified on the various types of objects in an IEEE 1451.1 system.

5.1.2.1 The relationship between NCAPs, processes, and Block Objects

The top-level relations between Block Objects, software processes, and an NCAP shall be as illustrated in Figure 4.

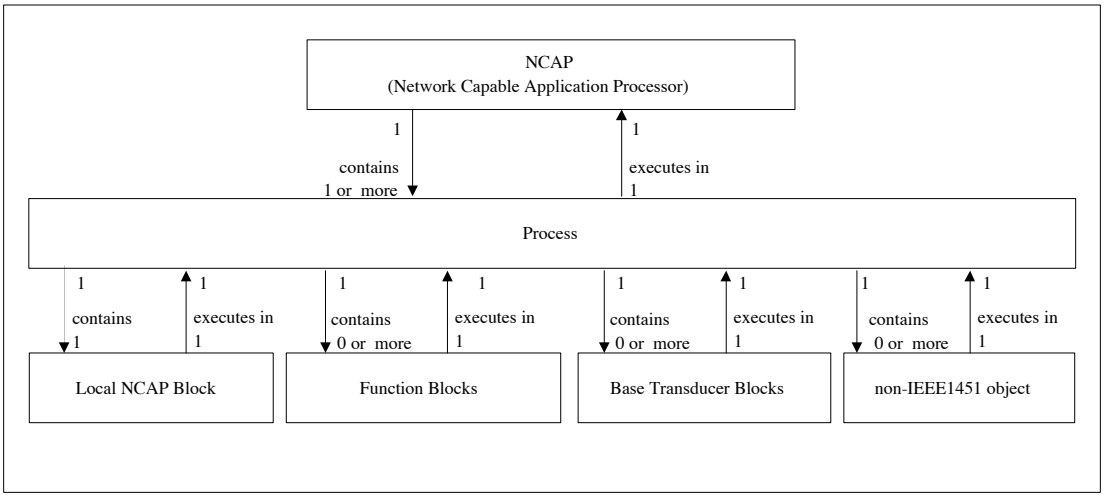

Figure 4—Top-level object relationships

Each NCAP in an Application System is modeled to contain at least one software process. The term *software process* in this standard is used only as a convenient term for defining an address space in which objects in an NCAP reside. The term *process* or any equivalent term shall not appear in the signatures of any Object's operations.

Within a given NCAP, there may be several active software processes. Within each process, there shall be exactly one Object of class `IEEE1451_NCAPBlock` along with zero or more Objects of class `IEEE1451_FunctionBlock`, zero or more Objects of class `IEEE1451_BaseTransducerBlock`, and zero or more non-IEEE 1451 objects. Each object in an Application System shall always be associated with exactly one NCAP process.

 Copyright © 2000 IEEE. All rights reserved.

5.1.2.2 Local NCAP Block definition

Each Object, O, in an Application System exists only within the scope of a unique NCAP process, and within each NCAP process there shall be a unique NCAP Block Object. That NCAP Block Object is called O's Local NCAP Block.

5.1.2.3 Network Visible Object definition

An object shall be Network Visible if, and only if, all of the following apply:

— Its class is a subclass of `IEEE1451_Entity`, **AND**

— It is registered with its Local NCAP Block, (see 9.2), **AND**

— It is Owned by its Local NCAP Block (see 5.1.3).

An NCAP Block Object shall always be made Network Visible. The selection of which additional Block, Component and Service Objects to make Network Visible is application-specific, except where otherwise specified in the standard.

Any Object whose class is defined by this standard that participates in network communications shall be Network Visible. An object is defined as "participating in network communications" if, and only if

— The object directly requests the network infrastructure to transmit a message, **OR**

— The network infrastructure directly invokes an operation on the object.

For each Network Visible Object, whose class is defined by this standard, only those operations specifically designated in the Object's class definition as being Network Visible operations shall be capable of being invoked by the network infrastructure.

Those operations specified in the standard that are not Network Visible operations are denoted as Local operations. Local operations on an object O, shall be invoked only by other objects in O's process space even if O is itself Network Visible. Network Visible operations on a Network Visible Object may be invoked either locally or remotely.

NOTE—The requirement that Objects participating in network communication be Network Visible requires that any Port Object used for network communication be Network Visible. For an instance Object of one of the Port classes defined in this standard (see Clause 11) the Object's Object Tag or Publication Topic is modifiable over the network. If, in a given IEEE 1451.1 system, it is desired that these Port attributes be fixed when a Port Object is created, the appropriate Port class can be subclassed such that the subclass allows the Object Tag or the Publication Topic to be read but not
written.

5.1.3 Object Ownership definition

For each NCAP process space in the system, the Owns relation has as its domain all Block Objects in the process space, and as its range, all objects in the space.

The Owns relation is defined as follows:

— The Local NCAP Block in the space Owns itself, and it shall be the only Block Object in the space to do so, **AND**

— For a Block Object B in the space and an object O in the space, B Owns O (equivalently, O is Owned by B) if and only if O being operational implies that B is operational. Here, an object T is operational

Copyright © 2000 IEEE. All rights reserved.

if and only if T is capable of accepting and processing operation requests from other objects in the space.

The Block Objects in an IEEE 1451.1 system shall satisfy the following Owns relation constraint:

Let $\mathcal{E}$ be the set of all Blocks in an NCAP process space that Own at least one object in the space. Then, for an Owned object O in the space, if B_1 and B_2 are distinct Block Objects in $\mathcal{E}$ that both Own O, then either B_1 Owns B_2 or B_2 Owns B_1.

From this definition of Owns and the above constraint, the properties of the Owns relation in an IEEE 1451.1 system are as follows:

— Only Block Objects may Own objects.

— The Owns relation does not cross process space boundaries.

— The Owns relation is transitive.

— An NCAP Block Object is the only object type that Owns itself.

— An NCAP Block Object is not Owned by any other object.

— The Owns/Owned-by relation defines a tree structure for all the objects in an NCAP process space owned by an NCAP Block where the root of the tree is the Local NCAP Block Object.

— For each Owned object, O, there shall be precisely one Block Object which Owns O and does not Own any other Block Object that also Owns O. This Block Object is said to Directly Own O.

An Owned object O is usually Owned by multiple Block Objects via the transitive nature of the Owns relation. However, there is only one Block Object that Directly Owns O. The other Block Objects that Own O are said to Indirectly Own O.

Specifications in this standard that restrict the behavior of objects Owned by a Block Object on the basis of the state of that Block Object shall use the definition of Owned given in this clause. Restrictions shall apply to the Owned objects irrespective of whether the object is an Object or not, and if an Object is Network Visible or not, unless otherwise stated in the object's class definition. The mechanisms for enforcing ownership responsibilities are implementation-specific and outside the scope of this standard.

NOTE—The Owning relation is used during IEEE 1451.1 system design, boot-up, initialization, and configuration.

5.1.3.1 Owning Block definition

Among the Blocks Owning a Network Visible Object, there shall be a unique Block denoted as the Object's Owning Block. The definition of a Network Visible Object's Owning Block shall be as follows:

Given a process space within an NCAP and O, a Network Visible Object in the space, the Owning Block of O shall be the Block Object B in the space satisfying the following conditions:

— B is Network Visible, **AND**

— If a Network Visible Block B_1 Owns O, then B_1 = B or B_1 Owns B.

From this definition, the properties of Owning Blocks are as follows:

— Every Network Visible Object in the space has an Owning Block. This follows from the definition because the Object's Local NCAP Block is always Network Visible and the Owns relation is transitive.

— The Owning Block of a Network Visible Object in the space is unique.

If a Network Visible Block Object B in a process space has as its Owning Block the space's Local NCAP Block Object, then B is said to be a top-level Block.

 Copyright © 2000 IEEE. All rights reserved.

NOTES

1—The Owning Block relation is used for two primary purposes

— As the basis for mechanisms to cleanly start and shut down an Application System

— To help track and recognize dynamic system modifications, for example, a transducer “hot swap”

2—If a Block B is the Owning Block of an object O, then, by definition, B Owns O. The converse, however, need not be true, as a Block can Own O but not be Network Visible. In this case, that Block is not O’s Owning Block.

5.1.3.2 The relationship of Block Objects to Component, Service, and other objects

The relationship of the Block Objects to the Component Objects, Service Objects, and other objects in a process space of an IEEE 1451.1 system shall be as shown in Figure 5.

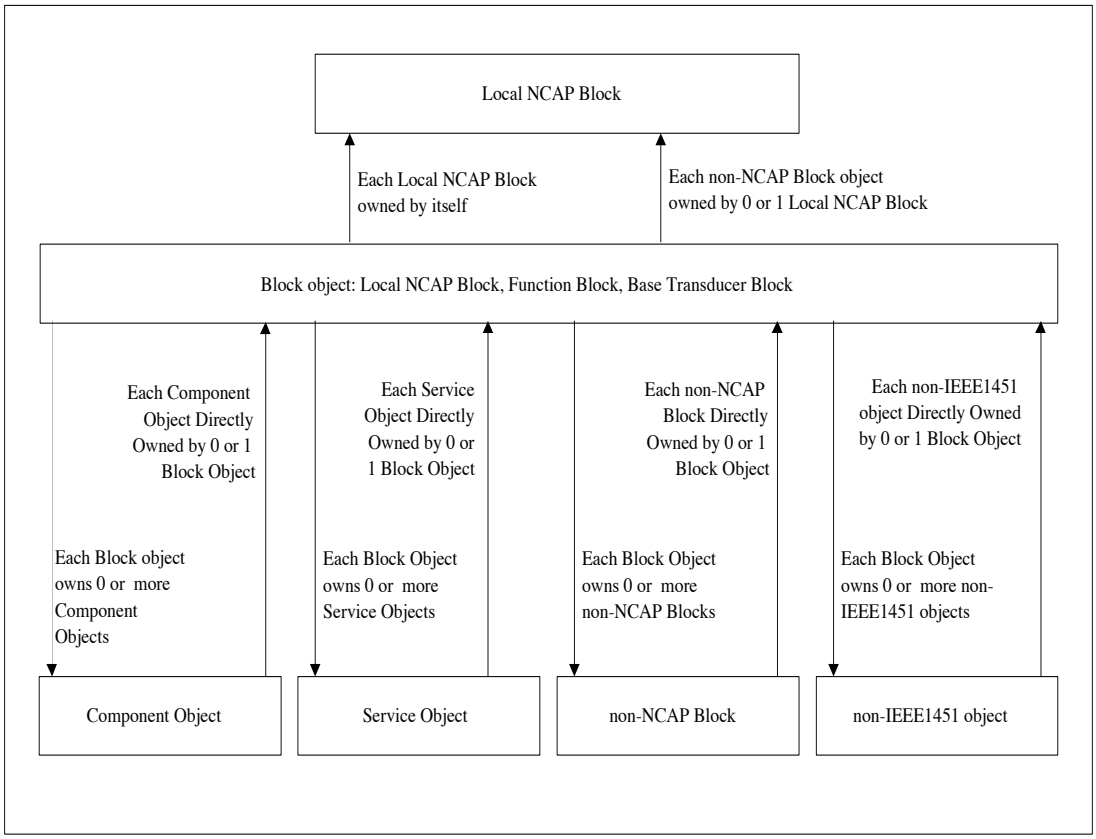

Figure 5—Relationships between Block, Component, Service, and non-IEEE objects

5.1.4 Object identifying properties

An Object’s identifying properties are the Object’s

— Class ID

— Class Name

— Object ID

— Object Tag

Copyright © 2000 IEEE. All rights reserved.

— Object Name

— Object Dispatch Address

NOTES

1—With the exception of Object Dispatch Address, the representations of the listed items are defined by this standard. The representation of an Object Dispatch Address is specific to each underlying communication network and IEEE 1451.1 implementation.

2—With the exception of Object Tag, the properties are “read only” from a remote NCAP.

5.1.4.1 Properties of Class ID

An Object’s Class ID

— Identifies the Object’s class

— Identifies the position in the class hierarchy of the Object’s class

— Helps identify the Object’s semantics

— Is established by the creator of the class

— Cannot be modified

— Is unique for the classes defined in this standard

5.1.4.2 Properties of Class Name

An Object’s Class Name

— Provides a human-readable description of the semantics of the class

— Is established by the creator of the class

— Cannot be modified

— Is unique for the classes specified in this standard

NOTE—For those classes specified by this standard, Class Names are represented using the IEEE 1451.1 character set; see Annex F.

5.1.4.3 Properties of Object ID

An Object’s Object ID

— Is unique within a system. This standard specifies several algorithms for generating values for Object ID that are unique for all systems; see 8.2.3.5.

— Unambiguously distinguishes the Object from any other Object.

— Unless fixed, that is “read only,” is generated by the Object’s Local NCAP Block.

5.1.4.4 Properties of Object Tag

An Object’s Object Tag

— Defines a logical endpoint for the server side of client-server communications

— Is usually assigned by the end user as a part of the system configuration process

— Is preferably unique within a given system

5.1.4.5 Properties of Object Name

An Object’s Object Name

 Copyright © 2000 IEEE. All rights reserved.

— Provides a human-readable semantic description of an instance of a class
— Is usually established by an IEEE 1451.1 component developer or system developer
— Is bound when the Object is created

5.1.4.6 Properties of Object Dispatch Address

An Object's Object Dispatch Address

— Provides an unambiguous network communication reference for the Object
— Shall be unique within an executing system
— Is communication network and IEEE 1451.1 implementation-specific

5.2 Data model specifications

The IEEE 1451.1 data model specifies the type and form of the information communicated across the IEEE 1451.1-specified Object interfaces in both local and remote communications. The model is realized in an implementation of IEEE 1451.1 as a collection of primitive datatypes and a collection of structure datatypes. Listed here are examples of the datatypes.

A complete description of all of the IEEE 1451.1 datatypes is given in Clause 6.

5.2.1 Primitive types

The data model includes the usual variety of primitive datatypes and arrays of these types, including

— A Boolean type
— An octet type
— Integer types
— Floating point types
— String types

5.2.2 Structure types

5.2.2.1 Physical quantity types

The data model has a number of data structures designed for use in measurement and control systems. These datatypes include

— Datatypes for representing time
— Datatypes for representing the value of a physical quantity
— Datatypes for representing the metadata of a physical quantity (e.g., the units of a measurement)

5.2.2.2 Object property types

There are several structure datatypes used to carry information about Objects. These types include

— `ObjectProperties`
— `ClientPortProperties`
— `PublisherInformation`

Copyright © 2000 IEEE. All rights reserved.

5.2.2.3 Network communication related types

There are several IEEE 1451.1 datatypes used in network communications.

The following datatypes are used to direct and filter publications:

— `PublicationTopic`

— `SubscriptionQualifier`

— `PubSubDomain`

A special IEEE 1451.1-specified datatype shall be used to represent application data in all IEEE 1451.1 network communications. This datatype, `Argument`, is a container, which can hold any of the other IEEE 1451.1 types. All application data in a network communication shall be carried in arrays of `Argument`, the `ArgumentArray` datatype.

5.3 Network communication models

The IEEE 1451.1 standard provides two models of network communication between objects in an IEEE 1451.1 system

— A tightly coupled, point-to-point client-server model for one-to-one communications

— A loosely coupled, publish-subscribe model, for one-to-many and many-to-many communications

The communication models define the syntax and the semantics of the software interfaces between application objects and a communications network. The IEEE 1451.1 standard does this by specifying Client, Publisher, and Subscriber Port Objects and the `Perform` operation on Server Objects. These interfaces are independent of the specific network being used.

The IEEE 1451.1 standard does not specify any network transfer syntax or protocol. For each specific network, it is expected that network software suppliers will provide code libraries that contain routines for the calls between the IEEE 1451.1 communication operations and the network. These libraries will include marshaling and demarshaling routines for transforming IEEE 1451.1 datums to and from their on-the-wire format. In this standard, the services provided by such a library are collectively referred to as the network infrastructure.

Although the IEEE 1451.1 communication models are presented only as network communication models, these presentations shall not preclude implementations of IEEE 1451.1 that bypass the network for communications within the same NCAP process.

NOTE—It is expected that many of the details of both client-server and publish-subscribe operations will be hidden from application developers by the network-specific libraries and generic Function Blocks, Components, and Services provided by manufacturers. In particular, Operation IDs and the details of `Execute`, `ExecuteAsynchronous`, and `Perform` may be hidden from application writers dealing primarily with Blocks.

5.3.1 Client-server communication

The client-server communication model is supported by two complementary application-level operations

— `Execute` on client-side Client Port Objects

— `Perform` on all Network Visible, server-side Objects

 Copyright © 2000 IEEE. All rights reserved.

The `Execute` and the `Perform` operations work together to provide a remote-object-operation-invocation style messaging service.

The object organization providing client-server communications is illustrated in Figure 6.

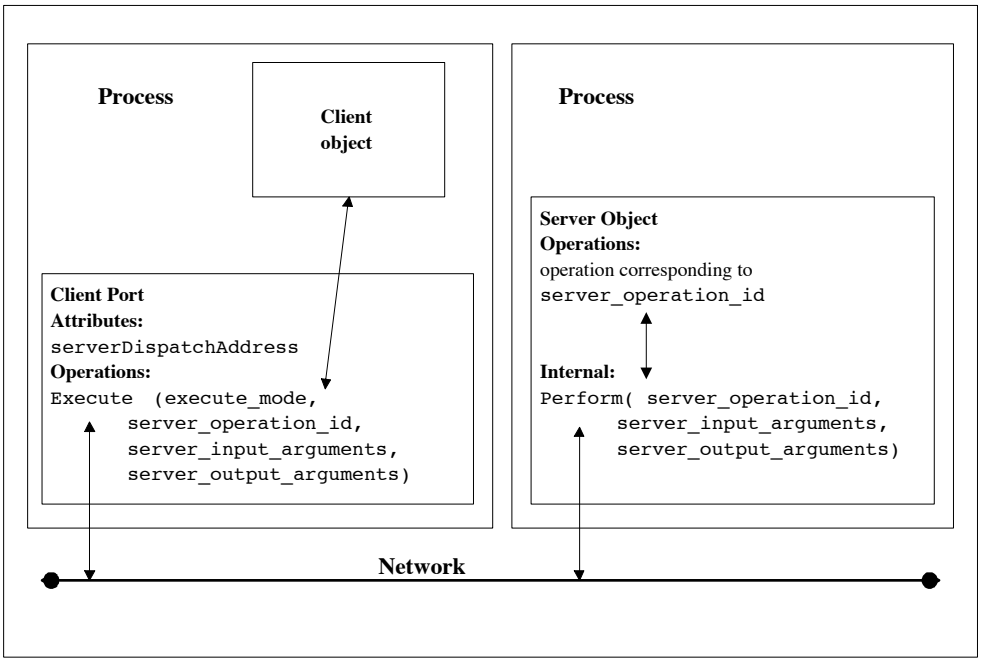

Figure 6—Client-server communication components

The steps required for a Client object to remotely invoke an operation on a Server Object are as follows:

Step 1: During system commissioning, the `serverDispatchAddress` attribute of a Client Port Object in the Client object's process space is bound to the Server Object's Object Dispatch Address value.

Step 2: During system initialization, the Client object is provided with a reference to the Client Port Object.

Step 3: The Client object invokes the `Execute` operation on the Client Port Object, indicating which operation of the remote Server Object is to be executed. It provides that operation's input arguments and references to its output arguments, each encoded into an Argument Array; see 11.4.1.1.

Step 4: Using the network infrastructure, the invocation of the Client Port Object's `Execute` operation results in the invocation of the `Perform` operation on the target Server Object.

Step 5: The `Perform` operation then invokes the desired Server Object operation.

Step 6: When the Server Object's operation completes, it returns to `Perform` with its output arguments bound.

Step 7: The `Perform` operation uses the network infrastructure to return the operation's results to the Client Port Object.

Step 8: The `Execute` operation returns to the invoking Client object with the output Argument Array's Arguments bound.

Copyright © 2000 IEEE. All rights reserved.

NOTES

1—The `Execute` operation on a Client Port Object also supports a "send and forget" messaging mode. In this mode, the operation returns to the invoking Client object as soon as the message is transmitted. Any returns from the remote server operation are ignored.

2—There is a second type of Client Port, the `IEEE1451_AsynchronousClientPort.` This Port type provides an `ExecuteAsynchronous` operation, which returns to the invoking Client object as soon as the message is transmitted. Later, the client can query the Port as to whether the remote server operation has completed and, if so, request from the Port any returned data.

3—For more details on Client-server interactions see Clause 11 and Annex B.

5.3.2 Publish-subscribe communication

The publish-subscribe communication model provides a loosely coupled mechanism for network communications between objects where the sending object, the Publisher object, does not need to be aware of the receiving objects, the Subscriber objects.

The IEEE 1451.1 publish-subscribe communication model is supported by two operations

— `Publish` on Publisher Port Objects
— `AddSubscriber` and an associated callback operation on Subscriber Port Objects

The object organization providing publish-subscribe communications is illustrated in Figure 7.

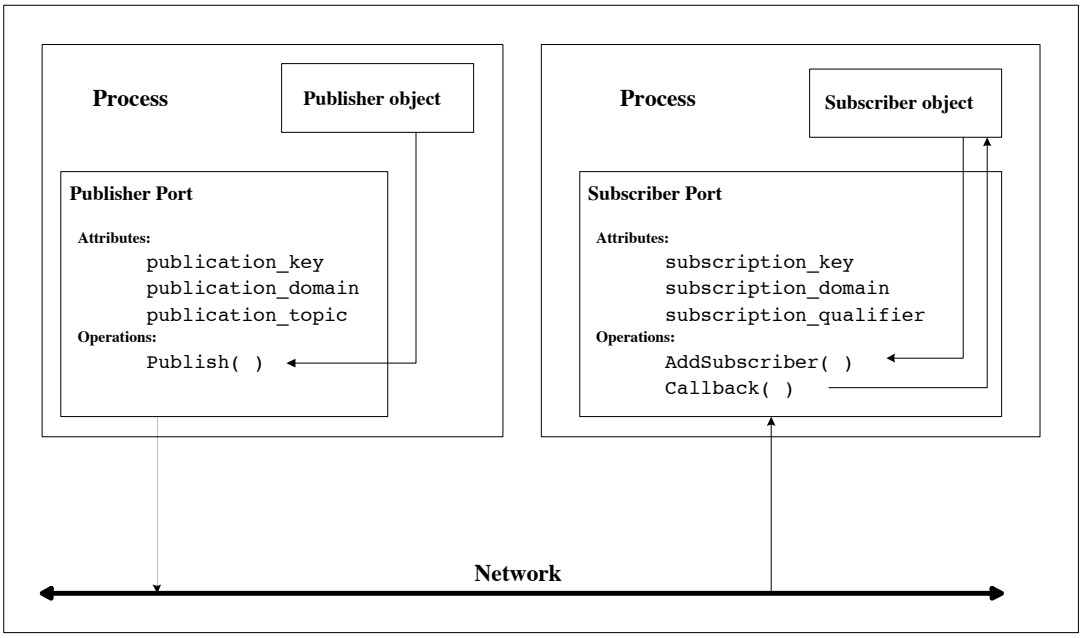

Figure 7—Publish-subscribe communication components

Each IEEE 1451.1 publication carries with it

— A Publication Domain that defines a distribution scope for the publication
— A Publication Key that identifies the application-independent syntax and semantics of the publication from the Publisher's point of view
— A Publication Topic that can be used to identify the application-specific syntax and semantics for the publication's contents

 Copyright © 2000 IEEE. All rights reserved.

A Subscriber Port Object has the attributes

— Subscription Domain

— Subscription Key

— Subscription Qualifier

The combination of Domains, keys, and topics/qualifiers allows only those publications of interest to a Subscriber object to be selected from publications received by a Subscriber Port.

The IEEE 1451.1 publish-subscribe process is as follows:

Step 1: During IEEE 1451.1 system initialization

— The Publication Key attribute of a Publisher Port Object in the Publisher object's process space is bound.

— The Subscription Key attribute of a Subscriber Port Object in the Subscriber object's process space is bound to a value matching the Publication Key of publications of interest.

— The Publisher object is provided with a local reference to the Publisher Port Object.

— The Subscriber object is provided with a local reference to the Subscriber Port object.

Step 2: During system commissioning

— The Publication Domain and Publication Topic attributes of the Publisher Port Object are bound

— The Subscription Domain and Subscription Qualifier attributes of the Subscriber Port Object are bound to values that "match" the values of the Publisher Port Object's Publication Domain and Publication Topic attributes, respectively, so that precisely the desired publications are selected

— The Subscriber object uses the Subscriber Port Object's `AddSubscriber` operation to register with the Port its interest in receiving the publications selected by the Port. The `AddSubscriber` operation requires as an input argument a reference to an operation on the Subscriber object that is invoked by the Port whenever a selected publication is received. This operation is called a subscriber callback operation and is shown in Figure 7.

Step 3: Any time after the system is commissioned

— The Publisher object invokes the `Publish` operation on its associated local Publisher Port passing in as an input argument the publication's contents.

— Using the network infrastructure, the invocation of the `Publish` operation results in the delivery of the publication to all Subscriber Ports in the publication's Domain.

— The receiving Subscriber Ports each use the values of their Subscription Key, Subscription Domain, and Subscription Qualifier attributes to filter the incoming publication.

— If the publication passes a Subscriber Port's filter, the Port invokes all of its registered Subscribers' callback operations, see Figure 7, providing as an input argument the publication's contents. Subscribers register using the `AddSubscriber` operation, 11.10.2.1.

Copyright © 2000 IEEE. All rights reserved.

NOTES

1—IEEE 1451.1 provides other types of publisher ports

— The `IEEE1451_SelfIdentifyingPublisherPort` class (publications contain information about the Publisher)

— The `IEEE1451_EventGeneratorPublisherPort` class (publications have special contents that include a description of when the event occurred)

2—For more details on publish-subscribe interactions, see Clause 11 and Annex C.

5.4 IEEE 1451.1 users

There are three primary categories of users of the IEEE 1451.1 standard. These user categories are referred to throughout the remaining clauses of the standard.

The primary categories of users of the IEEE 1451.1 standard are

— System developers

— Component developers

— End users

5.4.1 System developers

These are primarily manufacturers of

— NCAPs

— NCAP Block classes

— Transducer Block classes

— Other Object classes

— Network-specific infrastructure libraries

5.4.2 Component developers

These are primarily manufacturers of reusable Function Block classes to be used as components in IEEE 1451.1 systems

5.4.3 End users

These are primarily builders or installers of specific end-use Application Systems.

NOTES

1—It is expected that system developers manufacturing NCAPs will also provide software implementing the IEEE 1451.1 standard on those NCAPs. It is expected that these system developers will deliver an `IEEE1451_NCAPBlock` class and one or more variations of the class `IEEE1451_TransducerBlock` all integrated with their hardware. This software will allow application functionality to be realized by application-specific `IEEE1451_FunctionBlock` classes, including the possibility of downloading these Block classes over the network. Likewise, it is expected that component developers will provide reusable `IEEE1451_FunctionBlock` classes implementing common application functions. The standard does specify mechanisms to aid in integrating “off-the-shelf” blocks into an Application System. These mechanisms include the File components and the various registration features of the NCAP Block. The details will be implementation-specific.

2—End users may use components obtained from the system developers and the component developers or, in some cases, develop their own specialization of components provided by developers.

 Copyright © 2000 IEEE. All rights reserved.

6. Datatypes in an IEEE 1451.1 system

For every datatype defined in an IEEE 1451.1 system, there shall be a default value designated. Quantities declared as having a type shall be instantiated with the default value unless otherwise specified in the class specification. The normative definitions of Table 1 are used in this clause.

Table 1—Datatype conventions

Definition	Meaning
IDL: `typedef <datatype> <datatypeArray>[ ];`	An implementation of IEEE 1451.1 Array of the datatype: `<datatype>`.
IDL: `typedef <datatype1> <datatype2>;`	Where `<datatype1>` is a primitive or previously defined IEEE 1451.1 datatype, it shall mean that `<datatype2>` is a mnemonic synonym for `<datatype1>`.

If an IEEE 1451.1 datatype appears in the signature of one or more IEEE 1451.1-specified Network Visible operations, it has its own `TypeCode`. This is the case even when that the datatype is a mnemonic synonym for another datatype. The datatype synonyms are used for semantic clarity in both the IEEE 1451.1 standard and implementations of the standard.

6.1 Primitive datatypes

The datatypes defined in Table 2 shall be defined in all IEEE 1451.1 systems. All other datatypes shall be derived from these primitive types.

The definition of data of these types shall be correctly represented in the local system after such data has been communicated over the network.

6.1.1 Simple primitive types

In any IEEE 1451 implementation, these datatypes shall be mapped to implementation language-dependent primitives.

Table 2—Simple primitive types

Datatype	Default value	Definition
`Boolean`	`FALSE`	`TRUE` or `FALSE`
`Integer8`	0	8-bit signed integer
`UInteger8`	0	8-bit unsigned integer
`Integer16`	0	16-bit signed integer
`UInteger16`	0	16-bit unsigned integer
`Integer32`	0	32-bit signed integer
`UInteger32`	0	32-bit unsigned integer
`Integer64`	0	64-bit signed integer
`UInteger64`	0	64-bit unsigned integer
`Float32`	+0.0	IEEE Std 754-1985 single-precision floating point number [IEEE 754]
`Float64`	+0.0	IEEE Std 754-1985 double-precision floating point number [IEEE 754]
`Octet`	All bits set to 0	8-bit quantity not interpreted as a number

Copyright © 2000 IEEE. All rights reserved.

6.1.2 Type `String` specification

IDL: struct `String`

```
{
   UInteger8 characterSet;
   UInteger8 characterCode;
   UInteger8 language;
   OctetArray stringData;
  };
```

The default value for a variable of type `String` shall be the String with `stringData` of length 0, and the values of `characterSet`, `characterCode`, and `language` all 0.

IDL: `typedef String StringArray[];`

The default value for a `StringArray` shall be the 0-length Array of that type.

6.1.2.1 Member `characterSet` specification

The allowed values of `characterSet` shall be taken from the `StringCharacterSet` enumeration shown in Table 3. The informative column indicates the standard that defines the referenced character set and the common usage. The default value for `characterSet` shall be 0.

IDL: enumeration `StringCharacterSet;`

Table 3—`StringCharacterSet` enumeration

Value	Enumeration	Meaning (informative)
0	SCS_Latin1	ISO 8859-1:1988 Latin characters: set 1
1	SCS_Latin2	ISO 8859-2:1999 Latin characters: set 2
2	SCS_Latin3	ISO 8859-3:1999 Latin characters: set 3
3	SCS_Latin4	ISO/IEC 8859-4:1998 Latin characters: set 4
4	SCS_Cyrillic	ISO 8859-5:1999 Cyrillic characters
5	SCS_Arabic	ISO 8859-6:1999 Arabic characters
6	SCS_Greek	ISO 8859-7:1987 Greek characters
7	SCS_Hebrew	ISO 8859-8:1999 Hebrew characters
8	SCS_Latin5	ISO/IEC 8859-9:1999 Latin characters: set 5
9	SCS_Latin6	ISO/IEC 8859-10:1998 Latin characters: set 6
10	SCS_ASCII	ANSI X3.4:1986 (Reaff 1992) same characters as ASCII, and ISO 646 IRV
11	SCS_10646L1UCS2	ISO 10646-1, level 1:1993 UCS 2 characters
12	SCS_10646L2UCS2	ISO 10646-1, level 2:1993 UCS 2 characters
13	SCS_10646L1UCS4	ISO 10646-1, level 1:1993 UCS 4 characters
14	SCS_10646L2UCS4	ISO 10646-1, level 2:1993 UCS 4 characters
15	SCS_Japanese1978	JIS X0208:1997 Japanese characters circa 1978
16	SCS_Japanese1990	JIS X0212:1990 Japanese supplemental set
17	SCS_Korea	KS X 1001:1992 Korean basic set
18	SCS_PRC	GB 2312-80: 1980 characters of the People's Republic of China
19	SCS_Taiwan	CNS 11643-1992:1992 characters used in Taiwan
20	SCS_Thailand	TIS 620-2529:1990 characters used in Thailand
21	SCS_KOREA_SUP	KS X 1002:1991 Korean supplemental set
22–254		Reserved for future expansion
255	SCS_IEEE1451DOT1	Annex F: characters used for certain Strings in this standard

Copyright © 2000 IEEE. All rights reserved.

6.1.2.2 Member `characterCode` specification

The allowed values of `characterCode` shall be taken from the `StringCharacterCode` enumeration shown in Table 4. The informative column indicates the standard that defines the referenced character code width and the common usage. The default value for `characterCode` shall be 0.

IDL: enumeration `StringCharacterCode;`

Table 4—`StringCharacterCode` enumeration

Value	Enumeration	Meaning (informative)
0	SCC_OCTET1	1 Octet fixed-width
1	SCC_OCTET2	2 Octets fixed-width
2	SCC_OCTET4	4 Octets fixed-width
3	SCC_2022	ISO 2022:1994
4	SCC_SJIS	Shift-JIS (SJIS) Japan
5	SCC_6429	ISO/IEC 6429:1992
6	SCC_UTF8	FSS-UTF AT&T's "File System Safe Universal Transformation Format." See also ISO 10646-1
7	SCC_EUC	ISO 2022:1994 "Extended UNIX Code" (EUC).
8	SCC_2022_JP	ISO 2022:1994 locale specific
9	SCC_2022_JP1	ISO 2022:1994 locale specific
10	SCC_2022_JP2	ISO 2022:1994 locale specific
11	SCC_2022_CN	ISO 2022:1994 locale specific
12	SCC_2022_CN_EXT	ISO 2022:1994 locale specific
13	SCC_2022_KR	ISO 2022:1994 locale specific
14	SCC_EUC_JP	ISO 2022:1994 "Extended UNIX Code" (EUC) local variant Japan
15	SCC_EUC_KR	ISO 2022:1994 "Extended UNIX Code" (EUC) local variant Korea
16	SCC_EUC_CN	ISO 2022:1994 "Extended UNIX Code" (EUC) local variant PRC
17	SCC_EUC_TW	ISO 2022:1994 "Extended UNIX Code" (EUC) local variant Taiwan
18–255		Reserved for future expansion

Unless otherwise specified by the relevant standard, the octet order of a multi-octet character representation, shall be big endian.

NOTE—The term big endian commonly used in computer technology is defined as follows:

> Let C be a byte-addressable computer and d any addressable, multiple-byte integer datum in C's store. For example, d might be a 32-bit unsigned integer or a 16-bit signed integer. The address of d is always the numerically least byte address of the bytes comprising d.
>
> The computer C is said to have a little endian (or right-to-left) architecture if the address of d is the address of d's low-order byte. C is said to have a big endian (or left-to-right) architecture if the address of d is the address of d's high-order byte.

6.1.2.3 Member `language` specification

The allowed values of `language` shall be taken from the `StringLanguage` enumeration. The default value for `language` shall be 0.

Copyright © 2000 IEEE. All rights reserved.

IDL: enumeration `StringLanguage;`

The enumerated value corresponds to the languages of the [ISO 639] standard. Only edition 1988-04-01 of [ISO 639] shall be used. (For the assignment of values for this enumeration, see Annex G).

All languages listed in edition 1988-04-01 of the [ISO 639] standard have enumeration values assigned in Annex G. The enumerated value for languages not listed in Annex G shall not use the reserved enumeration values from 137 to 252.

6.1.2.3.1 Representing languages other than specified by 6.1.2.3

Two values of the enumeration, 253 and 254, are available to users who wish to use languages other than specified by 6.1.2.3. The print behavior of Strings using the values 253 and 254 is outside the scope of this standard.

The enumeration value 255 indicates that the language is the IEEE 1451.1 language defined in Annex F, for use in certain Strings specified by this standard.

6.1.2.4 Member `stringData` specification

The contents of the Octet Array `stringData` shall be the representation of the desired String using character representations defined for the language specified in the `language` field.

6.1.2.5 String semantics

The IEEE1451.1 `String` datatype appears in the signatures of operations for two different purposes

— To provide human-readable information only. Strings used for this purpose never need to be compared, parsed, or otherwise processed for any of the operations or behaviors mandated by this standard. Examples are the String returned by `GetBlockManufacturerID` or `GetBlockModelNumber` on the NCAP Block.

— To provide human-readable information and as an input to a computation such as a comparison. The String returned by the `GetObjectName` operation on any Object is an example. This String is used to convey human-readable information, for example, to a configuration tool. This String is also used as a “key” in the operation `LookupMembersByName` on the Component Group Class. This comparison is done according to the matching algorithm described later in this clause.

Annex F defines a character set, the IEEE 1451.1 character set, that is used to represent

— All Strings defined by this standard representing the name of an Object as returned by `GetObjectName`; see 8.2.1.4.

— All Strings representing the name of a class defined by this standard as returned by `GetClassName`; see 8.1.1.1.

Two Strings shall be identical if, and only if, all of the following are true:

— They have identical `characterSet` values

— They have identical `characterCode` values

— They have identical `language` values

— The lengths of their `stringData` members are identical

 Copyright © 2000 IEEE. All rights reserved.

— An octet-by-octet comparison of their `stringData` members indicates that corresponding octets of the two `stringData` members are identical

where `characterSet`, `characterCode`, `language`, and `stringData` values are the member values of the String datums being compared.

NOTE—Configuration tools may present information about Object Tags, Publication Topics, and Subscription Qualifiers to users; see Annex D. All of these are of underlying type `OctetArray.` There is no print form specified. The only computations defined on these quantities are matching algorithms. The generation of print forms for use in configuration tools is outside the scope of this standard.

6.1.3 Type `ObjectDispatchAddress` specification

IDL: `ObjectDispatchAddress;`

This type is specific to the network used for communication.

6.2 Derived datatypes

The datatypes in this clause are based on the primitive datatypes. The definition of data of these types shall be correctly represented in the local system after the data has been communicated over the network.

Only the primitive or derived datatypes defined in this clause shall be allowed as part of the signature for any IEEE 1451 system operation. Purely local operations, except those specified in this standard, may use any user-defined datatypes, deemed appropriate by the implementer. Data based on such user-defined datatypes shall be encoded into the IEEE 1451.1 primitive or derived datatypes if they are to be made visible over the network.

Unless otherwise stated

— The default properties for a struct type shall be the struct with each member having the default value for the member type.
— The default value for an Array type shall be the 0 length Array of that type.

Unless otherwise stated, when required, the array element with index 0, that is the 0th element, shall be designated the most significant member of the array, the element with index 1 the next most significant member, and so forth.

6.2.1 Specification of Arrays of simple primitive types

Any implementation of this standard shall provide a mechanism by which the number of elements in an IEEE 1451.1 Array can be determined. This number shall be represented as a `UInteger32`. The implementation of the mechanism is outside the scope of this standard.

```
IDL: typedef Boolean     BooleanArray[];

IDL: typedef Integer8    Integer8Array[];

IDL: typedef UInteger8   UInteger8Array[];

IDL: typedef Integer16   Integer16Array[];

IDL: typedef UInteger16  UInteger16Array[];
```

Copyright © 2000 IEEE. All rights reserved.

```
IDL: typedef Integer32   Integer32Array[];

IDL: typedef UInteger32  UInteger32Array[];

IDL: typedef Integer64   Integer64Array[];

IDL: typedef UInteger64  UInteger64Array[];

IDL: typedef Float32     Float32Array[];

IDL: typedef Float64     Float64Array[];

IDL: typedef Octet       OctetArray[];
```

6.2.2 Class identifier type specification

```
IDL: typedef OctetArray ClassID;
```

6.2.3 Object identifier type specifications

```
IDL: typedef OctetArray ObjectTag;
IDL: typedef ObjectTag ObjectTagArray[];
IDL: typedef OctetArray ObjectID;
```

6.2.4 Time type specifications

The `TimeRepresentation` type may be used to specify both

— Timestamps (Time with respect to an epoch)

— Time increments

```
IDL: struct TimeRepresentation

    {
     UInteger32 seconds;
     Integer32 nanoseconds;
    };
```

The range of the absolute value of the `nanoseconds` member shall be restricted to

$$0 \leq \left| \texttt{nanoseconds} \right| < 10^9$$

The sign of the `nanoseconds` member shall be interpreted as the sign of the entire representation.

A negative timestamp shall indicate time prior to the epoch.

A negative increment shall result, for example, from subtracting a timestamp A from a timestamp B, where A is an instant later in time than B.

Example: A timestamp whose `seconds` member value is 10 and whose `nanoseconds` member value is -5×10^8 represents an instant 10.5 seconds before the epoch.

The `TimeInterval` type may be used to specify time intervals with a specific origin.

 Copyright © 2000 IEEE. All rights reserved.

```
IDL: struct TimeInterval
      {
      TimeRepresentation startTime;
      TimeRepresentation deltaTime;
      };
IDL: typedef TimeRepresentation TimeRepresentationArray[];
IDL: typedef TimeInterval TimeIntervalArray[];
```

6.2.5 Bit sequence type specifications

```
IDL: typedef OctetArray BitSequence;
```

The `OctetArray` shall be interpreted on a bit-by-bit basis to represent a sequence of bits. There is no requirement that all bits of each octet be meaningful.

```
IDL: typedef BitSequence BitSequenceArray[];
```

6.2.6 Object properties structure specifications

```
IDL: struct ObjectProperties
      {
      ObjectTag objectTag;
      ObjectTag owningBlockObjectTag;
      ObjectDispatchAddress objectDispatchAddress;
      String objectName;
      UInteger16 blockCookie;
      };
IDL: ObjectProperties ObjectPropertiesArray[];


IDL: struct ClientPortProperties
      {
      ObjectTag clientPortObjectTag;
      String clientPortName;
      ObjectDispatchAddress clientPortDispatchAddress;
      ObjectTag serverObjectTag;
      ObjectDispatchAddress serverDispatchAddress;
      };
IDL: typedef ClientPortProperties ClientPortPropertiesArray[];
```

6.2.7 Return code type specifications

```
IDL: typedef UInteger16 OpReturnCode;
IDL: typedef UInteger32 ClientServerReturnCode;
```

NOTE—The internal structures of the `OpReturnCode` and `ClientServerReturnCode` types are defined in 7.2.3.1.

6.2.8 Publication Topic, Subscription Qualifier and Domain specifications

```
IDL: typedef OctetArray PublicationTopic;
IDL: typedef OctetArray SubscriptionQualifier;
IDL: typedef Octet PubSubDomain[8];
```

Copyright © 2000 IEEE. All rights reserved.

The length of `PubSubDomain` shall be 8 octets. The default value of `PubSubDomain` shall be value 1 for all bits in the Array.

6.2.9 Units type specifications

IEEE 1451.1 units representations shall follow the specification for units in 3.3.8 of [IEEE 1451.2].

```
IDL: typedef Octet Units[10];
```

The length of `Units` shall be 10 octets. The default value for `Units` shall be the value 0 for all bits in the Array.

```
IDL: typedef Units UnitsArray[];
```

6.2.10 Uncertainty type specifications

```
IDL: typedef struct Uncertainty
      {
      UInteger8 interpretation;
      Float32     uncertaintyValue;
      Float32     coverageFactor,
      Float32     customFactor;
      };
IDL: typedef Uncertainty UncertaintyArray[ ];
```

6.2.11 Enumeration type specifications

An enumeration member of an IEEE 1451.1 enumeration type is represented as a `UInteger8`. Unless otherwise noted, the assignment of values to the enumeration members shall start at 0 (zero), for the first member listed of an enumeration, and shall increment by +1 (one), for each succeeding member. Unless otherwise stated, the default value of an `enumeration` member shall be 0.

Any enumeration used in an IEEE 1451.1 system shall have members of the form `<prefix>_<suffix>`. Here, `<prefix>` shall be enumeration-type-specific, shall be the same for all members of the enumeration, and shall be unique in the space of all enumeration prefixes occurring in the system. This requirement shall apply to all IEEE 1451.1-specified enumerations and user-specified enumerations with the exception of the IEEE 1451.1-specified `TypeCode` enumeration. The members of the `TypeCode` enumeration, because of the special role of `TypeCode` as the discriminant type of the `Argument` union data type, have the unique form `<data_type_designator>_TC`.

6.2.11.1 Enumeration semantics

Enumerations defined in this standard always appear in operation signatures or publication formats as `UInteger8` types. For each enumeration, the value of the `UInteger8` has been assigned for each member of the enumeration. Enumerations have no “print form” specified. Implementations shall not include any operational dependency on the particular textual description used in the definition of the enumeration. Any textual description of the semantics associated with the enumeration value shall not appear in the signature of any operation, or in the format of any publication, except for informational purposes.

6.2.11.2 Enumeration summary list

Most enumerations are defined in the clause where they are first used. Table 5 provides a list of all enumerations defined in this standard along with a reference to the defining clause.

 Copyright © 2000 IEEE. All rights reserved.

Table 5—Enumeration summary

Enumeration Name	Reference clause
ActionState	10.10.1.3.1
AsynchronousClientResultStatus	11.5.1.4.2
BlockMajorState	9.1.1.3.1
CalibrationKey	Annex H.4.2.6
ChannelDataModel	Annex H.4.2.5
ChannelGrouping	Annex H.4.2.2
ChannelTypeKey	Annex H.4.2.1
CoordinateSystem	10.7.2.5.3
CorrectionMode	9.5.1.3.1
EventGeneratorState	11.9.1.1.1
ExecuteMode	11.4.1.1.1
FileState	10.11.1.7.1
FunctionBlockState	9.3.1.1.1
MajorReturnCode	7.2.3.2.2
MessagePriority	11.2.1.3.1
MinorReturnCode	7.2.3.2.1
NCAPBlockState	9.2.1.1.1
ParameterBuffering	10.4.1.4.2
ParameterInterpretation	10.4.1.4.1
PartitionedFileSubstate	10.12.1.4.1
PartitionReturn	10.12.1.1
PhysicalParameterType	10.4.1.1.1
PhysicalUnitsInterpretation	Annex H.4.2.4
PublicationContentCode	11.8.3.4.3
PublicationContentsReturn	11.6.2.1
PublisherMetadataReturn	11.8.1.1
PubSubKey	12.3.2
SetClientPortServerObjectBindingsReturn	9.2.1.7
StringCharacterCode	6.1.2.2
StringCharacterSet	6.1.2.1
StringLanguage	6.1.2.3
TimeEpoch	13.2.1.6
TimeType	10.9.1.2.1
TypeCode	6.2.14
UncertaintyInterpretation	7.4.1
UUIDAlgorithmID	8.2.3.5.1
Wildcard	12.3.4

6.2.12 Structure `PublisherInformation` specification

```
IDL: struct PublisherInformation
      {
      ObjectTag portObjectTag;
      UInteger16 publicationID;
      UInteger16 publicationChangeID;
      };
```

Copyright © 2000 IEEE. All rights reserved.

6.2.13 Union type specifications

Any union datatype that occurs either directly or indirectly in the signature of a Network Visible operation shall be a discriminated union with the discriminant being an enumeration. A union datatype U occurs indirectly in the signature of an operation if for some data type in the signature, say T, U occurs as a node in the datatype tree representing the recursive decomposition of T into primitive datatypes.

Any instance of a discriminated union datatype shall satisfy one of the following restrictions:

— The value of the instance shall always be well defined—that is, an attempt to access the union's value shall not fail.

— There shall be a mechanism for detecting that the value is not well defined that may be used prior to attempts to access the union's value.

Attempts by the network infrastructure to marshal a union whose value is not well defined shall result in an error signaled by the return code `MJ_FAILED_MARSHALING` to the client initiating the marshaling.

In the special case of unions with discriminate of datatype `TypeCode`, the `TypeCode` enumeration has a distinguished value, `EMPTY_TC`. If the value of an instance of such a discriminated union is undefined, the value of the discriminant shall be `EMPTY_TC`.

For a discussion of discriminated unions, see [B4].

6.2.14 Type `Argument` specification

The IEEE 1451.1 type `Argument` is declared in the IDL as a discriminated union where the discriminator is a `TypeCode`.

In this standard, the term "value of the Argument" shall be interpreted as follows:

— An `Argument` is modeled as a discriminated union with an associated `TypeCode` as the discriminant.

— The member of the union corresponding to this `TypeCode` has a value represented in the datatype corresponding to this `TypeCode.`

— The "value of the Argument" shall be this member value.

More generally, for any IEEE 1451.1-specified data structure, ***S***, that is declared in the IDL as a discriminated union, the term "value of the union" shall be interpreted as follows:

— The member of the union corresponding to the value of the discriminant has a value represented in the datatype corresponding to this discriminant.

— The value of the union shall be this member value.

There is a `TypeCode` for every distinct IEEE 1451.1 declared nonobject type that can be communicated over the network.

```
IDL: enumeration TypeCode
      {
      EMPTY_TC,
      ARGUMENT_ARRAY_TC,
      BIT_SEQUENCE_TC,
      BIT_-SEQUENCE_ARRAY_TC,
      BOOLEAN_TC,
      BOOLEAN_ARRAY_TC,
      CLASS_ID_TC,
```

 Copyright © 2000 IEEE. All rights reserved.

```
CLIENT_PORT_PROPERTIES_TC,
CLIENT_PORT_PROPERTIES_ARRAY_TC,
CLIENT_SERVER_RETURN_CODE_TC,
FLOAT32_TC,
FLOAT32_ARRAY_TC,
FLOAT64_TC,
FLOAT64_ARRAY_TC,
INTEGER16_TC,
INTEGER16_ARRAY_TC,
INTEGER32_TC,
INTEGER32_ARRAY_TC,
INTEGER64_TC,
INTEGER64_ARRAY_TC,
INTEGER8_TC,
INTEGER8_ARRAY_TC,
OBJECT_DISPATCH_ADDRESS_TC,
OBJECT_ID_TC,
OBJECT_PROPERTIES_TC,
OBJECT_PROPERTIES_ARRAY_TC,
OBJECT_TAG_TC,
OBJECT_TAG_ARRAY_TC,
OCTET_TC,
OCTET_ARRAY_TC,
OP_RETURN_CODE_TC,
PHYSICAL-_PARAMETER_DATA_TC,
PHYSICAL_PARAMETER_DATA_ARRAY_TC,
PHYSICAL_PARAMETER_METADATA_TC,
PHYSICAL_PARAMETER_METADATA_ARRAY_TC,
PUBLICATION_TOPIC_TC,
PUBLISHER_INFORMATION_TC,
PUBSUB_DOMAIN_TC,
STRING_TC,
STRING_ARRAY_TC,
SUBSCRIPTION_QUALIFIER_TC,
TIME_INTERVAL_TC,
TIME_INTERVAL_ARRAY_TC,
TIME_REPRESENTATION_TC,
TIME_REPRESENTATION_ARRAY_TC,
UINTEGER16_TC,
UINTEGER16_ARRAY_TC,
UINTEGER32_TC,
UINTEGER32_ARRAY_TC,
UINTEGER64_TC,
UINTEGER64_ARRAY_TC,
UINTEGER8_TC,
UINTEGER8_ARRAY_TC,
UNCERTAINTY_TC,
UNCERTAINTY_ARRAY_TC,
UNITS_TC,
UNITS_ARRAY_TC
};
```

Copyright © 2000 IEEE. All rights reserved.

```
IDL: union Argument switch (TypeCode)
      {
      case ARGUMENT_ARRAY_TC:
            ArgumentArray argumentArrayVal;
      case BIT_SEQUENCE_TC:
            BitSequence bitSequenceVal;
      case BIT_SEQUENCE_ARRAY_TC:
            BitSequenceArray bitSequenceArrayVal;
      case BOOLEAN_TC:
            Boolean booleanVal;
      case BOOLEAN_ARRAY_TC:
            BooleanArray booleanArrayVal;
      case CLASS_ID_TC:
            ClassID classIDVal;
      case CLIENT_PORT_PROPERTIES_TC:
            ClientPortProperties clientPortPropertiesVal;
      Case CLIENT_PORT_PROPERTIES_ARRAY_TC:
            ClientPortPropertiesArray
                  clientPortPropertiesArrayVal;
      case CLIENT_SERVER_RETURN_CODE_TC:
            ClientServerReturnCode clientServerReturnCodeVal;
      case FLOAT32_TC:
            Float32 float32Val;
      case FLOAT32_ARRAY_TC:
            Float32Array float32ArrayVal;
      case FLOAT64_TC:
            Float64 float64Val;
      case FLOAT64_ARRAY_TC:
            Float64Array float64ArrayVal;
      case INTEGER16_TC:
            Integer16 integer16Val;
      case INTEGER16_ARRAY_TC:
            Integer16Array integer16ArrayVal;
      case INTEGER32_TC:
            Integer32 integer32Val;
      case INTEGER32_ARRAY_TC:
            Integer32Array integer32ArrayVal;
      case INTEGER64_TC:
            Integer64 integer64Val;
      case INTEGER64_ARRAY_TC:
            Integer64Array integer64ArrayVal;
      case INTEGER8_TC:
            Integer8 integer8Val;
      case INTEGER8_ARRAY_TC:
            Integer8Array integer8ArrayVal;
      case OBJECT_DISPATCH_ADDRESS_TC:
            ObjectDispatchAddress objectDispatchAddressVal;
      case OBJECT_ID_TC:
            ObjectID objectIDVal;
      case OBJECT_PROPERTIES_TC:
            ObjectProperties objectPropertiesVal;
      case OBJECT_PROPERTIES_ARRAY_TC:
            ObjectPropertiesArray objectPropertiesArrayVal;
```

 Copyright © 2000 IEEE. All rights reserved.

```
case OBJECT_TAG_TC:
      ObjectTag objectTagVal;
case OBJECT_TAG_ARRAY_TC:
      ObjectTagArray objectTagArrayVal;
case OCTET_TC:
      Octet octetVal;
case OCTET_ARRAY_TC:
      OctetArray octetArrayVal;
case OP_RETURN_CODE_TC:
      OpReturnCode opReturnCodeVal;
case PHYSICAL_PARAMETER_DATA_TC:
      PhysicalParameterData physicalParameterDataVal;
case PHYSICAL_PARAMETER_DATA_ARRAY_TC:
      PhysicalParameterDataArray
            physicalParameterDataArrayVal;
case PHYSICAL_PARAMETER_METADATA_TC:
      PhysicalParameterMetadata
            physicalParameterMetadataVal;
case PHYSICAL_PARAMETER_METADATA_ARRAY_TC:
      PhysicalParameterMetadataArray
            physicalParameterMetadataArrayVal;
case PUBLICATION_TOPIC_TC:
      PublicationTopic publicationTopicVal;
case PUBLISHER_INFORMATION_TC:
      PublisherInformation publisherInformationVal;
case PUBSUB_DOMAIN_TC:
      PubSubDomain pubsubDomainVal;
case STRING_TC:
      String stringVal;
case STRING_ARRAY_TC:
      StringArray stringArrayVal;
case SUBSCRIPTION_QUALIFIER_TC:
      SubscriptionQualifier subscriptionQualifierVal;
case TIME_INTERVAL_TC:
      TimeInterval timeIntervalVal;
case TIME_INTERVAL_ARRAY_TC:
      TimeIntervalArray timeIntervalArrayVal;
case TIME_REPRESENTATION_TC:
      TimeRepresentation timeRepresentationVal;
case TIME_REPRESENTATION_ARRAY_TC:
            TimeRepresentationArray
            timeRepresentationArrayVal;
case UINTEGER16_TC:
      UInteger16 uInteger16Val;
case UINTEGER16_ARRAY_TC:
      UInteger16Array uInteger16ArrayVal;
case UINTEGER32_TC:
      UInteger32 uInteger32Val;
case UINTEGER32_ARRAY_TC:
      UInteger32Array uInteger32ArrayVal;
case UINTEGER64_TC:
      UInteger64 uInteger64Val;
case UINTEGER64_ARRAY_TC:
      UInteger64Array uInteger64ArrayVal;
```

Copyright © 2000 IEEE. All rights reserved.

```
      case UINTEGER8_TC:
            UInteger8 uInteger8Val;
      case UINTEGER8_ARRAY_TC:
            UInteger8Array uInteger8ArrayVal;
      case UNCERTAINTY_TC:
            Uncertainty uncertaintyVal;
      case UNCERTAINTY_ARRAY_TC:
            UncertaintyArray uncertaintyArrayVal;
      case UNITS_TC:
            Units unitsVal;
      case UNITS_ARRAY_TC:
            UnitsArray unitsArrayVal;
            };
```

IDL: `typedef Argument ArgumentArray[];`

6.2.15 Physical Parameter related structure specifications

```
IDL: struct PhysicalParameterSingletonData
      {
      ArgumentArray value;
      TimeRepresentation timestamp;
      };

IDL: struct PhysicalParameterSeriesData
      {
      ArgumentArray value;
      TimeRepresentation timestamp;
      Argument abscissaIncrement;
      Argument abscissaOrigin;
      };

IDL: union PhysicalParameterData switch (PhysicalParameterType)
      {
      case PP_SCALAR_ANALOG:
            PhysicalParameterSingletonData singletonData;
      case PP_SCALAR_DISCRETE:
            PhysicalParameterSingletonData singletonData;
      case PP_SCALAR_DIGITAL:
            PhysicalParameterSingletonData singletonData;
      case PP_SCALAR_ANALOG_SERIES:
            PhysicalParameterSeriesData seriesData;
      case PP_SCALAR_DISCRETE_SERIES:
            PhysicalParameterSeriesData seriesData;
      case PP_SCALAR_DIGITAL_SERIES:
            PhysicalParameterSeriesData seriesData;
      case PP_VECTOR_ANALOG:
            PhysicalParameterSingletonData singletonData;
      case PP_VECTOR_DISCRETE:
            PhysicalParameterSingletonData singletonData;
      case PP_VECTOR_DIGITAL:
            PhysicalParameterSingletonData singletonData;
      case PP_VECTOR_ANALOG_SERIES:
            PhysicalParameterSeriesData seriesData;
```

 Copyright © 2000 IEEE. All rights reserved.

```
      case PP_VECTOR_DISCRETE_SERIES:
            PhysicalParameterSeriesData seriesData;
      case PP_VECTOR_DIGITAL_SERIES:
            PhysicalParameterSeriesData seriesData;
      };

IDL: typedef PhysicalParameterData PhysicalParameterDataArray[];

IDL: struct ScalarAnalogMetadata
      {
      String parameterName;
      UInteger8 parameterInterpretation;
      UInteger8 buffering;
      UInteger8 datatype;
      Units units;
      Argument upperLimit;
      Argument lowerLimit;
      Uncertainty uncertainty;
      };

IDL: struct ScalarDigitalMetadata
      {
      String parameterName;
      UInteger8 parameterInterpretation;
      UInteger8 buffering;
      UInteger8 datatype;
      Units units;
      UInteger16 numberOfOctets;
      UInteger16 numberOfSignificantBits;
      Boolean rightJustifiedFlag;
      };

IDL: struct ScalarDiscreteMetadata
      {
      String parameterName;
      UInteger8 parameterInterpretation;
      UInteger8 buffering;
      UInteger8 datatype;
      Units units;
      Argument upperLimit;
      Argument lowerLimit;
      };

IDL: struct ScalarAnalogSeriesMetadata
      {
      String parameterName;
      UInteger8 parameterInterpretation;
      UInteger8 buffering;
      UInteger8 datatype;
      Units units;
      Argument upperLimit;
      Argument lowerLimit;
      Units abscissaUnits;
      Argument abscissaIncrement;
```

Copyright © 2000 IEEE. All rights reserved.

```
      Argument abscissaOrigin;
      Uncertainty abscissaIncrementUncertainty;
      Uncertainty abscissaOriginUncertainty;
      Uncertainty uncertainty;
      };

IDL: struct ScalarDigitalSeriesMetadata
      {
      String parameterName;
      UInteger8 parameterInterpretation;
      UInteger8 buffering;
      UInteger8 datatype;
      Units units;
      Units abscissaUnits;
      Argument abscissaIncrement;
      Argument abscissaOrigin;
      Uncertainty abscissaIncrementUncertainty;
      Uncertainty abscissaOriginUncertainty;
      UInteger16 numberOfOctets;
      UInteger16 numberOfSignificantBits;
      Boolean rightJustifiedFlag;
      };

IDL: struct ScalarDiscreteSeriesMetadata
      {
      String parameterName;
      UInteger8 parameterInterpretation;
      UInteger8 buffering;
      UInteger8 datatype;
      Units units;
      Argument upperLimit;
      Argument lowerLimit;
      Units abscissaUnits;
      Argument abscissaIncrement;
      Argument abscissaOrigin;
      Uncertainty abscissaIncrementUncertainty;
      Uncertainty abscissaOriginUncertainty;
      };

IDL: struct VectorAnalogMetadata
      {
      String parameterName;
      UInteger8 parameterInterpretation;
      UInteger8 buffering;
      UInteger16 dimension;
      CoordinateSystem coordinateSystem;
      UInteger8Array datatype;
      UnitsArray units;
      ArgumentArray upperLimit;
      ArgumentArray lowerLimit;
      UncertaintyArray uncertainty;
      };
```

 Copyright © 2000 IEEE. All rights reserved.

```
IDL: struct VectorDigitalMetadata
      {
      String parameterName;
      UInteger8 parameterInterpretation;
      UInteger8 buffering;
      UInteger16 dimension;
      CoordinateSystem coordinateSystem;
      UInteger8Array datatype;
      UnitsArray units;
      UInteger16 Array numberOfOctets;
      UInteger16 Array numberOfSignificantBits;
      Boolean Array rightJustifiedFlag;
      };

IDL: struct VectorDiscreteMetadata
      {
      String parameterName;
      UInteger8 parameterInterpretation;
      UInteger8 buffering;
      UInteger16 dimension;
      CoordinateSystem coordinateSystem;
      UInteger8Array datatype;
      UnitsArray units;
      ArgumentArray upperLimit;
      ArgumentArray lowerLimit;
      };

IDL: struct VectorAnalogSeriesMetadata
      {
      String parameterName;
      UInteger8 parameterInterpretation;
      UInteger8 buffering;
      UInteger16 dimension;
      CoordinateSystem coordinateSystem;
      Units abscissaUnits;
      Argument abscissaIncrement;
      Argument abscissaOrigin;
      Uncertainty abscissaIncrementUncertainty;
      Uncertainty abscissaOriginUncertainty;
      UInteger8Array datatype;
      UnitsArray units;
      ArgumentArray upperLimit;
      ArgumentArray lowerLimit;
      UncertaintyArray uncertainty;
      };

IDL: struct VectorDigitalSeriesMetadata
      {
      String parameterName;
      UInteger8 parameterInterpretation;
      UInteger8 buffering;
      UInteger16 dimension;
      CoordinateSystem coordinateSystem;
      Units abscissaUnits;
```

Copyright © 2000 IEEE. All rights reserved.

```
      Argument abscissaIncrement;
      Argument abscissaOrigin;
      Uncertainty abscissaIncrementUncertainty;
      Uncertainty abscissaOriginUncertainty;
      UInteger8Array datatype;
      UnitsArray units;
      UInteger16Array numberOfOctets;
      UInteger16Array numberOfSignificantBits;
      BooleanArray rightJustifiedFlag;
      };

IDL: struct VectorDiscreteSeriesMetadata
      {
      String parameterName;
      UInteger8 parameterInterpretation;
      UInteger8 buffering;
      UInteger16 dimension;
      CoordinateSystem coordinateSystem;
      Units abscissaUnits;
      Argument abscissaIncrement;
      Argument abscissaOrigin;
      Uncertainty abscissaIncrementUncertainty;
      Uncertainty abscissaOriginUncertainty;
      UInteger8Array datatype;
      UnitsArray units;
      ArgumentArray upperLimit;
      ArgumentArray lowerLimit;
      };

IDL: union PhysicalParameterMetadata switch
            (PhysicalParameterType)
      {
      case PP_SCALAR_ANALOG:
            ScalarAnalogMetadata scalarAnalogMetadata;
      case PP_SCALAR_DISCRETE:
            ScalarDiscreteMetadata scalarDiscreteMetadata;
      case PP_SCALAR_DIGITAL:
            ScalarDigitalMetadata scalarDigitalMetadata;
      case PP_SCALAR_ANALOG_SERIES:
            ScalarAnalogSeriesMetadata
                  scalarAnalogSeriesMetadata;
      case PP_SCALAR_DISCRETE_SERIES:
            ScalarDiscreteSeriesMetadata
                  scalarDiscreteSeriesMetadata;
      case PP_SCALAR_DIGITAL_SERIES:
            ScalarDigitalSeriesMetadata
                  scalarDigitalSeriesMetadata;
      case PP_VECTOR_ANALOG:
            VectorAnalogMetadata vectorAnalogMetadata;
      case PP_VECTOR_DISCRETE:
            VectorDiscreteMetadata vectorDiscreteMetadata;
      case PP_VECTOR_DIGITAL:
            VectorDigitalMetadata vectorDigitalMetadata;
```

 Copyright © 2000 IEEE. All rights reserved.

```
      case PP_VECTOR_ANALOG_SERIES:
            VectorAnalogSeriesMetadata
                  VectorAnalogSeriesMetadata;
      case PP_VECTOR_DISCRETE_SERIES:
            VectorDiscreteSeriesMetadata
                  vectorDiscreteSeriesMetadata;
      case PP_VECTOR_DIGITAL_SERIES:
            VectorDigitalSeriesMetadata
                  vectorDigitalSeriesMetadata;
      };

IDL: typedef PhysicalParameterMetadata
            PhysicalParameterMetadataArray[];
```

7. Common properties

This clause specifies common properties for

- Class designators
- Object operation properties
- Expression of uncertainty in a value

7.1 Class designator properties

This clause defines the header format used in the specifications of class definitions in Clause 8 through Clause 11.

7.1.1 Class header format

Abstract Class (if required for the class definition)

Class: `IEEE1451_ClassDescriptor`

Parent Class: `IEEE1451_ParentClassDescriptor`

Class ID: `classID`

These terms are defined in Table 6.

7.1.2 Specification of Class IDs

The value of the Class ID

- Shall be of type `ClassID`, and therefore represented as Octet Arrays; see 6.2.2
- Shall encode the position of a class in the class hierarchy
- Shall consist of an ordered listing of identifiers, which constitute the path in the class tree to a particular class
- Shall be ordered such that the most significant, that is the 0th, identifier of the listing is the root of the class tree

Copyright © 2000 IEEE. All rights reserved.

Table 6—Class header terms

Format	Definition
Abstract Class	A class designated as an "Abstract Class" shall serve as a specification of commonality applicable to all subclasses. Abstract Classes shall not be instantiated. Only subclasses of Abstract Classes can be instantiated.
Class: `IEEE1451_ClassDescriptor`	The value of: `IEEE1451_ClassDescriptor` shall be the formal name for the class.
Parent Class: `IEEE1451_ParentClass Descriptor`	The value of: `IEEE1451_ParentClassDescriptor` shall be the `IEEE1451_ClassDescriptor` of the immediate parent class in the hierarchy from which this class is subclassed.
Class ID: `classID`	The value of `classID` shall identify the position in the IEEE 1451.1 class structure. The position in the class hierarchy implicitly defines the complete specification of an Object. Class IDs specified in this standard are unique.

7.1.2.1 Rules for Class ID identifiers

For the assignment of identifiers and for the definition of additional classes, over and above those defined in this standard, the following rules shall be followed:

— Each identifier shall be represented by a single octet, which shall be interpreted as an `UInteger8.`

— Assignment of values for the root identifier shall be restricted to specification by this standard and any subsequent balloted amendments, that is, by IEEE 1451.1 authority.

— For each subsequent identifier, (that is at any node in the class hierarchy tree), the values 0–127 shall be reserved for assignment by IEEE 1451.1 authority. (i.e., when any class is considered for refinement into subclasses, the first 128 subclasses of it shall be reserved for IEEE 1451.1 standards activity).

— All values of an identifier greater than 127 are not specified by IEEE 1451.1, but may be used by industry groups or other developers of subclasses of the classes defined by this standard.

— Manufacturers of subclasses shall assign Class ID values to their subclasses that are unique within the scope of the specific manufacturer.

— It is recommended that system integrators or developers of specific installations ensure that the Class IDs of objects in the system are unique.

Example: The Class ID with Octet Array {1,3,45} represents the identifier list 1.3.45. The list is of the form W.X.Y, where W, X, and Y are successive unsigned elements from the Octet Array interpreted as `UInteger8` numbers. The numbers denote the class, subclass, and sub-subclass identifier fields, respectively.

Two Class ID instances shall be the same if, and only if

— The number of fields is the same for both, **AND**

— All corresponding fields have the same value

 Copyright © 2000 IEEE. All rights reserved.

7.1.2.2 Subclass determination algorithm

Given two classes, A and B, with Class IDs class_IDA and class_IDB, class A is a proper subclass of class B if, and only if

— The number of fields in class_IDB is less than the number of fields in class_IDA,

AND

— All fields of class_IDB have the same value as the corresponding fields of class_IDA

7.1.2.3 Purpose of Class IDs

The primary reasons for the hierarchically structured Class IDs are

— To provide a principled way of identifying the heritage of specializations of the IEEE 1451.1-specified classes.

— To provide a way for an object to identify its class to itself and to other objects.

For example, a server object operation may want to restrict client access to clients that are of a given class or a subclass of that class. By requiring a client's Class ID as an argument to that operation, the operation could check if a client was an acceptable client. Class names would not work for this.

7.1.3 Inherited class definition properties

For classes defined in this standard, the following shall all be inherited by the subclass definition from the parent class and grandparent class definitions up to and including the `IEEE1451_Root` class:

— All operations, both Network Visible and Local

— All publications and subscriptions

— All state machines and other material defining the class's behavior

— All required owned Objects

7.2 Object operation properties

All operation definitions include

— Implementation requirements

— Designators for the operation

— The signature of the operation

— The behavior of the operation

7.2.1 Specification of implementation requirements

All operations are indicated in the class summary for each class definition as being either

— Mandatory (m)

— Optional (o)

7.2.1.1 Mandatory (m): definition

The term Mandatory means that the referenced Network Visible functionality associated with the operation shall be implemented as specified. The major return code `MJ_NOP_OPERATION` shall not be returned for a mandatory operation. This option is designated as a Full Implementation.

Copyright © 2000 IEEE. All rights reserved.

7.2.1.2 Optional (o): definition

The term Optional shall designate one of two implementations of the operation

— Interface Only Implementation

— Full Implementation

7.2.1.2.1 Interface Only Implementation: definition

An implementation is defined to be an Interface Only Implementation if the designated operation has no meaning for the implementation of the class (i.e., a write on a read-only object). The invocation of the operation shall not cause any change of state of the object and shall return immediately with a major return code of: `MJ_NOP_OPERATION`. This option is designated as an Interface Only Implementation.

7.2.1.2.2 Full Implementation: definition

An implementation is defined to be a Full Implementation if the designated operation has meaning for the implementation of the class. The result of an invocation of the operation shall be exactly as required for a mandatory operation. Class specifications may include circumstances for which implementers shall fully implement the operations as defined by this clause. This option is designated as a Full Implementation.

7.2.2 Specification of designators for an operation

All operations defined in this standard are designated by an Operation Name and an Operation ID. These designators are listed in the class summary clause of each class definition.

All operations specified in this standard are invoked locally.

References to operations defined in this standard or otherwise defined can appear in the signatures of operations defined in this standard. These references appear only in the signatures of Local operations and appear only as a single “in” or “out” argument. If the specific NCAP implementation requires that references consist of more than a single piece of information, then the “in” or “out” argument of the signature shall be construed as having the appropriate structure to represent the several pieces of reference information.

Class definitions that refer to objects and operations in the Local environment as part of an operation signature denote these references as follows:

— `<local object reference>`

— `<local operation reference>`

The specification of these references shall have the indicated interpretation expressed in the local language and environment, and will be written as follows:

7.2.2.1 `<local object reference>`: definition

This shall be a local reference that allows unambiguous identification of an instance of an IEEE 1451.1 or a non-IEEE 1451.1 object. The notation `<local object reference>[]` shall be used to denote an Array of local object references.

7.2.2.2 `<local operation reference>`: definition

This reference shall provide sufficient information to allow the invocation of any local operation.

 Copyright © 2000 IEEE. All rights reserved.

Depending on the language and implementation this reference may include a `<local object reference>` and an Operation ID or may be as simple as a local function pointer.

7.2.2.3 Operation Name: definition

The Operation Name shall be a lexical reference to the name of the operation that shall be unique within the class defining the operation. Within a class, the Operation Name and the Operation ID shall be in a one-to-one correspondence.

7.2.2.4 Operation ID: definition

The Operation ID shall provide a reference to the operation that shall be unique within the class defining the operation.

An Operation ID shall be of type `UInteger16`.

The underlying network implementations shall preserve this identifier across the network. An implementation may choose to represent these identifiers as an enumeration for clarity in code development. Enumerations of these identifiers are outside the scope of this standard.

The Operation ID allowed values shall be based on the position in the class hierarchy as defined in Table 7.

Table 7—Operation ID values restrictions

Level	Values reserved for IEEE 1451 standard	Values available to industry groups and application developers
0 (that is Root class)	0–2047	none
1	2048–3071	3072–4095
2	4096–5119	5120–6143
3	6144–7167	7168–8191
4	8192–9215	9216–10239
5	10240–11263	11264–12287
6	12288–13311	13312–14335
7	14336–15359	15360–16383
8	16384–17407	17408–18431
9	18432–19455	19456–20479
10	20480–21503	21504– 22527
11	22528–23551	23552–24575
12	24576–25599	25600–26623
12 < N ≤ 31	2048N through 2048N + 1023	2048N +1024 through 2048N + 2047

The Operation ID assigned to operations in classes specified in this standards document are unique across all classes in addition to conforming to the values allowed by Table 7. There is no requirement that user or industry-defined subclasses assign values to the Operation ID for operations that are unique across classes.

Copyright © 2000 IEEE. All rights reserved.

7.2.3 Operation signature specification

All operations are specified by using IDL syntax. Operation signatures shall define the syntax of the operation by specifying

— Return codes
— Operation Name
— Arguments
— Argument order
— Formal names for arguments

Within a class definition, several operations may share the same formal name for operation arguments. If the argument specifications for these arguments with shared names are identical, they will be given at the first appearance of the formal name and not repeated throughout the class definition.

7.2.3.1 Types of return codes for operations

There are two types of return codes defined in this standard

— `OpReturnCode`
— `ClientServerReturnCode`

7.2.3.1.1 Specification of `OpReturnCode`

The `OpReturnCode` is used by most operations. The underlying type is `UInteger16`; see 6.2.7. An `OpReturnCode` shall be expressed as a sequence of two fields

— Minor Field
— Major Field

The Minor Field shall be the high-order 8 bits, interpreted as a `UInteger8`, of the `OpReturnCode` underlying type `UInteger16`.

The Major Field shall be the low-order 8 bits, interpreted as an `UInteger8`, of the `OpReturnCode` underlying type `UInteger16`.

Example: An `OpReturnCode` value of 259 would have

— A Major Field = 3
— A Minor Field = 1

The Major Field shall represent system-level return-status information while the Minor Field is operation-specific.

7.2.3.1.2 Specification of `ClientServerReturnCode`

A `ClientServerReturnCode` is used by the operations

— `Execute`
— `GetResult`
— `Perform`

 Copyright © 2000 IEEE. All rights reserved.

The underlying type is `UInteger32`; see 6.2.7.

A `ClientServerReturnCode` shall be expressed as a sequence of fields defined in Table 8.

Table 8—`ClientServerReturnCode` fields

Field Name	Bits in Client-Server Return Code	Information
`portCode`	High-order 8 bits interpreted as an `UInteger8.`	The return code of the client-side Port Object `Execute` or `GetResult` operation. Values shall be selected from the `MajorReturnCode` enumeration.
`performCode`	Next 8 bits interpreted as an `UInteger8.`	The return code of the server-side Object `Perform` operation. Values shall be selected from the `MajorReturnCode` enumeration.
`operationMinorCode`	Next 8 bits interpreted as an `UInteger8.`	The Minor Field of the `OpReturnCode` of the operation invoked on the Server Object.
`operationMajorCode`	Lower-order 8 bits interpreted as an `UInteger8.`	The Major Field of the `OpReturnCode` of the operation invoked on the Server Object.

Example: A `ClientServerReturnCode` value of 67 240 195, that is

16 777 216 × 4 + 65 536 × 2 + 256 × 1 +1 × 3, would have the following field values:

— `portCode = 4`
— `performCode = 2`
— `operationMinorCode = 1`
— `operationMajorCode = 3`

7.2.3.2 Return Code enumeration specifications

Return code enumerations are specified for

— Minor Field
— Major Field

7.2.3.2.1 Minor Field enumeration specification

The Minor Field shall be the enumeration defined in Table 9.

IDL: enumeration `MinorReturnCode;`

— A Minor Field return other than `MI_NO_ADDITIONAL_INFORMATION,` shall be indicated by a Major Field return other than `MJ_COMPLETE.`

— A Major Field return other than `MJ_COMPLETE,` may be accompanied by a Minor Field return other than `MI_NO_ADDITIONAL_INFORMATION.`

— Classes may specify additional members of the Minor Field return code for specific operations. In those cases, the Minor Field is defined by a new enumeration. The meaning of the first six members of the new enumeration shall correspond to the first six members of the `MinorReturnCode` enumeration.

Copyright © 2000 IEEE. All rights reserved.

Table 9—`MinorReturnCode` values

Name	Value	Meaning
`MI_NO_ADDITIONAL_INFORMATION`	0	No additional information available.
`MI_MEMORY_ALLOCATION_ERROR`	1	Insufficient memory.
`MI_MISSING_INPUT`	2	An input argument was not provided.
`MI_INVALID_TYPE`	3	The argument type is not valid.
`MI_INVALID_VALUE`	4	The value is not valid. For example: out of range.
`MI_COMPUTATION_ERROR`	5	Math errors and exceptions.
Reserved	6–127	
User-defined	128–255	Operation-specific.

7.2.3.2.2 Major Field enumeration specification

The Major Field shall be an enumeration of the status of the operation invocation. Permitted values shall be selected from the `MajorReturnCode` enumeration defined in Table 10.

IDL: enumeration `MajorReturnCode;`

Table 10—`MajorReturnCode` values

Name	Value
`MJ_COMPLETE`	0
`MJ_NOP_OPERATION`	1
`MJ_NO_RETURN_CODE`	2
`MJ_FAILED_NON_SPECIFIC`	3
`MJ_COMMUNICATION_ERROR`	4
`MJ_BUSY`	5
`MJ_SERVICE_UNAVAILABLE`	6
`MJ_ILLEGAL_OPERATION`	7
`MJ_FAILED_INPUT_ARGUMENT`	8
`MJ_FAILED_OUTPUT_ARGUMENT`	9
`MJ_FAILED_MARSHALING`	10
`MJ_FAILED_DEMARSHALING`	11
`MJ_ILLEGAL_TRANSACTION`	12
`MJ_OPERATION_INTERRUPTED`	13
`MJ_OPERATION_TIMEOUT`	14
`MJ_INSUFFICIENT_RESOURCES`	15
`MJ_BLOCK_COOKIE_MISMATCH`	16
`MJ_TRANSDUCER_ERROR`	17
`MJ_SUSPECT_INVOCATION_RESULT`	18
`MJ_NOT_PROPERLY_CONFIGURED`	19
Reserved	20–127
User-defined	128–255

 Copyright © 2000 IEEE. All rights reserved.

7.2.3.3 Restrictions on return codes

Unless otherwise stated, the allowed value and meaning of the Major Field of an `OpReturnCode` or the `portCode`, `performCode`, and `operationMajorCode` fields of a `ClientServerReturnCode` shall be in accordance with 7.2.3.3.1, 7.2.3.3.2, 7.2.3.3.3, and 7.2.3.3.4. Values marked N/A shall not be used in the indicated field.

7.2.3.3.1 Restrictions on allowed `portCode` field values

Errors may be detected in the following Base Client Port operations:

— `Execute`; see 11.4.2.3

— `GetResult`; see 11.5.2.7.1

These errors shall be indicated by an entry in the `portCode` field. The allowed values shall be selected from Table 11.

Table 11—Allowed `portCode` values

Name	Meaning
MJ_COMPLETE	No Base Client Port errors detected.
MJ_NOP_OPERATION	The Base Client Port operation was implemented with an Interface Only Implementation.
MJ_NO_RETURN_CODE	N/A
MJ_FAILED_NON_SPECIFIC	Nonspecific failure. Shall not be used if any of the subsequent enumerations apply.
MJ_COMMUNICATION_ERROR	No response from any Object at the Object Dispatch Address targeted by this Base Client Port.
MJ_BUSY	Requested operation is legal, but cannot be executed, since the Base Client Port is busy, and cannot support multiple requests.
MJ_SERVICE_UNAVAILABLE	The requested Base Client Port operation is not currently available because of some aspect of its internal state; for example, the Port is being configured.
MJ_ILLEGAL_OPERATION	Requested operation is illegal on the Base Client Port.
MJ_FAILED_INPUT_ARGUMENT	One or more operation input arguments are invalid or missing, for `Execute` or `GetResult`.
MJ_FAILED_OUTPUT_ARGUMENT	One or more operation output arguments are invalid or missing, for `Execute` or `GetResult`.
MJ_FAILED_MARSHALING	A failure to marshal to on-the-wire format.
MJ_FAILED_DEMARSHALING	A failure to demarshal from on-the-wire-format.
MJ_ILLEGAL_TRANSACTION	For `Execute`: N/A For `GetResult`: There is no such active Asynchronous Client Port transaction as indicated in the call.
MJ_OPERATION_INTERRUPTED	The operation did not complete because of errors other than a Client Port timeout expiration.
MJ_OPERATION_TIMEOUT	The indicated operation did not complete before a timeout period of the Base Client Port expired.
MJ_INSUFFICIENT_RESOURCES	Requested operation is legal but cannot be executed because of lack of resources available to the Base Client Port.
MJ_BLOCK_COOKIE_MISMATCH	N/A
MJ_TRANSDUCER_ERROR	N/A
MJ_SUSPECT_INVOCATION_RESULT	N/A
MJ_NOT_PROPERLY_CONFIGURED	The Base Client Port is not configured or is improperly configured.
Reserved	Reserved for future use by the IEEE 1451.1 standard committee.
User-defined	May be used by industry groups or implementers.

Copyright © 2000 IEEE. All rights reserved.

7.2.3.3.2 Restrictions on allowed `performCode` field values

Errors may be detected in the `Perform` operation on a server. These errors shall be indicated by an entry in the `performCode` field of the following operation's `ClientServerReturnCode`:

— `Perform`, see 8.2.2.1, on the Server Object, AND

— `Execute`, see 11.4.2.3, on a Client Port, OR

— `GetResult`, see 11.5.2.7.1, on an Asynchronous Client Port

The allowed values for the `performCode` field shall be selected from Table 12.

Table 12—Allowed `performCode` values

Name	Meaning
MJ_COMPLETE	`Perform` did not detect any error during execution.
MJ_NOP_OPERATION	N/A
MJ_NO_RETURN_CODE	No return code is available.
MJ_FAILED_NON_SPECIFIC	Nonspecific failure. Shall not be used if any of the subsequent enumerations apply.
MJ_COMMUNICATION_ERROR	N/A
MJ_BUSY	`Perform` cannot be executed since the server is busy and cannot support multiple requests.
MJ_SERVICE_UNAVAILABLE	`Perform` is currently not available because of some aspect of the server's internal state.
MJ_ILLEGAL_OPERATION	The requested Operation ID is not valid on the target Server Object.
MJ_FAILED_INPUT_ARGUMENT	One or more input arguments are invalid or missing for `Perform.`
MJ_FAILED_OUTPUT_ARGUMENT	One or more output arguments are invalid or missing for `Perform.`
MJ_FAILED_MARSHALING	A failure to marshal to on-the-wire format.
MJ_FAILED_DEMARSHALING	A failure to demarshal from on-the-wire-format.
MJ_ILLEGAL_TRANSACTION	N/A
MJ_OPERATION_INTERRUPTED	`Perform` did not complete.
MJ_OPERATION_TIMEOUT	N/A
MJ_INSUFFICIENT_RESOURCES	`Perform` cannot be executed because of lack of resources in the target Server Object.
MJ_BLOCK_COOKIE_MISMATCH	A Block Cookie mismatch has been detected.
MJ_TRANSDUCER_ERROR	N/A
MJ_SUSPECT_INVOCATION_RESULT	N/A
MJ_NOT_PROPERLY_CONFIGURED	N/A
Reserved	Reserved for future use by the IEEE 1451.1 standard committee.
User-defined	May be used by industry groups or implementers.

7.2.3.3.3 Restrictions on allowed Major Field and `operationMajorCode` field values

Errors may be detected in the execution of the invoked server operation. These errors shall be indicated by an entry in the Major Field of the `OpReturnCode` for the invoked operation. These entries are returned to the client via the `operationMajorCode` field of the following operation's `ClientServerReturnCode`:

— `Perform`, see 8.2.2.1, on the server, AND

 Copyright © 2000 IEEE. All rights reserved.

— `Execute`, see 11.4.2.3, on a Client Port, OR

— `GetResult`, see 11.5.2.7.1, on an Asynchronous Client Port

The allowed values for these entries shall be selected from Table 13.

Table 13—Allowed `operationMajorCode` values

Name	Meaning
MJ_COMPLETE	The operation did not detect any error during execution.
MJ_NOP_OPERATION	The operation has an Interface Only Implementation.
MJ_NO_RETURN_CODE	No return code is available (not applicable to the `OpReturnCode` of the invoked operation itself).
MJ_FAILED_NON_SPECIFIC	Nonspecific failure. Shall not be used if any of the subsequent enumerations apply.
MJ_COMMUNICATION_ERROR	N/A
MJ_BUSY	Requested operation is legal but cannot be executed since the target Server Object is busy and cannot support multiple requests.
MJ_SERVICE_UNAVAILABLE	The requested service is not currently available because of the internal state of the Server Object, for example, when it is disabled.
MJ_ILLEGAL_OPERATION	N/A
MJ_FAILED_INPUT_ARGUMENT	One or more input arguments are invalid or missing.
MJ_FAILED_OUTPUT_ARGUMENT	One or more output arguments are invalid or missing.
MJ_FAILED_MARSHALING	N/A
MJ_FAILED_DEMARSHALING	N/A
MJ_ILLEGAL_TRANSACTION	There is no such server transaction as indicated in the call.
MJ_OPERATION_INTERRUPTED	The operation did not complete because of errors other than a server timeout.
MJ_OPERATION_TIMEOUT	The indicated operation did not complete before a server timeout expired.
MJ_INSUFFICIENT_RESOURCES	Requested operation is legal but cannot be executed because of lack of resources in the target Server Object.
MJ_BLOCK_COOKIE_MISMATCH	N/A
MJ_TRANSDUCER_ERROR	A transducer failed during operation.
MJ_SUSPECT_INVOCATION_RESULT	A value provided by the target may be in error (e.g., transducer overloaded, computation overflow).
MJ_NOT_PROPERLY_CONFIGURED	The Server Object has detected some state that is not configured or is improperly configured.
Reserved	Reserved for future use by the IEEE 1451.1 standard committee.
User-defined	May be used by industry groups or implementers.

7.2.3.3.4 Restrictions on allowed Major Field values for an `OpReturnCode`

Errors may be detected in the execution of operations having an `OpReturnCode` as part of the signature. The Major Field shall be selected from Table 14.

Copyright © 2000 IEEE. All rights reserved.

Table 14—Allowed `OpReturnCode` values

Name	Meaning
MJ_COMPLETE	The operation did not detect any error during execution.
MJ_NOP_OPERATION	The operation has an Interface Only Implementation.
MJ_NO_RETURN_CODE	N/A
MJ_FAILED_NON_SPECIFIC	Nonspecific failure. Shall not be used if any of the subsequent enumerations apply.
MJ_COMMUNICATION_ERROR	N/A (see exceptions).
MJ_BUSY	Requested operation is legal but cannot be executed since the target Object is busy and cannot support multiple requests.
MJ_SERVICE_UNAVAILABLE	The requested service is not currently available because of the internal state of the Object, for example, when it is disabled.
MJ_ILLEGAL_OPERATION	N/A
MJ_FAILED_INPUT_ARGUMENT	One or more input arguments are invalid or missing.
MJ_FAILED_OUTPUT_ARGUMENT	One or more output arguments are invalid or missing.
MJ_FAILED_MARSHALING	N/A (see exceptions).
MJ_FAILED_DEMARSHALING	N/A
MJ_ILLEGAL_TRANSACTION	There is no such transaction as indicated in the call.
MJ_OPERATION_INTERRUPTED	The operation did not complete because of errors other than a timeout defined on the target Object.
MJ_OPERATION_TIMEOUT	The indicated operation did not complete before a timeout defined on the target Object expired.
MJ_INSUFFICIENT_RESOURCES	Requested operation is legal but cannot be executed because of lack of resources in the target Object.
MJ_BLOCK_COOKIE_MISMATCH	N/A
MJ_TRANSDUCER_ERROR	A Transducer failed during operation.
MJ_SUSPECT_INVOCATION_RESULT	A value provided by the target may be in error (e.g., Transducer overloaded, computation overflow).
MJ_NOT_PROPERLY_CONFIGURED	The Object has detected some state that is not configured or is improperly configured.
Reserved	Reserved for future use by the IEEE 1451.1 standard committee.
User-defined	May be used by industry groups or implementers.

In the case of the `ExecuteAsynchronous` operation on an Asynchronous Client Port Object, a `Publish` operation on a Publisher Port, or a `PublishWithIdentification` operation on a Self Identifying Publisher Port, the following exceptions shall be observed:

— The Major Field value of `MJ_COMMUNICATION_ERROR` shall be allowed and shall indicate a general communication error.

— The Major Field value of `MJ_FAILED_MARSHALING` shall be allowed and shall indicate a failure to correctly marshal data.

Copyright © 2000 IEEE. All rights reserved.

7.2.4 The behavior of operations

7.2.4.1 Get and Set Semantics operation requirements

Get and Set Semantics operations shall obey the concurrency requirements of 7.2.4.2.

7.2.4.1.1 Get Semantics: definition

A Get Semantics operation is any operation, Network Visible or Local that allows any portion of the internal state of an object to be observed.

The returned values shall be determined by the internal behavior defined for the class.

Get Semantics operations include operations described as `Read` or with other Operation Names that potentially operate to view the internal state variables of an object.

7.2.4.1.2 Set Semantics: definition

A Set Semantics operation is any operation, Network Visible or Local, that allows any portion of the internal state of an object to be modified.

If a Set Semantics operation is provided for a class, the value that would be returned by a Get Semantics operation that precedes in time any Set Semantics operation on the same state variable shall return either

— The default value defined for the datatype of the variable

— An initial value for the variable specified as part of the class definition

— An application-specific initial value for the variable.

If a Network Visible Set Semantics operation is provided, a corresponding Network Visible Get Semantics operation shall be provided.

Set Semantics operations include operations described as `Write` or with other Operation Names that potentially operate to change the internal state variables of an object.

7.2.4.2 Concurrency requirements

The concurrency requirements of 7.2.4.2 shall apply to all IEEE 1451.1 classes or subclasses thereof. Subclass definitions shall include the specification of any additional concurrency requirements as part of each class's behavioral definition.

The term Concurrent Execution shall be interpreted to mean that subsequent to the invocation of an operation on an object, and prior to the completion of that operation, an operation is invoked on the same or a different object.

With respect to the concurrency specifications of this clause and its subclauses, the term operation shall include

— Both Network Visible and Local operations

— Any implementation-specific Local operations

— Any other object activity, including internal activity that potentially interacts with a concurrent operation

Copyright © 2000 IEEE. All rights reserved.

Unless otherwise stated, the resolution of concurrency violations is outside the scope of this standard.

In 7.2.4.2, serialization of two operations shall be interpreted as meaning that one operation completes before the second operation starts execution.

The management of the various forms of Concurrent Execution is specified in 7.2.4.2.1 and 7.2.4.2.2.

These and other related problems are inherent in distributed systems. Analogous difficulties arise in centralized systems. These problems are the subject of a rich literature [B1].

7.2.4.2.1 Inter-NCAP concurrency specifications

Any IEEE 1451.1 system that has more than one NCAP is a distributed multiprocessor system and, as such, must contend with inter-NCAP concurrent, asynchronous operation execution.

The management of inter-NCAP concurrency is outside the scope of this standard.

To help IEEE 1451.1 application developers deal with inter-NCAP concurrency, implementations may use the Mutex Service and Condition Variable Services; see 11.11 and 11.12. These Services provide the application interfaces to two underlying mechanisms for synchronizing the execution of two or more code segments residing in distinct NCAP process spaces.

7.2.4.2.2 Intra-NCAP concurrency specifications

An IEEE 1451.1 system must also contend with intra-NCAP Concurrent Execution if the operating system of any NCAP in the system supports multiple processes, threads, or interrupts.

Case 1: Multiple process concurrency

If an IEEE 1451.1 NCAP has multiple executing processes, each process has its own Local NCAP Block, and interprocess communication is logically the same as inter-NCAP communication. The management of this type of concurrency is outside the scope of this standard.

Case 2: Multiple operations on objects within a single process

An IEEE 1451.1 implementation may allow Concurrent Execution of operations on objects within an NCAP process.

Except as provided in cases 3 and 4 of this clause, if Concurrent Execution is implemented, two or more concurrently executing operations shall behave as if the operation execution is serialized.

Unless otherwise stated in a subclass definition, there shall be no dependence assumed on the order of any possible serial execution.

Unless otherwise stated in a class definition, there shall be no requirement to support the Concurrent Execution of operations.

NOTE—If an implementation is capable of detecting incipient concurrency violations, possible actions are to serialize, or to return an appropriate error code to one or more of the operations.

 Copyright © 2000 IEEE. All rights reserved.

Case 3: Interfering operations: Precedence Concurrency

Classes may specify a special set of operations obeying Precedence Concurrency defined as follows:

— An operation A that is a member of the set, invoked on an instance of the class concurrently with another operation B not a member of the set on the same instance, shall execute to completion irrespective of the completion status of operation B

— The behavior of operation B in these circumstances is outside the scope of this standard unless otherwise stated in a class definition

— The invocation of two operations from this set shall obey the concurrency requirements Case 2 for multiple operations on objects within a single process.

NOTE—This form of concurrency is required of operations that may be required to interfere with ongoing activity to properly implement their behavior. A typical example is a Reset operation that is being used to "clear" a fault in an object that is preventing normal operation of the system.

Case 4: Operations and continuing object activity: Active Concurrency

Classes may specify a special set of operations obeying Active Concurrency, defined as follows:

— An operation A that is a member of the set, invoked on an instance of the class concurrently with some continuing activity initiated earlier by some other operation on the instance, shall execute to completion irrespective of any possible termination of the continuing activity.

— If operation A accesses any internal state variable, also accessed by the continuing activity, the accesses to this state variable shall behave as though serialized.

— The invocation of two operations from this set shall obey the concurrency requirements of Case 2 for multiple operations on objects within a single process.

NOTE—This form of concurrency is required of operations that may be required to cooperate with ongoing activity to properly implement their behavior. A typical example is a status operation that is being used to observe the progress of an ongoing activity. System and application developers should note that the failure of operations initiating concurrent ongoing activity may introduce significant problems in an operating system. A typical failure scenario is the failure of a client to receive a transaction ID before the expiration of a Client Port timeout, when initiating ongoing activity. Other clients will be prevented from initiating a transaction but the original client is unaware that the activity has actually started. Classes in this standard subject to this behavior include

— Actions
— Files
— Partitioned Files
— Mutex Services
— Condition Variable Services
— Asynchronous Client Ports

7.2.4.3 Operation failure behavior specifications

Unless otherwise stated in a class or subclass definition, operations that produce a change of state in the target Object shall behave as follows:

— The successful completion of an operation shall produce the specified behavior. Any return code shall so indicate the successful completion by a Major Field value of `MJ_COMPLETE`.

— An unsuccessful execution of an operation may produce unspecified behavior in the target Object. Any return code shall so indicate an unsuccessful execution by a Major Field value not equal to `MJ_COMPLETE`.

Copyright © 2000 IEEE. All rights reserved.

— The execution of an operation that creates a new Object instance of a class A may fail. Such failure shall leave the local process in a state such that attempts to invoke operations of class A on the new Object shall fail.

— The execution of an operation that creates a new transaction within an existing Object instance of a class A may fail. Such failure shall leave the Object instance in a state such that attempts to invoke operations on the Object instance that are specific to the new transaction shall fail. That is, it shall appear that the new transaction was not created.

— The execution of an operation that changes the state of a state machine specified in the state behavior of an Object instance of class A may fail. Such failure shall leave the state of the Object instance unchanged or shall result in a change to a class specific state defined as an error condition state.

7.3 Block Cookie properties

Block Cookies provide a principled way for a client to tell whether the context of the server has changed. The responsibility for declaring a change of context lies with the server. The management of cookie values is mediated by the Local NCAP Block.

7.3.1 Block Cookies associated with an Object

Only Objects of class `IEEE1451_Block` shall have a Block Cookie.

All Network Visible Objects of class `IEEE1451_Block` shall have a Block Cookie.

All Network Visible Objects of class `IEEE1451_Entity` have associated with them a Block Cookie. This Block Cookie shall be termed the Associated Block Cookie for purposes of this standard. This Associated Block Cookie shall be determined as follows:

— If the Object is an instance of an `IEEE1451_Block`, then the Associated Block Cookie shall be the Block Cookie of the Block itself.

— If the Object is not an instance of an `IEEE1451_Block`, then the Associated Block Cookie shall be the Block Cookie of the Owning Block of the instance, see 5.1.3.

7.3.2 Block Cookie implementation restrictions

The implementation of the Block Cookie mechanism for a Block is outside the scope of this standard.

Implementation of the Block Cookie is subject to the restrictions of 7.3.2.1 through 7.3.2.8.

7.3.2.1 Block Cookie Restriction 1

The Block Cookie of a Block shall be locally visible for the following purposes:

— For implementing the `GetObjectProperties` operation, see 8.2.1.7

— For implementing the `GetNetworkVisibleServerObjectProperties` operation; see 9.1.1.11

— For implementing the publishing of publications with a Publication Key `PSK_NETWORK_VISIBLE_SERVER_OBJECT_PROPERTIES`; see 9.2.3.2

— For implementing the server-side behavior of client-server communication; see 8.2.3.2

— For any purpose of any object Owned by the Block

 Copyright © 2000 IEEE. All rights reserved.

7.3.2.2 Block Cookie Restriction 2

The underlying type for the Block Cookie shall be `UInteger16`; see `ObjectProperties` structure in 6.2.6.

A `UInteger16` variable with value 0 when interpreted as a Block Cookie has a Block Cookie value NOT_SET.

7.3.2.3 Block Cookie Restriction 3

A `UInteger16` variable that when interpreted as a Block Cookie has a value NOT_SET shall not represent the Block Cookie of any Block Object.

7.3.2.4 Block Cookie Restriction 4

The locally visible Block Cookie of a Block shall not have the value NOT_SET.

7.3.2.5 Block Cookie Restriction 5

The values of two Block Cookies shall match if and only if

— The values are identical, **AND**

— Neither value is NOT_SET

7.3.2.6 Block Cookie Restriction 6

The value of the Block Cookie of a Block shall change if any of the following conditions occur:

— The Network Visible interface of a Block or of an object Directly or Indirectly Owned by the Block has changed. For example, the datatype of an argument of an operation on the Block has changed.

— The clients view of the semantic definition of the Block or of an object Directly or Indirectly Owned by the Block has changed. For example, a Parameter no longer represents the same quantity even though the Parameter interface remains unchanged.

It is recommended that developers of servers carefully consider whether a change in server state warrants a change in the Block Cookie under this clause.

7.3.2.7 Block Cookie Restriction 7

The occurrence of repeating Block Cookie values resulting from changes in the Block's Block Cookie shall be minimized.

7.3.2.8 Block Cookie Restriction 8

A Block may cause its Block Cookie to be changed only by invoking the local `ChangeBlockCookie` operation on its Local NCAP Block; see 9.2.2.5.

Copyright © 2000 IEEE. All rights reserved.

7.4 Value uncertainty specification

The uncertainty in a scalar value shall be represented by the `Uncertainty` datatype structure; see 6.2.10. The members of this structure are

— `interpretation`
— `uncertaintyValue`
— `coverageFactor`
— `customFactor`

It is strongly recommended that the ISO uncertainty guidelines [ISO uncertainty] be followed wherever possible.

The computation of the `uncertaintyValue` shall be based on the same units as the value of the variable to which the uncertainty pertains unless otherwise stated in a class definition.

7.4.1 Member `interpretation` specification

The value of the `interpretation` member shall be taken from the `UncertaintyInterpretation` enumeration of Table 15.

IDL: enumeration `UncertaintyInterpretation;`

Table 15—`Uncertainty Interpretation` enumeration

Enumeration	Value
`UI_ISO_GUIDELINES`	0
`UI_NO_ERROR`	1
`UI_CUSTOM`	2
Reserved	3–255

7.4.2 Specification of the remaining members

The interpretation of the remaining members of the `Uncertainty` datatype structure is dependent on the value of the `interpretation` member.

Case 1: For an `interpretation` value of `UI_ISO_GUIDELINES`, the following restrictions shall apply:

— The `uncertaintyValue` member shall be the “uncertainty” computed by following the ISO guidelines.
— The `coverageFactor` member shall be the “coverage factor” computed by following the ISO guidelines. If an appropriate value is not available under these guidelines the value 2 shall be assigned following the recommendation of the NIST guidelines [NIST].
— The interpretation of the `uncertaintyValue` member shall be that the actual value of the variable for which this uncertainty is applicable shall be in the range

  value – `uncertaintyValue` ≤ value ≤ value + `uncertaintyValue`

— The `customFactor` has no meaning for this value of interpretation and shall be set to the default value for its type.

 Copyright © 2000 IEEE. All rights reserved.

Case 2: For an `interpretation` value of `UI_NO_ERROR` the following restrictions shall apply:

— The actual value of the variable for which this uncertainty is applicable shall be interpreted as having perfect accuracy; that is, the "uncertainty" is zero.

— The `uncertaintyValue`, `coverageFactor`, and `customFactor` members shall be set to the default values for their types.

Case 3: For the interpretation value of `UI_CUSTOM`, the interpretation of the `uncertaintyValue`, `coverageFactor`, `customFactor` members is application-specific and outside the scope of this standard.

NOTE—One possible use of this interpretation is to augment the definition of uncertainty used with `UI_ISO_GUIDELINES`, to represent uncertainty that is asymmetrical around the expected value. The `customFactor` member is provided for this purpose. This problem is studied in the ISA guidelines [ISA uncertainty].

8. Top-level class definitions

8.1 Root abstract class

Abstract Class

Class: `IEEE1451_Root`

Parent Class: `IEEE1451_Root`

Class ID: 1

Description: The Root abstract class shall be the root for the class hierarchy of all objects defined by this standard.

Class summary:

Network Visible operations (see Table 16).

Table 16—`IEEE_Root` Network Visible operations

Operation Name	Requirement	Operation ID
`GetClassName`	(m)	1
`GetClassID`	(m)	2

Publications: There are no publications defined for this class.

Subscriptions: There are no subscriptions defined for this class.

8.1.1 Operation specifications: Network Visible

8.1.1.1 Operation `GetClassName` specification

IDL: `OpReturnCode GetClassName(out String class_name);`

There are no operation-specific additions to the Minor Field of the return code enumeration.

Copyright © 2000 IEEE. All rights reserved.

8.1.1.1.1 argument `class_name` specification

The argument `class_name` shall have the value of the Class Name for the class of which the addressed object is an instance. The String form of the `IEEE1451_ClassDescriptor` appearing in the class specification header shall be the value of the Class Name for each instance of the class that is identified by the `class_id`. In this case, the descriptor is the `String` form of `IEEE1451_Root` and the identifier has the value 1.

The `class_name` argument for classes defined by this standard shall be represented in the IEEE 1451.1 character set defined in Annex F.

Example: The value of the output argument `class_name` returned by the operation `GetClassName` on an instance of the class `IEEE1451_Root`, is the `String` with member values

```
characterSet      = 255  (i.e. SCS_IEEE1451DOT1)
characterCode     = 0    (i.e. SCC_OCTET1)
language          = 255  (i.e. SL_IEEE)
stringData        = an Octet Array of length 1 with the value of the octet being 10 (decimal)
```

The choice of string, character set, and language representations for the `String` form of `class_name` values for classes not defined by this standard is outside the scope of the standard.

8.1.1.2 Operation `GetClassID` specification

IDL: `OpReturnCode GetClassID(out ClassID class_id);`

There are no operation-specific additions to the Minor Field of the return code enumeration.

8.1.1.2.1 argument `class_id` specification

The `class_id` argument shall be value of the Class ID of the class. The Class ID is the designator of the position in the class hierarchy of the class for which the target object is an instance. The Class ID appearing in object description header, for example, in this case 1 shall be the value for the `class_id` for each instance of the class.

8.1.2 Common behavior

There is no additional behavior defined at this level.

8.2 Entity abstract class

Abstract Class

Class: `IEEE1451_Entity`

Parent Class: `IEEE1451_Root`

Class ID: 1.1

Description: The Entity abstract class shall be the root for the class hierarchy of all Objects defined by this standard that may be made visible over the network.

Class summary:

Network Visible operations (see Table 17).

 Copyright © 2000 IEEE. All rights reserved.

Table 17—`IEEE1451_Entity` Network Visible operations

Operation Name	Requirement	Operation ID
`GetObjectTag`	(m)	2048
`SetObjectTag`	(m)	2049
`GetObjectID`	(m)	2050
`GetObjectName`	(m)	2051
`GetDispatchAddress`	(m)	2052
`GetOwningBlockObjectTag`	(m)	2053
`GetObjectProperties`	(m)	2054

Local operations (see Table 18).

Table 18—`IEEE1451_Entity` Local operations

Operation Name	Requirement
`Perform`	(m)

Publications: There are no publications defined for this class.

Subscriptions: There are no subscriptions defined for this class.

8.2.1 Operations specifications: Network Visible

8.2.1.1 Operation `GetObjectTag` specification

IDL: `OpReturnCode GetObjectTag(out ObjectTag object_tag);`

There are no operation-specific additions to the Minor Field of the return code enumeration.

8.2.1.1.1 argument `object_tag` specification

The `object_tag` argument value shall be the current value of the Object Tag of the Object. Object Tags are logical names for the network address of a network-addressable Object. The properties and generation of Object Tags are defined in 8.2.3.4.

8.2.1.2 Operation `SetObjectTag` specification

IDL: `OpReturnCode SetObjectTag(in ObjectTag object_tag);`

There are no operation-specific additions to the Minor Field of the return code enumeration.

8.2.1.3 Operation `GetObjectID` specification

IDL: `OpReturnCode GetObjectID(out ObjectID object_id);`

There are no operation-specific additions to the Minor Field of the return code enumeration.

Copyright © 2000 IEEE. All rights reserved.

8.2.1.3.1 argument `object_id` specification

The `object_id` argument value shall be the current value of the Object ID of the Object. The properties and generation of Object IDs are defined in 8.2.3.5.

8.2.1.4 Operation `GetObjectName` specification

IDL:
```
OpReturnCode GetObjectName(out String object_name);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

8.2.1.4.1 argument `object_name` specification

The `object_name` argument value shall be the value of the Object Name for the Object. The properties and generation of Object Names are defined in 8.2.3.6.

8.2.1.5 Operation `GetDispatchAddress` specification

IDL:
```
OpReturnCode GetDispatchAddress(
            out ObjectDispatchAddress dispatch_address);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

8.2.1.5.1 argument `dispatch_address` specification

The `dispatch_address` argument value shall be the value of the Object Dispatch Address of the Object. The Object Dispatch Address shall be the network-specific address used by the underlying network infrastructure to address the Object. The properties and generation of Object Dispatch Addresses are defined in 8.2.3.7.

8.2.1.6 Operation `GetOwningBlockObjectTag` specification

IDL:
```
OpReturnCode GetOwningBlockObjectTag(
            out ObjectTag owning_block_object_tag);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

8.2.1.6.1 argument `owning_block_object_tag` specification

The `owning_block_object_tag` argument value shall be the current value of the Object Tag of the Owning Block Object; see 5.1.3.1.

8.2.1.7 Operation `GetObjectProperties` specification

IDL:
```
OpReturnCode GetObjectProperties(
            out ObjectProperties object_properties);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

8.2.1.7.1 argument `object_properties` specification

The `object_properties` argument shall be an instance of the `ObjectProperties` structure, holding the current values for the Object Tag, Object Dispatch Address, Object Name, and the Associated Block Cookie of this Object and the Object Tag of the Owning Block of this Object.

 Copyright © 2000 IEEE. All rights reserved.

Table 72—`CoordinateSystem` enumeration

Enumeration	Value	Meaning
`CC_ARBITRARY`	0	Starting with the 0th argument of the value, the values represent dimensions of a user-defined vector.
`CC_CARTESIAN`	1	Starting with the 0th argument of the value, the values represent the x, y, and z components of a right-handed cartesian coordinate space.
`CC_CYLINDRICAL`	2	Starting with the 0th argument of the value, the values represent the r, Θ , and z components of a right-handed cylindrical coordinate space.
`CC_SPHERICAL`	3	Starting with the 0th argument of the value, the values represent the r, Θ , and Φ components of a right-handed spherical coordinate space.
`CC_PLANETARY`	4	Starting with the 0th argument of the value, the values represent the latitude, longitude, and elevation components of a planetary coordinate space.
`CC_ORIENTATION`	5	Starting with the 0th argument of the value, the values represent the roll, pitch, and yaw components of the angular orientation with respect to a defined spatial direction.
Reserved	6–127	
Open to industry	128–255	

10.8 Vector Series Parameter class

Class: `IEEE1451_VectorSeriesParameter`

Parent Class: `IEEE1451_VectorParameter`

Class ID: 1.1.2.1.1.1.2.1

Description: The Vector Series Parameter class is used to model a uniform series of physical world quantities that have dimensions and orientation associated with them and are appropriately represented as mathematical vectors. Examples of such quantities are electric field, acceleration, and torque and arbitrary relations such as temperature/pressure pairs. This class may be used for single-axis vectors, for example, a single-axis accelerometer, to convey the correct physical interpretation.

Class summary:

Network Visible operations (see Table 73).

Table 73—`IEEE1451_VectorSeriesParameter` Network Visible operations

Operation Name	Requirement	Operation ID
`GetAbscissaUnits`	(m)	14342
`SetAbscissaUnits`	(o)	14343
`GetAbscissaIncrement`	(m)	14344
`SetAbscissaIncrement`	(o)	14345
`GetAbscissaOrigin`	(m)	14346
`SetAbscissaOrigin`	(o)	14347

Copyright © 2000 IEEE. All rights reserved.

Publications: There are no publications defined for this class.

Subscriptions: There are no subscriptions defined for this class.

10.8.1 Operation specifications: Network Visible

10.8.1.1 Operation `GetAbscissaUnits` specification

```
IDL: OpReturnCode GetAbscissaUnits(out Units abscissa_units);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.8.1.1.1 argument `abscissa_units` specification

The `abscissa_units` argument value shall be the units of the `abscissa_increment` and `abscissa_origin`.

10.8.1.2 Operation `SetAbscissaUnits` specification

```
IDL: OpReturnCode SetAbscissaUnits(in Units abscissa_units);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.8.1.3 Operation `GetAbscissaIncrement` specification

```
IDL: OpReturnCode GetAbscissaIncrement(
           out Argument abscissa_increment);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.8.1.3.1 argument `abscissa_increment` specification

The `abscissa_increment` argument value shall be the increment in the designated `abscissa_units` between each member of the series. Negative values of `abscissa_increment` are allowed. The datatype of the value of the argument shall be the same datatype as would be used if the incremented abscissa variable were a measured value represented as a Parameter, for example, `TimeRepresentation` for an abscissa that represents time.

10.8.1.4 Operation `SetAbscissaIncrement` specification

```
IDL: OpReturnCode SetAbscissaIncrement(
           in Argument abscissa_increment);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.8.1.5 Operation `GetAbscissaOrigin` specification

```
IDL: OpReturnCode GetAbscissaOrigin(
           out Argument abscissa_origin);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

 Copyright © 2000 IEEE. All rights reserved.

10.8.1.5.1 argument `abscissa_origin` specification

The value of the `abscissa_origin` argument value shall be the starting value of the abscissa. This starting value is the abscissa value corresponding to the value of the 0th member Argument of the Argument Array, `data`, the output argument of the `Read` operation on the Parameter. `abscissa_origin` shall be represented in the designated `abscissa_units` and of the same datatype as `abscissa_increment`.

10.8.1.6 11.Operation `SetAbscissaOrigin` specification

```
IDL: OpReturnCode SetAbscissaOrigin(in Argument abscissa_origin);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.8.2 Vector Series Parameter class behavior

10.8.2.1 Specific Vector Series Parameter operation restrictions based on state

In addition to and conformant with the restrictions of 10.7.2.1, the allowed operations defined on a Vector Series Parameter instance shall be restricted as defined in Table 74. Where an operation is indicated as operational, it still shall be restricted to specific uses defined by the restrictions of 9.1.3.2.1, 9.1.3.2.2, and 9.1.3.2.3.

Table 74—`IEEE1451_VectorSeriesParameter` class specific restrictions

Operation	Specified value of `rule_basis`		
	BL_ UNINITIALIZED	BL_ INACTIVE	BL_ ACTIVE
`GetAbscissaUnits`, `GetAbscissaIncrement`, `GetAbscissaOrigin`	Local invocation only	Operational	Operational
`SetAbscissaUnits`, `SetAbscissaIncrement`, `SetAbscissaOrigin`	Local invocation only	Operational	Not operational

10.8.2.2 Operation `SetAbscissaUnits` behavior specification

The `SetAbscissaUnits` operation may be an Interface Only Implementation if the Parameter metadata is “read only;” otherwise, it shall be a Full Implementation.

10.8.2.3 Operations `SetAbscissaIncrement` and `SetAbscissaOrigin` behavior specifications

The `SetAbscissaIncrement` and `SetAbscissaOrigin` operations may be Interface Only Implementations if these aspects of the Parameter are “read only;” otherwise, they shall have Full Implementations.

10.8.2.4 Data structure semantics

10.8.2.4.1 Restrictions on the value of Physical Parameter Type

The allowed values for Physical Parameter Type for this class are

Copyright © 2000 IEEE. All rights reserved.

— PP_VECTOR_ANALOG_SERIES

— PP_VECTOR_DISCRETE_SERIES

— PP_VECTOR_DIGITAL_SERIES

The structure and interpretation of both data and metadata is Physical Parameter Type dependent.

10.8.2.4.2 Argument data behavior specifications for the read and write operations of the Parameter

For the vector series datatypes, the Argument Array, data, that is the argument of the various Read and Write operations shall be as follows; see 10.4.2.4:

— The Argument Array, data, shall be of length 1

— The TypeCode discriminant of the 0th member Argument of data shall be PHYSICAL_PARAMETER_DATA_TC. The union PhysicalParameterData, which is the value of this Argument, shall have discriminant PP_VECTOR_ANALOG_SERIES, PP_VECTOR_DISCRETE_SERIES, or PP_VECTOR_DIGITAL_SERIES. In all three cases, the value of the PhysicalParameterData union is a PhysicalParameterSeriesData structure with members defined per 6.2.15 as follows:

  — The value field of the PhysicalParameterSeriesData structure is of type ArgumentArray. This Argument Array shall be of length dimension. Each Argument member of the Argument Array shall represent a single dimension of the vector.

  — The value of the jth member Argument of the Argument Array, value-dimension-j-array, shall be an Array of length the number of vectors in the series. The ith member of this Array, value-dimension-j-array[i] shall contain the value of the jth dimension of the ith vector in the series.

  — The basic datatype of the Array, value-dimension-j-array, shall be that corresponding to the TypeCode value of the jth member of the TypeCode Array returned by the GetDatatype operation on the Parameter.

  — The units of each member of value-dimension-j-array shall be those of the jth member of the UnitsArray returned by the GetUnits operation on the Parameter.

  — The 0th Array member of each value-dimension-j-array shall be the value of the jth dimension of the vector corresponding to the abscissaOrigin field of the PhysicalParameterSeriesData structure.

— The timestamp field of the PhysicalParameterSeriesData structure shall represent the time at which the value field—that is, the entire vector series—was updated.

— The Argument in the abscissaIncrement field of the PhysicalParameterSeriesData structure shall have as value the value returned in the output argument, abscissa_increment, by the GetAbscissaIncrement operation on the Vector Series Parameter.

— The Argument in the abscissaOrigin field of the PhysicalParameterSeriesData structure shall have as value the value returned in the output argument, abscissa_origin, by the GetAbscissaOrigin operation on the Vector Series Parameter.

These relationships are shown schematically in Figure 21.

10.8.2.4.3 metadata behavior specifications associated with the Parameter

PP_VECTOR_ANALOG_SERIES: For vector analog series datatypes, the metadata shall be represented by the VectorAnalogSeriesMetadata structure.

 Copyright © 2000 IEEE. All rights reserved.

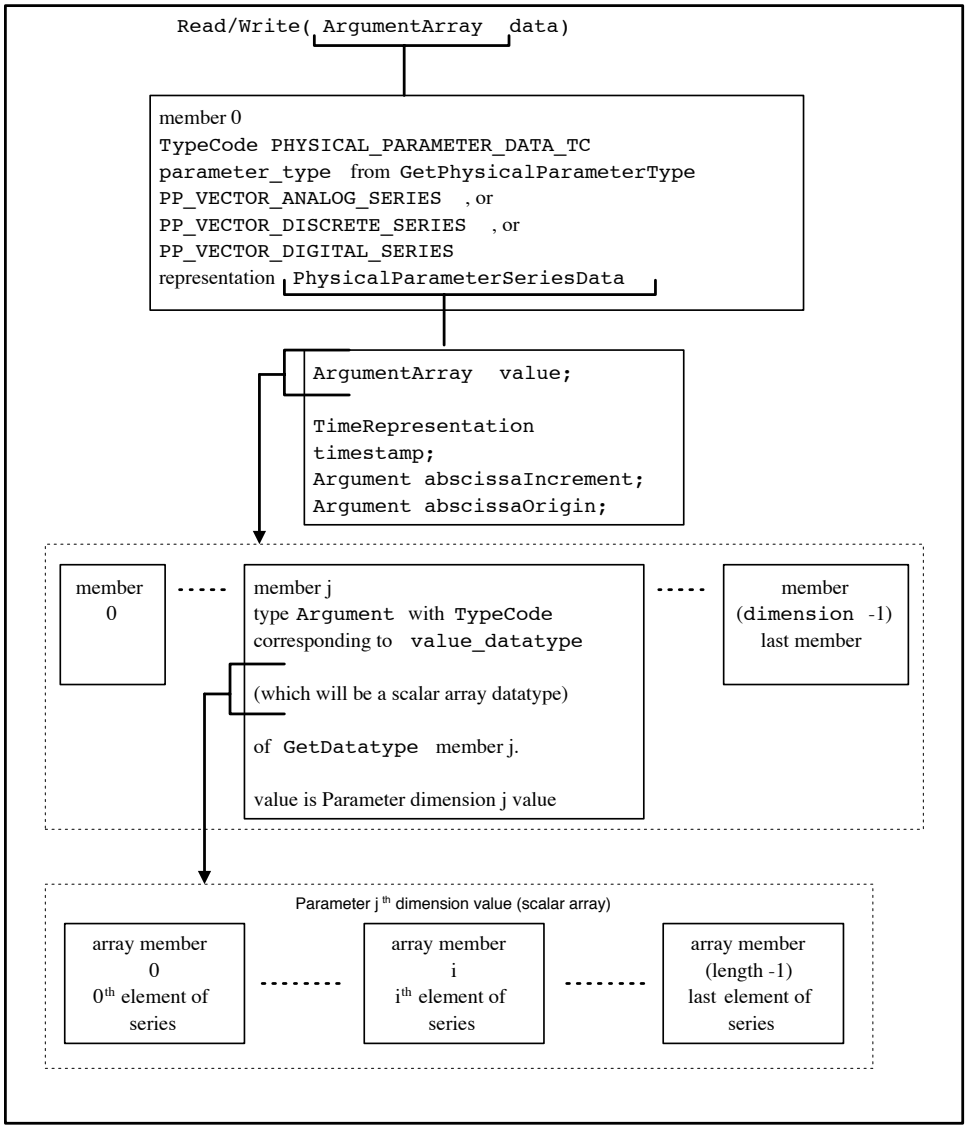

Figure 21—Vector Series Parameter data representation

`PP_VECTOR_DISCRETE_SERIES:` For vector discrete series datatypes, the metadata shall be represented by the `VectorDiscreteSeriesMetadata` structure.

`PP_VECTOR_DIGITAL_SERIES:` For vector digital series datatypes, the metadata shall be represented by the `VectorDigitalSeriesMetadata` structure.

The interpretation of the fields of these data structures shall be as specified in 10.4.2.

10.8.2.5 Member `abscissaIncrement` behavior specification

The series of vectors represented as described in 10.8.2.4.2 by the Argument Array in the `value` field of the `PhysicalParameterSeriesData` structure, the ordinate values of the Parameter, shall be the value of the Parameter corresponding to successive uniform intervals of an abscissa variable. The increment between each successive abscissa value shall be the value of member `abscissaIncrement`. The actual value of a given abscissa value corresponding to a particular ordinate value shall be determined by the

Copyright © 2000 IEEE. All rights reserved.

`abscissaOrigin` member value added to a multiple of the value of the member `abscissaIncrement.` This multiple `is determined by the` ordinate value's position in the series.

10.9 Time Parameter class

Class: `IEEE1451_TimeParameter`

Parent Class: `IEEE1451_Parameter`

Class ID: 1.1.2.1.2

Description: The Time Parameter class may be used to represent time parametric values. The purpose of the Time Parameter class and its subclasses is to model Network Visible variables that directly or indirectly represent the time of some event, or the duration between two events, where the significant characteristic of the event is the time rather than some other value.

Class summary:

Network Visible operations (see Table 75).

Table 75—`IEEE1451_TimeParameter` Network Visible operations

Operation Name	Requirement	Operation ID
`GetUncertainty`	(m)	8208
`GetTimeType`	(m)	8209
`GetEpochRepresentation`	(m)	8210
`GetEpoch`	(m)	8211

Local Operations (see Table 76).

Table 76—`IEEE1451_TimeParameter` Local operations

Operation Name	Requirement
`RegisterNotifyOnUpdate`	(o)
`DeregisterNotifyOnUpdate`	(o)

Publications: There are no publications defined for this class.

Subscriptions: There are no subscriptions defined for this class.

10.9.1 Operation specification: Network Visible

10.9.1.1 Operation `GetUncertainty` specification

```
IDL: OpReturnCode GetUncertainty(out Uncertainty uncertainty);
```

 Copyright © 2000 IEEE. All rights reserved.

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.9.1.1.1 argument `uncertainty` specification

The `uncertainty` argument value shall be based on time, expressed in seconds, and shall represent the uncertainty of the value of the Parameter.

10.9.1.2 Operation `GetTimeType` specification

```
IDL: OpReturnCode GetTimeType(out UInteger8 time_type);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.9.1.2.1 argument `time_type` specification

The `time_type` argument value shall be taken from the `TimeType` enumeration shown in Table 77, and shall define the interpretation of the value of the Parameter.

IDL: enumeration `TimeType;`

Table 77—`TimeType` enumeration

Enumeration	Value	Meaning
`TT_TIMESTAMP`	0	The time value is a timestamp marking the time with respect to an epoch.
`TT_TIME_DURATION`	1	The value represents the difference between two timestamps.
Reserved	2–255	

10.9.1.3 Operation `GetEpochRepresentation` specification

```
IDL: OpReturnCode GetEpochRepresentation(
          out UInteger8 epoch_representation);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.9.1.3.1 argument `epoch_representation` specification

The `epoch_representation` argument value shall be taken from the `TimeEpoch` enumeration; see 13.2.1.6. `epoch_representation` defines the method of specifying the epoch or origin of the time scale on which the value of the Parameter is based.

10.9.1.4 Operation `GetEpoch` specification

```
IDL: OpReturnCode GetEpoch(out TimeRepresentation epoch);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

Copyright © 2000 IEEE. All rights reserved.

10.9.1.4.1 argument `epoch` specification

The `epoch` argument value shall represent the origin of the timescale on which the value of the Parameter is based.

10.9.2 Operations specifications: Local

10.9.2.1 Local operation `RegisterNotifyOnUpdate` specification

```
IDL: OpReturnCode RegisterNotifyOnUpdate(
           in <local operation reference>
               notification_operation,
           in UInteger16 registration_id);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.9.2.1.1 argument `notification_operation` specification

The `notification_operation` argument value shall be a local reference to the local operation of the registering object that shall be invoked to signal notification. The signature of the referenced `notification_operation` shall be

```
OpReturnCode <local operation name>(
           in UInteger16 registration_id,
           in TimeRepresentation timestamp);
```

10.9.2.1.2 argument `timestamp` specification

The `timestamp` argument value shall be the occurrence time of the class-specific change, mandating the notification.

10.9.2.1.3 argument `registration_id` specification

The `registration_id` argument value shall be an identifier generated by the object registering for update to distinguish among registrations with different target objects.

10.9.2.2 Local operation `DeregisterNotifyOnUpdate` specification

```
IDL: OpReturnCode DeregisterNotifyOnUpdate(
           in <local operation reference>
               notification_operation,
           in UInteger16 registration_id);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.9.3 Time Parameter class behavior

10.9.3.1 Specific Time Parameter operation restrictions based on state

In addition to and conformant with the restrictions of 10.2.2.1, the allowed operations defined on a Time Parameter instance shall be restricted as defined in Table 78. Where an operation is indicated as operational, it still shall be restricted to specific uses defined by the restrictions of 9.1.3.2.1, 9.1.3.2.2, and 9.1.3.2.3.

 Copyright © 2000 IEEE. All rights reserved.

Table 78—`IEEE1451_TimeParameter` class specific restrictions

Operation	Specified value of `rule_basis`		
	BL_UNINITIALIZED	**BL_INACTIVE**	**BL_ACTIVE**
All operations defined by this class	Local invocation only	Operational	Operational

10.9.3.2 Operations `Read` and `Write` behavior specifications

The value of the 0th Argument member of the Argument Array argument of the `Read` or `Write` operations inherited from the `IEEE1451_Parameter` class shall be the value of the time represented by this Parameter.

10.9.3.3 Notification operations behavior specifications

The local `notification_operation` shall be invoked for all registered operations, whenever the value of the Time Parameter is altered.

`RegisterNotifyOnUpdate` and `DeregisterNotifyOnUpdate` shall both be Interface Only Implementations, or shall both be Full Implementations as designated by a subclass definition. The default shall be a Full Implementation.

10.9.3.4 arguments `time_type`, `epoch`, and `epoch_representation` behavior specifications

The allowed values for the `time_type`, `epoch_representation`, and `epoch` arguments shall be as shown in Table 79 and case statements.

Table 79—Time representation restrictions

Time type	Epoch Representation	Epoch
`TT_TIME_DURATION`	`TE_NOT_DEFINED`	Cases 1 and 2 shall apply
`TT_TIMESTAMP`	`TE_NOT_DEFINED`	Cases 1 and 3 shall apply
	`TE_UTC`	Cases 1 and 4 shall apply
	`TE_USER_DEFINED`	Case 5 shall apply
	(open to industry)	Case 6 shall apply

Case 1: The value of `epoch` is the default value for the `TimeRepresentation` type.

Case 2: The concept on an epoch is not relevant to a measure of the difference between two timestamps.

Case 3: The epoch is either undefined or not known. No interpretation of the value of the Parameter is possible on the basis of these values. The interpretation must be specified by other means outside the scope of this standard. This interpretation shall be documented by the creator of any code using this option.

Case 4: The epoch is the UTC epoch; see 13.2.1.7.

Case 5: The epoch is specified by the value of the `epoch` argument. The value of `epoch` is itself a timestamp relative to some origin selected by the user. It is recommended that the value of `epoch` be represented

Copyright © 2000 IEEE. All rights reserved.

on a UTC timescale. The interpretation of the epoch value shall be documented by the creator of any code using this option.

Case 6: The correct interpretation of the epoch and therefore of the value of the Parameter itself shall be documented by the creator of the extension to the `TimeEpoch` enumeration.

When the epoch is defined, the value of the Parameter represents a timestamp on a timescale whose origin is given by the epoch. In the case of the UTC epoch, this timescale origin is fixed by international standard; see [B7] for a complete discussion.

10.10 Action class

Class: `IEEE1451_Action`

Parent Class: `IEEE1451_Component`

Class ID: 1.1.2.2

Description: The Action class provides a model to represent activities that alter system state and that require significant time to execute compared to other activities in the system.

Class summary:

Network Visible operations (see Table 80).

Table 80—`IEEE1451_Action` Network Visible variables

Operation Name	Requirement	Operation ID
`GetActionFailedTimeoutDuration`	(m)	6168
`InvokeTransaction`	(m)	6169
`GetActionState`	(m)	6170
`AbortTransaction`	(m)	6171
`ReleaseTransaction`	(m)	6172
`ForceAcquireTransaction`	(m)	6173

Publications: There are no publications defined for this class.

Subscriptions: There are no subscriptions defined for this class.

10.10.1 Operation specifications: Network Visible

10.10.1.1 Operation `GetActionFailedTimeoutDuration` specification

```
IDL: OpReturnCode GetActionFailedTimeoutDuration(
          out TimeRepresentation
             action_failed_timeout_duration);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

 Copyright © 2000 IEEE. All rights reserved.

10.10.1.1.1 argument `action_failed_timeout_duration` specification

The `action_failed_timeout_duration` argument value shall be the time duration after which the Action is considered to have failed. A value of 0 shall indicate that the Action's timeout is infinite, that is the Action may continue indefinitely and still retain the intended behavior of the artifact modeled by the Action. A negative value shall be considered an argument error.

10.10.1.2 Operation `InvokeTransaction` specification

```
IDL: OpReturnCode InvokeTransaction(
          in ArgumentArray input_arguments,
          out UInteger16 transaction_id);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.10.1.2.1 argument `input_arguments` specification

The value of the Argument members of `input_arguments` shall be the input arguments for the Action's `InvokeTransaction` operation.

10.10.1.2.2 argument `transaction_id` specification

The `transaction_id` argument value shall identify a particular invocation of the Action.

10.10.1.3 Operation `GetActionState` specification

```
IDL: OpReturnCode GetActionState(out UInteger8 action_state);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.10.1.3.1 argument `action_state` specification

The `action_state` argument value shall be defined by the following enumeration, shown in Table 81, and shall reflect the current state of the Action.

IDL: enumeration `ActionState;`

Table 81—`ActionState` enumeration

Enumeration	Value	Meaning
`AC_IDLE`	0	Any invoked action has completed and has been released.
`AC_EXECUTING`	1	An invoked action is still in progress.
`AC_COMPLETED`	2	An invoked action has been completed, but not released by the client.
`AC_FAILED`	3	The Component has detected an error, the timeout has expired, or an `AbortTransaction` has been invoked.
Reserved	4–255	

Copyright © 2000 IEEE. All rights reserved.

10.10.1.4 Operation `AbortTransaction` specification

```
IDL: OpReturnCode AbortTransaction(in UInteger16 transaction_id);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.10.1.5 Operation `ReleaseTransaction` specification

```
IDL: OpReturnCode ReleaseTransaction(
           in UInteger16 transaction_id);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.10.1.6 Operation `ForceAcquireTransaction` specification

```
IDL: OpReturnCode ForceAcquireTransaction(
           out UInteger16 new_transaction_id);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.10.1.6.1 argument `new_transaction_id` specification

The `new_transaction_id` argument value shall identify a new value for the `transaction_id` of the Action.

10.10.2 Action class behavior

10.10.2.1 Specific Action operation restrictions based on state

In addition to and conformant with the restrictions of 10.1.2.2, the allowed operations defined on an Action instance shall be restricted as defined in Table 82. Where an operation is indicated as operational, it still shall be restricted to specific uses defined by the restrictions of 9.1.3.2.1, 9.1.3.2.2, and 9.1.3.2.3.

Table 82—`IEEE1451_Action` class specific restrictions

Operation	Specified value of `rule_basis`		
	BL_UNINITIALIZED	BL_INACTIVE	BL_ACTIVE
All operations defined by this class	Local invocation only	Operational	Operational

10.10.2.2 Concurrency requirements

The `AbortTransaction` and `ForceAcquireTransaction` operations shall support Precedence Concurrency.

The `GetActionState` operation shall support Active Concurrency.

10.10.2.3 State behavior specification

The state behavior of an Action shall be as shown in Figure 22.

 Copyright © 2000 IEEE. All rights reserved.

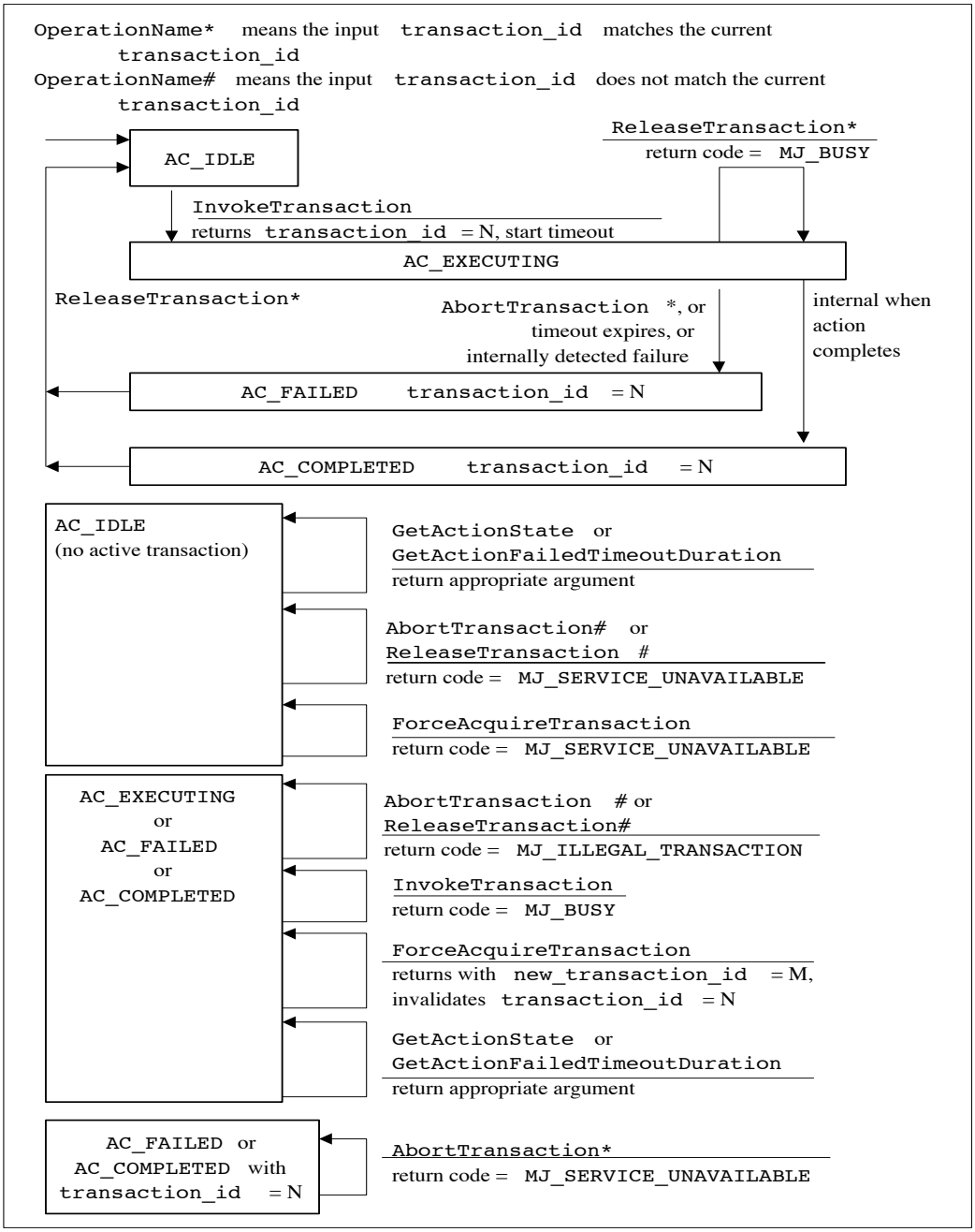

Figure 22—State machine for an Action

10.11 File class

Class: IEEE1451_File

Parent Class: IEEE1451_Component

Class ID: 1.1.2.3

Copyright © 2000 IEEE. All rights reserved.

Description: The File class is an abstraction of a data resource. Files represent a block of memory, which may be opened, closed, read from, and written to.

Class summary:

Network Visible operations (see Table 83).

Table 83—`IEEE1451_File` Network Visible operations

Operation Name	Requirement	Operation ID
`OpenForRead`	(o)	6174
`OpenForWrite`	(o)	6175
`Read`	(o)	6176
`Write`	(o)	6177
`Close`	(m)	6178
`ForceClose`	(m)	6179
`GetFileState`	(m)	6180

Publications: There are no publications defined for this class.

Subscriptions: There are no subscriptions defined for this class.

10.11.1 Operation specifications: Network Visible

10.11.1.1 Operation `OpenForRead` specification

```
IDL: OpReturnCode OpenForRead(
           out UInteger16 transaction_id,
           out UInteger32 actual_image_size);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.11.1.1.1 argument `transaction_id` specification

The `transaction_id` argument value shall define the client invocation that opened this File for reading. This value may be used to enforce the required File state behavior.

10.11.1.1.2 argument `actual_image_size` specification

The `actual_image_size` argument value shall be the number of octets in the current stored File image. This argument shall be the maximum number of octets that may be transferred by a successful sequence of contiguous `Read` operations.

10.11.1.2 Operation `OpenForWrite` specification

```
IDL: OpReturnCode OpenForWrite(
           out UInteger16 transaction_id,
           out UInteger32 max_file_size);
```

 Copyright © 2000 IEEE. All rights reserved.

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.11.1.2.1 argument `transaction_id` specification

The `transaction_id` argument value shall define the client invocation that opened this File for writing. This value is used to enforce the required File state behavior.

10.11.1.2.2 argument `max_file_size` specification

The `max_file_size` argument value shall be the maximum number of octets that may be transferred by a successful contiguous sequence of `Write` operations.

10.11.1.3 Operation `Read` specification

```
IDL: OpReturnCode Read(
           in UInteger16 transaction_id,
           in UInteger32 requested_number_of_octets,
           out UInteger32 actual_number_of_octets_read,
           out OctetArray data);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.11.1.3.1 argument `transaction_id` specification

The `transaction_id` argument value identifies a particular instance of an `OpenForRead` operation.

10.11.1.3.2 argument `requested_number_of_octets` specification

The `requested_number_of_octets` argument value shall be the number of octets that the client is requesting from the File Object.

10.11.1.3.3 argument `actual_number_of_octets_read` specification

The `actual_number_of_octets_read` argument value shall indicate the number of octets actually returned to the client.

10.11.1.3.4 argument `data` specification

The `data` argument value shall be the portion of the File's data returned to the client.

10.11.1.4 Operation `Write` specification

```
IDL: OpReturnCode Write(
           in UInteger16 transaction_id,
           in UInteger32 requested_number_of_octets,
           in OctetArray data,
           out UInteger32 actual_number_of_octets_written);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.11.1.4.1 argument `transaction_id` specification

The `transaction_id` argument value identifies a particular instance of an `OpenForWrite` operation.

Copyright © 2000 IEEE. All rights reserved.

10.11.1.4.2 argument `requested_number_of_octets` specification

The `requested_number_of_octets` argument value shall be the number of octets that the client is requesting to write to the File Object.

10.11.1.4.3 argument `data` specification

The `data` argument value shall be the File contents information sent to the File Object.

10.11.1.4.4 argument `actual_number_of_octets_written` specification

The `actual_number_of_octets_written` argument value shall indicate the number of octets actually accepted and written by the File Object.

10.11.1.5 Operation `Close` specification

```
IDL: OpReturnCode Close(in UInteger16 transaction_id);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.11.1.6 Operation `ForceClose` specification

```
IDL: OpReturnCode ForceClose( );
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.11.1.7 Operation `GetFileState` specification

```
IDL: OpReturnCode GetFileState(
           out UInteger8 file_state,
           out UInteger32 actual_image_size,
           out UInteger32 max_file_size);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.11.1.7.1 argument `file_state` specification

The `file_state` argument value shall be defined by the enumeration shown in Table 84.

IDL: enumeration `FileState;`

Table 84—`FileState` enumeration

Enumeration	Value	Meaning
`FL_CLOSED`	0	This File is closed.
`FL_OPENED_FOR_READ`	1	The File has been opened for `Read` operations.
`FL_OPENED_FOR_WRITE`	2	The File has been opened for `Write` operations.
Reserved	3–255	

 Copyright © 2000 IEEE. All rights reserved.

10.11.2 File class behavior

10.11.2.1 Specific File operation restrictions based on state

In addition to and conformant with the restrictions of 10.1.2.2, the allowed operations defined on a File instance shall be restricted as defined in Table 85. Where an operation is indicated as operational, it still shall be restricted to specific uses defined by the restrictions of 9.1.3.2.1, 9.1.3.2.2, and 9.1.3.2.3.

Table 85—`IEEE1451_File` class specific restrictions

Operation	Specified value of `rule_basis`		
	BL_UNINITIALIZED	**BL_INACTIVE**	**BL_ACTIVE**
All operations defined by this class	Local invocation only	Operational	Operational

10.11.2.2 Concurrency requirements

The `ForceClose` operation may support Precedence Concurrency.

10.11.2.3 Operations `OpenForRead` and `Read` behavior specifications

The `OpenForRead` and `Read` operations may either both be Interface Only Implementations if the File is "write only," or they shall both be Full Implementations.

With a Full Implementation, the successful execution of the `OpenForRead` operation shall cause the first subsequent `Read` operation to start at the beginning of the File. Additional `Read` operations return the requested block of data in the File that immediately follows the last data returned.

The `actual_number_of_octets_read` argument, returned from a `Read` operation, shall be zero, if either of the following conditions is `TRUE:`

— The `requested_number_of_octets` is zero

— The total number of octets transferred by previous `Read` operations equals `max_file_size` for the File

10.11.2.4 Operations `OpenForWrite` and `Write` behavior specifications

The `OpenForWrite` and `Write` operations may either both be Interface Only Implementations if the File is "read only," or they shall both be Full Implementations.

With a Full Implementation, the successful execution of the `OpenForWrite` operation shall cause the first subsequent `Write` operation to start at the beginning of the File. The first `Write` operation shall invalidate any data initially present in the File Object. Additional `Write` operations append the data to the File.

The `actual_number_of_octets_written` argument, returned from a `Write` operation, shall be zero if either of the following conditions is `TRUE:`

— The `requested_number_of_octets` is zero

— The total number of octets transferred by previous `Write` operations equals `max_file_size` for the File specified by the `OpenForWrite` operation

Copyright © 2000 IEEE. All rights reserved.

10.11.2.5 State behavior specification

The File behavior shall be as described by the state machine of Figure 23. Operations not defined on a state shall have no effect. The initial state of a File shall be `FL_CLOSED` as indicated by the entry transition of the state machine.

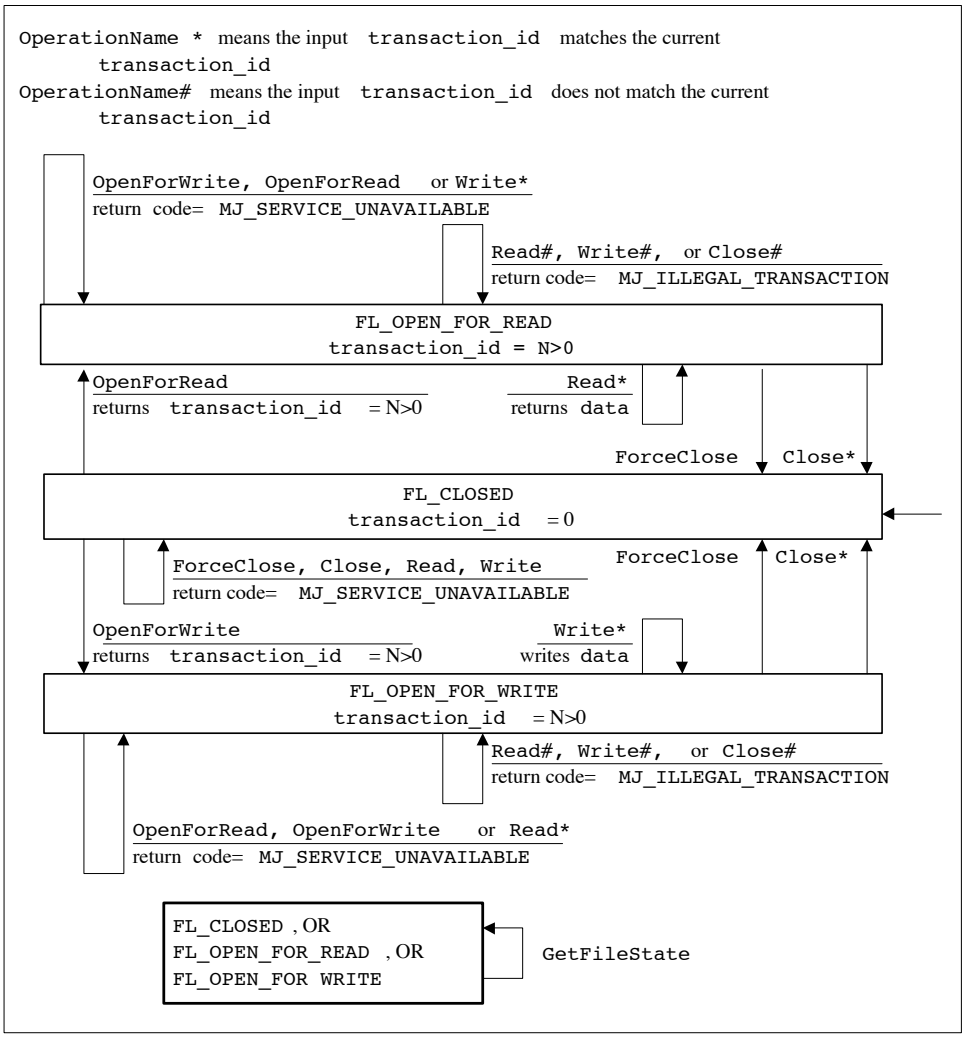

Figure 23—State machine for a File

10.11.2.5.1 argument `transaction_id` behavior specification

A valid `transaction_id` shall have the value NO_TRANSACTION following the successful execution of a `Close` or `ForceClose` operation.

The NO_TRANSACTION value shall be 0.

Two `transaction_id` values match if, and only if

— They have the same value, **AND**

— Neither is the NO_TRANSACTION value

 Copyright © 2000 IEEE. All rights reserved.

10.11.2.6 Persistence behavior specification

The persistence properties of a File are outside the scope of this standard unless otherwise noted.

10.12 Partitioned File class

Class: `IEEE1451_PartitionedFile`

Parent Class: `IEEE1451_File`

Class ID: 1.1.2.3.1

Description: The Partitioned File class may be used for Files that are subdivided into a number of Partitions.

Class summary:

Network Visible operations (see Table 86).

Table 86—`IEEE1451_PartitionedFile` Network Visible operations

Operation Name	Requirement	Operation ID
`SeekPartition`	(m)	8212
`GetCurrentPartition`	(m)	8213
`GetNumberOfPartitions`	(m)	8214
`GetPartitionedFileSubstate`	(m)	8215

Publications: There are no publications defined for this class.

Subscriptions: There are no subscriptions defined for this class.

10.12.1 Operation specifications: Network Visible

10.12.1.1 Operation `SeekPartition` specification

```
IDL: OpReturnCode SeekPartition(
            in UInteger16 transaction_id,
            in UInteger16 partition_id,
            out UInteger32 actual_partition_image_size,
            out UInteger32 max_partition_size);
```

The Minor Field of the return code shall be taken from the enumeration shown in Table 87.

IDL: enumeration `PartitionReturn;`

The meanings of the first six members of this enumeration shall be identical to the meanings of the first six members of the `MinorReturnCode` enumeration; see 7.2.3.2.1. The member `PR_MISSING_PARTITION` shall signify that the Partition with the specified `partition_id` does not exist or could not be found.

Copyright © 2000 IEEE. All rights reserved.

Table 87—`PartitionReturn` enumeration

Enumeration	Value
PR_NO_ADDITIONAL_INFORMATION	0
PR_MEMORY_ALLOCATION_ERROR	1
PR_MISSING_INPUT	2
PR_INVALID_TYPE	3
PR_INVALID_VALUE	4
PR_COMPUTATION_ERROR	5
PR_MISSING_PARTITION	6
Reserved	7–255

10.12.1.1.1 argument `partition_id` specification

The `partition_id argument` value shall represent a Partitioned File Object specific identifier of some subportion of the File's data space.

10.12.1.1.2 argument `actual_partition_image_size` specification

The `actual_partition_image_size` argument value shall be the number of octets currently stored in the Partition.

10.12.1.1.3 argument `max_partition_size` specification

The `max_partition_size` argument value shall be the maximum number of octets that can be stored in the Partition.

10.12.1.2 Operation `GetCurrentPartition` specification

```
IDL: OpReturnCode GetCurrentPartition(
           in UInteger16 transaction_id,
           out UInteger16 partition_id,
           out UInteger32 actual_partition_image_size,
           out UInteger32 max_partition_size);
```

The Minor Field of the return code shall be taken from the `PartitionReturn` enumeration.

10.12.1.3 Operation `GetNumberOfPartitions` specification

```
IDL: OpReturnCode GetNumberOfPartitions(
           in UInteger16 transaction_id,
           out UInteger16 number_of_partitions);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.12.1.3.1 argument `number_of_partitions` specification

The `number_of_partitions` argument value shall be the total number of Partitions of this File that can be accessed.

 Copyright © 2000 IEEE. All rights reserved.

10.12.1.4 Operation `GetPartitionedFileSubstate` specification

```
IDL: OpReturnCode GetPartitionedFileSubstate(
           in UInteger16 transaction_id,
           out UInteger8 partition_state);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.12.1.4.1 argument `partition_state` specification

The `partition_state` argument value shall be take from the `PartitionedFileSubstate` enumeration shown in Table 88.

IDL: enumeration `PartitionedFileSubstate;`

Table 88—`PartitionedFileSubstate` enumeration

Enumeration	Value	Meaning
`PL_PARTITION_NOT_SET`	0	Partition not in readable or writable state
`PL_PARTITION_SET`	1	Partition is available for reading or writing
Reserved	2–255	

10.12.2 Partitioned File class behavior

10.12.2.1 Specific Partitioned File operation restrictions based on state

In addition to and conformant with the restrictions of 10.11.2.1, the allowed operations defined on a Partitioned File instance shall be restricted as defined in the Table 89. Where an operation is indicated as operational, it still shall be restricted to specific uses defined by the restrictions of9.1.3.2.1, 9.1.3.2.2, and 9.1.3.2.3.

Table 89—`IEEE1451_PartitionedFile` class specific restrictions

Operation	Specified value of `rule_basis`		
	BL_UNINITIALIZED	BL_INACTIVE	BL_ACTIVE
All operations defined by this class	Local invocation only	Operational	Operational

10.12.2.2 Partitioning behavior specification

The File consists of `number_of_partitions` subdivisions that may be addressed via the `SeekPartition` operation.

— The value of `number_of_partitions` shall be set at class instantiation.

— The assignment of `partition_id` values to the corresponding portions of the partitioned file shall be made at class instantiation.

— Valid `partition_id` values shall be in the range

$0 \le$ `partition_id` $<$ `number_of_partitions`

Copyright © 2000 IEEE. All rights reserved.

— Void partitions, `max_partition_size` = 0, are allowed but can not be written to or read from.

— The following empty Partition condition is allowed, and such Partitions can be written to, or read from:

`max_partition_size` > 0

AND

`actual_partition_image_size` = 0

— Missing `partition_id` values within the range shall not be allowed.

A successful execution of the `SeekPartition` operation shall cause the first subsequent `Read` or `Write` operation (depending on whether the File was opened for reading or writing) to start at the beginning of the Partition.

10.12.2.3 Inherited operation `OpenForRead` behavior specification

The `actual_image_size` argument of the `OpenForRead` operation inherited from the File class shall be the sum of the `actual_partition_image_size` values for all Partitions in the File.

The `OpenForRead` and `Read` operations may either both be Interface Only Implementations if the File is "write only," or they shall both be Full Implementations.

With a Full Implementation, the successful execution of the `OpenForRead` operation followed by the successful execution of the `SeekPartition` operation shall cause the first subsequent `Read` operation to start at the beginning of the designated Partition of the File. Additional `Read` operations return the requested block of data in the designated Partition of the File that immediately follows the last data returned.

The `actual_number_of_octets_read` argument, returned from a `Read` operation, shall be zero, if either of the following conditions is `TRUE`:

— The `requested_number_of_octets` is zero.

— The total number of octets transferred by previous `Read` operations on the Partition equals `max_partition_size` for the Partition.

10.12.2.4 Inherited operation `OpenForWrite` behavior specification

The `max_file_size` argument of the `OpenForWrite` operation inherited from the File class shall be the sum of the `max_partition_size` values for all Partitions in the File.

The `OpenForWrite` and `Write` operations may either both be Interface Only Implementations if the File is "read only," or they shall both be Full Implementations.

With a Full Implementation, the successful execution of the `OpenForWrite` operation followed by the successful execution of the `SeekPartition` operation shall cause the first subsequent `Write` operation to start at the beginning of the designated Partition of the File. Additional `Write` operations transfer the requested block of data to the designated Partition of the File that immediately follows the last data written.

The `actual_number_of_octets_written` argument, returned from a `Write` operation, shall be zero, if either of the following conditions is `TRUE`:

— The `requested_number_of_octets` is zero.

 Copyright © 2000 IEEE. All rights reserved.

— The total number of octets transferred by previous `Write` operations on the Partition equals `max_partition_size` for the Partition.

10.12.2.5 State behavior specification

The File behavior shall be as described by Figure 24. Operations not defined on a state shall have no effect.

If a `SeekPartition` operation is unsuccessful in placing the File such that the next `Read` or `Write` will operate starting at the beginning of the new Partition, failure shall be signaled by means of the `OpReturnCode` and the File Object shall be returned to the `PL_PARTITION_NOT_SET` state. In this case the `OpReturnCode`

— Major Field value shall be `MJ_FAILED_INPUT_ARGUMENT`

— Minor Field value shall be `PR_MISSING_PARTITION`

10.13 Component Group class

Class: `IEEE1451_ComponentGroup`

Parent Class: `IEEE1451_Component`

Class ID: 1.1.2.4

Description: The Component Group class provides a way to specify set membership relations between Objects in a system.

Class summary:

Network Visible operations (see Table 90).

Table 90—`IEEE1451_ComponentGroup` Network Visible operations

Operation Name	Requirement	Operation ID
`AddMember`	(m)	6181
`DeleteMember`	(m)	6182
`GetMembers`	(m)	6183
`SetMembers`	(m)	6184
`LookupMembersByName`	(m)	6185
`LookupMemberByDispatchAddress`	(m)	6186
`LookupMemberByObjectTag`	(m)	6187
`GetNumberOfMembers`	(m)	6188
`GetNextMember`	(m)	6189
`CancelIteration`	(m)	6190

Publications: There are no publications defined for this class.

Subscriptions: There are no subscriptions defined for this class.

Copyright © 2000 IEEE. All rights reserved.

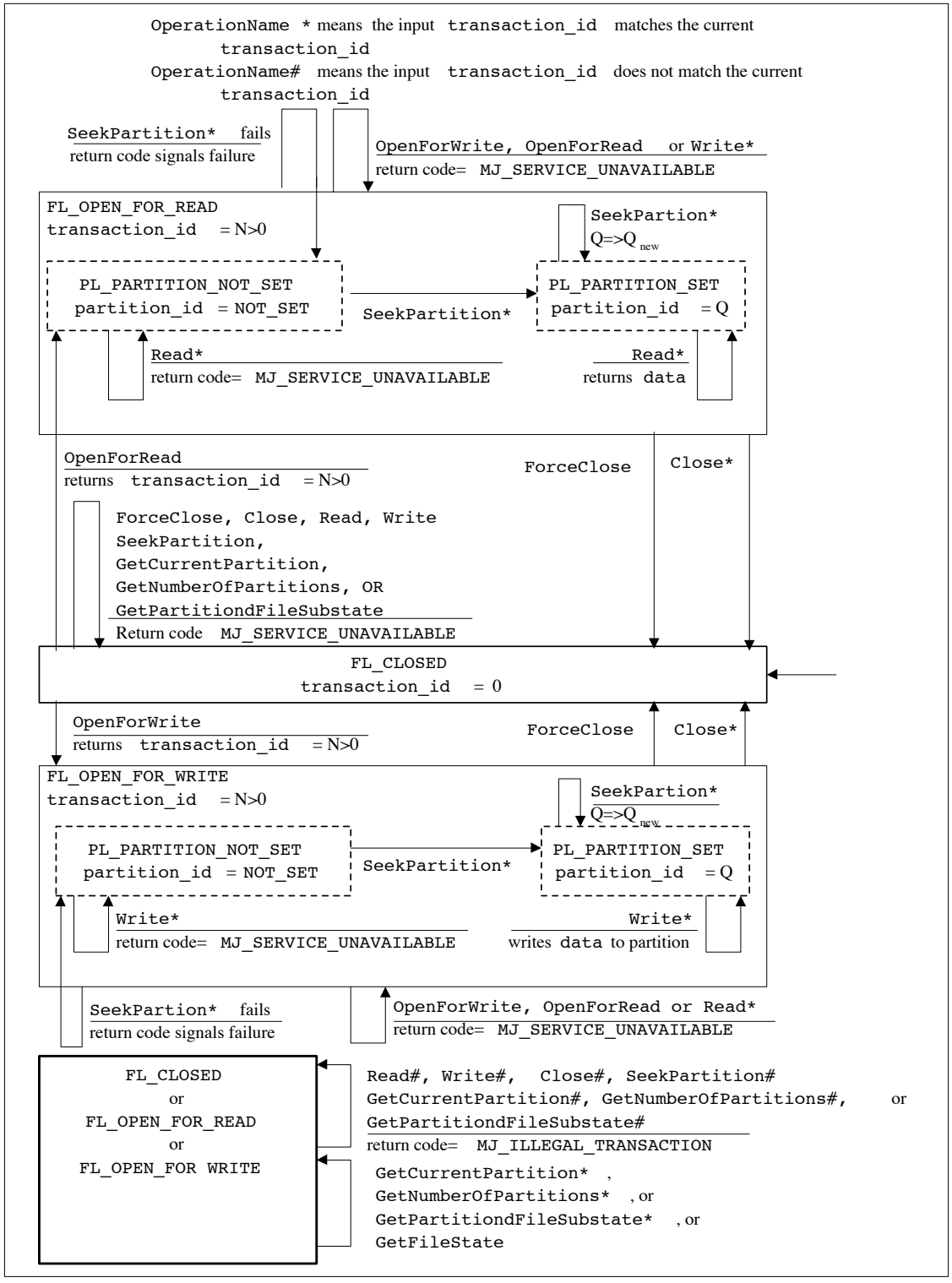

Figure 24—State machine for a Partitioned File

Copyright © 2000 IEEE. All rights reserved.

10.13.1 Operation specifications: Network Visible

10.13.1.1 Operation `AddMember` specification

```
IDL: OpReturnCode AddMember(
           in ObjectProperties member_properties);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.13.1.1.1 argument `member_properties` specification

The values of the fields of the `member_properties` structure shall be those of the Object being added to the group.

10.13.1.2 Operation `DeleteMember` specification

```
IDL: OpReturnCode DeleteMember(
           in ObjectProperties member_properties);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.13.1.2.1 argument `member_properties` specification

The values of the fields of the `member_properties` structure shall be those of the Object being deleted from the group.

10.13.1.3 Operation `GetMembers` specification

```
IDL: OpReturnCode GetMembers(
           out ObjectPropertiesArray members_properties);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.13.1.3.1 argument `members_properties` specification

Each member of the Array shall be an `ObjectProperties` structure for a group member.

10.13.1.4 Operation `SetMembers` specification

```
IDL: OpReturnCode SetMembers(
           in ObjectPropertiesArray members_properties);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.13.1.4.1 argument `members_properties` specification

Each member of the Array shall be an `ObjectProperties` structure for a member to be added to the group.

10.13.1.5 Operation `LookupMembersByName` specification

```
IDL: OpReturnCode LookupMembersByName(
           in String query_object_name,
           out ObjectPropertiesArray member_properties);
```

Copyright © 2000 IEEE. All rights reserved.

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.13.1.5.1 argument `query_object_name` specification

The `query_object_name` argument value shall be the value of the Object Name of the Object to be sought within the group.

10.13.1.5.2 argument `member_properties` specification

Each member of the Array `member_properties` shall be an `ObjectProperties` structure describing a member Object with

`objectName` member value `= query_object_name`

NOTE—There is no requirement for global uniqueness of Object Names. Therefore, it is possible for multiple members to have the same `objectName` member values.

10.13.1.6 Operation `LookupMemberByDispatchAddress` specification

```
IDL: OpReturnCode LookupMemberByDispatchAddress(
          in ObjectDispatchAddress query_dispatch_address,
          out ObjectPropertiesArray member_properties);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.13.1.6.1 argument `query_dispatch_address` specification

The `query_dispatch_address` argument value shall be the Object Dispatch Address of the Object to be sought within the group.

10.13.1.6.2 argument `member_properties` specification

The `member_properties` argument value shall be an `ObjectProperties` structure describing the Object with

Object Dispatch Address `= query_dispatch_address.`

10.13.1.7 Operation `LookupMemberByObjectTag` specification

```
IDL: OpReturnCode LookupMemberByObjectTag(
          in ObjectTag query_object_tag,
          out ObjectPropertiesArray member_properties);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.13.1.7.1 argument `query_object_tag` specification

The value of the argument `query_object_tag` shall be the Object Tag of the Object to be sought within the group.

 Copyright © 2000 IEEE. All rights reserved.

10.13.1.7.2 argument `member_properties` specification

Each `member_properties argument` shall be an `ObjectProperties` structure describing an Object with

Object Tag `= query_object_tag.`

10.13.1.8 Operation `GetNumberOfMembers` specification

```
IDL: OpReturnCode GetNumberOfMembers(
          out UInteger16 number_of_members);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.13.1.8.1 argument `number_of_members` specification

The `number_of_members` argument value shall be the number of Objects represented by the group.

10.13.1.9 Operation `GetNextMember` specification

```
IDL: OpReturnCode GetNextMember(
          in Boolean first_flag
          out ObjectProperties member_properties);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.13.1.9.1 argument `first_flag` specification

If the `first_flag` argument value is `TRUE`, the returned member is conceptually the first member of the group. It the argument is `FALSE`, the returned member is the next successive member.

NOTE—There is no specification on the order in which members are returned by `GetNextMember`. Successive iterations over the group membership may retrieve the members in a different order; see 10.13.2.2.

10.13.1.9.2 argument `member_properties` specification

The `member_properties` argument value shall be an `ObjectProperties` structure describing the member of the series. The default value for `ObjectProperties` shall indicate that there was no member.

10.13.1.9.3 Operation `CancelIteration` specification

```
IDL: OpReturnCode CancelIteration( );
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

10.13.2 Component Group class behavior

10.13.2.1 Specific Component Group operation restrictions based on state

In addition to and conformant with the restrictions of 10.1.2.2, the allowed operations defined on a Component Group instance shall be restricted as defined in Table 91. Where an operation is indicated as operational, it still shall be restricted to specific uses defined by the restrictions of 9.1.3.2.1, 9.1.3.2.2, and 9.1.3.2.3.

Copyright © 2000 IEEE. All rights reserved.

Table 91—`IEEE1451_ComponentGroup` class specific restrictions

Operation	Specified value of `rule_basis`		
	BL_UNINITIALIZED	**BL_INACTIVE**	**BL_ACTIVE**
All operations defined by this class	Local invocation only	Operational	Operational

10.13.2.2 State behavior specification

The state behavior of the Component Group shall be defined by the state machine in Figure 25. The initial state shall be `NON_ITERATION_MODE` as illustrated by the entry transition of the state machine. `First_flag = TRUE` implies that the iteration over the members of the group is starting. There is no requirement that the order of members returned be related to the order in which members were added to the group. Applications must not assume any dependence on the order in which members' `member_properties` are returned.

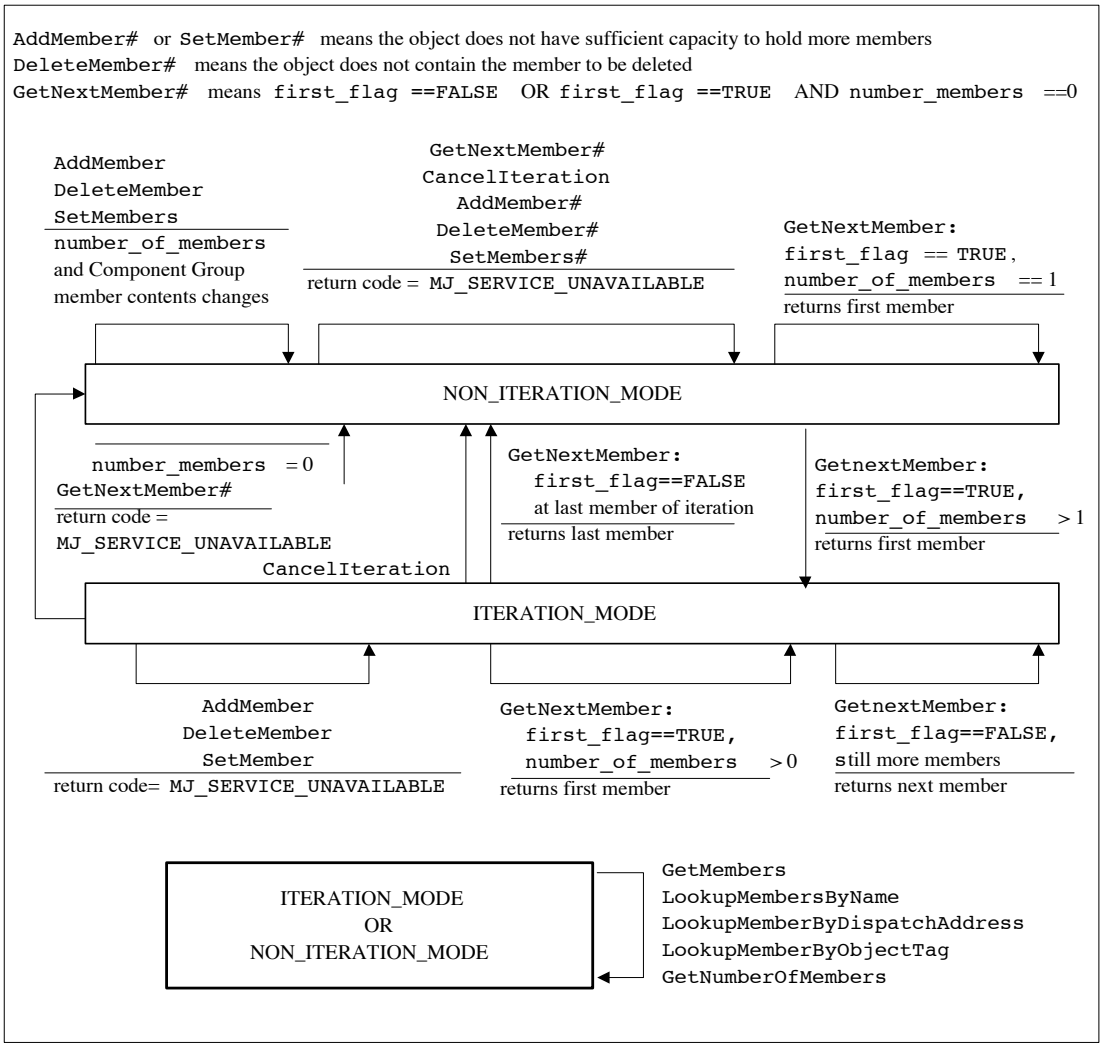

Figure 25—State machine for a Component Group

Copyright © 2000 IEEE. All rights reserved.

11. Service classes

11.1 Service abstract class

Abstract Class

Class: `IEEE1451_Service`

Parent Class: `IEEE1451_Entity`

Class ID: 1.1.3

Description: The Service abstract class shall be the root for the class hierarchy of all Service Objects. The Service classes represent Object types used to support communication and other aspects of block functionality.

Class summary:

Network Visible operations: There are no operations defined for this class.

Local Operations (see Table 92).

Table 92—`IEEE1451_Service` Local operations

Operation Name	Requirement
`SpecifyRuleBasis`	(m)

Publications: There are no publications defined for this class.

Subscriptions: There are no subscriptions defined for this class.

11.1.1 Operation specifications: Local

11.1.1.1 Local operation `SpecifyRuleBasis` specification

```
IDL: OpReturnCode SpecifyRuleBasis(in UInteger8 rule_basis);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.1.1.2 argument `rule_basis` specification

The `rule_basis` argument value shall specify the set of behavior rules governing the Object's behavior. The value of this argument shall be taken from the `BlockMajorState` enumeration; see 9.1.1.3.1.

11.1.2 Service class behavior

11.1.2.1 Operation `SpecifyRuleBasis` behavior specification

The `SpecifyRuleBasis` operation invoked on an Object T whose class is a subclass of Service shall be defined by the following rule:

Copyright © 2000 IEEE. All rights reserved.

A successful execution of the operation shall cause the Object T to obey the set of behavior restrictions designated by the argument `rule_basis`. These sets of restrictions are those defined for each of the Object's Owning Block's Block Major State values; see 9.1.3.2.1, 9.1.3.2.2, 9.1.3.2.3, or 9.2.5.2.1, depending on whether the Owning Block is an NCAP Block or only a Block.

The designation of the rule sets shall be as defined in Table 93.

Table 93—`rule_basis` restrictions

Rule basis value	Use restriction set defined for Block Major State value
BL_UNINITIALIZED	BL_UNINITIALIZED
BL_INACTIVE	BL_INACTIVE
BL_ACTIVE	BL_ACTIVE

The operation shall only be invoked under conditions defined in Table 94.

Table 94—`SpecifyRuleBasis` restrictions

`rule_basis` value	Block major state of Owning Block of Object T		
	BL_UNINITIALIZED	BL_INACTIVE	BL_ACTIVE
BL_UNINITIALIZED	Allowed	Allowed	Allowed
BL_INACTIVE	Not allowed	Allowed	Allowed
BL_ACTIVE	Not allowed	Not allowed	Allowed

NOTE—A common use for this operation is to promote orderly startup and shutdown of a Block and its Owned objects. For example, on startup a Block in the `BL_INACTIVE` state would first transition to the `BL_ACTIVE` state and then issue the `SpecifyRuleBasis` operation with a `rule_basis` value of `BL_ACTIVE` to each of its Owned Service objects. During shutdown, the Block in the `BL_ACTIVE` state would first issue the `SpecifyRuleBasis` operation with a `rule_basis` value of `BL_INACTIVE` to each of its Owned Service objects before transitioning to the `BL_INACTIVE` state.

11.1.2.2 Specific Service operation restrictions based on state

In addition to and conformant with the restrictions of 11.1.2.1, the allowed operations defined on a Service instance shall be restricted as defined in Table 95. Where an operation is indicated as operational, it still shall be restricted to specific uses defined by the restrictions of 9.1.3.2.1, 9.1.3.2.2, and 9.1.3.2.3.

Table 95—`IEEE1451_Service` class specific restrictions

Operation	Specified value of `rule_basis`		
	BL_UNINITIALIZED	BL_INACTIVE	BL_ACTIVE
All operations defined by this class except those in the following rows	Local invocation only	Operational	Operational
`SetObjectTag`	Local invocation only	Operational	Not operational
`Perform`	Not operational	Operational	Operational

 Copyright © 2000 IEEE. All rights reserved.

11.2 Base Port abstract class

Abstract Class

Class: `IEEE1451_BasePort`

Parent Class: `IEEE1451_Service`

Class ID: 1.1.3.1

Description: The Base Port abstract class shall be the root for the class hierarchy of all communication Port Objects used to send communications via the underlying network.

Class summary:

Network Visible operations (see Table 96).

Table 96—`IEEE1451_BasePort` Network Visible operations

Operation Name	Requirement	Operation ID
`SetTimeout`	(o)	6191
`GetTimeout`	(m)	6192
`SetMessagePriority`	(o)	6193
`GetMessagePriority`	(m)	6194

Publications: There are no publications defined for this class.

Subscriptions: There are no subscriptions defined for this class.

11.2.1 Operation specification: Network Visible

11.2.1.1 Operation `SetTimeout` specification

```
IDL: OpReturnCode SetTimeout(
           in TimeRepresentation client_timeout);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.2.1.2 argument `client_timeout` specification

The `client_timeout` argument value shall be the timeout associated with the Port. A negative timeout shall be considered an error.

11.2.1.3 Operation `GetTimeout` specification

```
IDL: OpReturnCode GetTimeout(
           out TimeRepresentation client_timeout);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

Copyright © 2000 IEEE. All rights reserved.

11.2.1.4 Operation `SetMessagePriority` specification

```
IDL: OpReturnCode SetMessagePriority(
          in UInteger8 message_priority);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.2.1.4.1 argument `message_priority` specification

The `message_priority` argument value shall be the `message_priority` for all client-server or publish-subscribe interactions emanating from the Port. The allowed values of `message_priority` shall be as defined in the enumeration shown in Table 97.

IDL: enumeration `MessagePriority;`

Table 97—`MessagePriority` enumeration

Enumeration	Value
MP_NORMAL	0
MP_EXPEDITE	1
Reserved	2–127
Open to industry	128–255

11.2.1.5 Operation `GetMessagePriority` specification

```
IDL: OpReturnCode GetMessagePriority(
          in UInteger8 message_priority);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.2.2 Base Port abstract class behavior

11.2.2.1 Specific Base Port operation restrictions based on state

In addition to and conformant with the restrictions of 11.1.2.2, the allowed operations defined on a Base Port shall be restricted as defined in Table 98. Where an operation is indicated as operational, it still shall be restricted to specific uses defined by the restrictions of 9.1.3.2.1, 9.1.3.2.2, and 9.1.3.2.3.

Table 98—`IEEE1451_BasePort` class specific restrictions

Operation	Specified value of `rule_basis`		
	BL_UNINITIALIZED	BL_INACTIVE	BL_ACTIVE
SetTimeout, SetMessagePriority	Local invocation only	Operational	Not operational
GetTimeout, GetMessagePriority	Local invocation only	Operational	Operational

 Copyright © 2000 IEEE. All rights reserved.

11.2.2.2 Timeout behavior specification

Timeout behavior shall be subclass-specific with the following restrictions:

— The default value for `client_timeout` shall be 0

— A timeout value of 0 shall define a timeout that never expires

The operation `SetTimeout` may be an Interface Only Implementation if the underlying operating system does not support timeouts; otherwise, this operation shall have a Full Implementation.

With an Interface Only Implementation of `SetTimeout`, `GetTimeout` shall return the default value.

11.2.2.3 Message priority behavior specification

The `message_priority` shall define the urgency of message delivery requested of the underlying communication structure. The default value of `message_priority` shall be `MP_NORMAL`.

The operation `SetMessagePriority` may be an Interface Only Implementation if the underlying network infrastructure does not support delivery priority; otherwise, this operation shall have a Full Implementation.

With an Interface Only Implementation of `SetMessagePriority`, `GetMessagePriority` shall return the default value.

A `message_priority` of `MP_NORMAL` shall cause the network communication implementation to deliver messages from this Port without preference compared to other messages.

A `message_priority` of `MP_EXPEDITE` shall cause the network communication implementation to deliver messages from this Port with preference compared to normal priority messages.

11.3 Base Client Port abstract class

Abstract Class

Class: `IEEE1451_BaseClientPort`

Parent Class: `IEEE1451_BasePort`

Class ID: 1.1.3.1.1

Description: The Base Client Port abstract class shall be the root for the class hierarchy of all client-server communication client-side Port Objects.

Class summary:

Network Visible operations (see Table 99).

Table 99—`IEEE1451_BaseClientPort` Network Visible operations

Operation Name	Requirement	Operation ID
`SetServerObjectTag`	(m)	8216
`GetServerObjectTag`	(m)	8217

Copyright © 2000 IEEE. All rights reserved.

Publications: There are no publications defined for this class.

Subscriptions: There are no subscriptions defined for this class.

11.3.1 Operation specifications: Network Visible

11.3.1.1 Operation `SetServerObjectTag` specification

```
IDL: OpReturnCode SetServerObjectTag(
           in ObjectTag server_object_tag);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.3.1.1.1 argument `server_object_tag` specification

The `server_object_tag` argument value shall be the Object Tag associated with the remote Object, which is the intended target of an execute semantics operation on the Port.

11.3.1.2 Operation `GetServerObjectTag` specification

```
IDL: OpReturnCode GetServerObjectTag(
           out ObjectTag server_object_tag);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.3.2 Base Client Port abstract class behavior

11.3.2.1 Specific Base Client Port operation restrictions based on state

In addition to and conformant with the restrictions of 11.2.2.1, the allowed operations defined on a Base Client Port shall be restricted as defined in Table 100. Where an operation is indicated as operational, it still shall be restricted to specific uses defined by the restrictions of 9.1.3.2.1, 9.1.3.2.2, and 9.1.3.2.3.

Table 100—`IEEE1451_BaseClientPort` class specific restrictions

Operation	Specified value of `rule_basis`		
	BL_UNINITIALIZED	BL_INACTIVE	BL_ACTIVE
`SetServerObjectTag`	Local invocation only	Operational	Not operational
`GetServerObjectTag`	Local invocation only	Operational	Operational

11.3.2.2 Communication behavior specification

The Port shall initiate an operation invocation request via the network infrastructure with the Server Object whose Object Dispatch Address has been provided based on the Server Object Tag of the Base Client Port. The Base Client Port's Server Object Tag shall be the Network Visible identifier that defines the target Object for client-server communications.

NOTE—NCAP Blocks provide a mechanism for identifying and providing to the Base Client Port, the Object Dispatch Address that corresponds to the Server Object Tag; see 9.2.5.6.

 Copyright © 2000 IEEE. All rights reserved.

11.3.2.3 Block Cookie behavior specification

The network infrastructure, as part of the invocation of an operation on the server, shall be responsible for delivering the latest client-side received value for the server's Block Cookie. This value is termed the Client Cached Block Cookie. If a value is not available, the default Block Cookie value, NOT_SET (see 7.3.2.2) shall be delivered.

NOTE—This latter circumstance can occur if the Base Client Port is redirected to a different server and no value of the Block Cookie was provided as part of the redirection and no client-server call has yet been made.

The initial value of the target Server Object's Block Cookie may be provided to the Base Client Port's Local NCAP Block as part of the configuration process by means of the `SetClientPortServerObjectBindings` operation; see 9.2.5.6.

For each client-server interaction, the target Server Object returns the current value of its Associated Block Cookie to the client-side network infrastructure; see 8.2.3.2.

The initial value of the Client Cached Block Cookie shall be updated to the latest available value of the Server Object's Associated Block Cookie.

11.4 Client Port class

Class: `IEEE1451_ClientPort`

Parent Class: `IEEE1451_BaseClientPort`

Class ID: 1.1.3.1.1.1

Description: The Client Port class provides the client-side application interface to client-server communications. The class abstracts the details of the specific network. Two models of client-server communication are provided: blocking and "send and forget."

Class summary:

Local Operations (see Table 101).

Table 101—`IEEE1451_ClientPort` Local operations

Operation Name	Requirement
`Execute`	(m)

Publications: There are no publications defined for this class.

Subscriptions: There are no subscriptions defined for this class.

11.4.1 Operation specifications: Local

11.4.1.1 Local operation `Execute` specification

```
IDL: ClientServerReturnCode Execute(
           in UInteger8 execute_mode,
```

Copyright © 2000 IEEE. All rights reserved.

```
in UInteger16 server_operation_id,
in ArgumentArray server_input_arguments,
out ArgumentArray server_output_arguments);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.4.1.1.1 argument `execute_mode` specification

The `execute_mode` argument value shall specify the return behavior of the clients call to `Execute`. The allowed values of `execute_mode` shall be as defined in the enumeration shown in Table 102.

IDL: enumeration `ExecuteMode;`

Table 102—`ExecuteMode` enumeration

Enumeration	Value
EM_RETURN_VALUE	0
EM_NO_RETURN_VALUE	1
Reserved	2–255

11.4.1.1.2 argument `server_operation_id` specification

The `server_operation_id` argument value is the Operation ID of the operation to be invoked on the Server Object.

11.4.1.1.3 argument `server_input_arguments` specification

The `server_input_arguments` argument value shall be the input arguments of the server operation, corresponding to the argument `server_operation_id`, encoded into an Argument Array.

11.4.1.1.4 argument `server_output_arguments` specification

This `server_output_arguments` argument value shall be a reference to an Argument Array into which the output arguments of the server operation, corresponding to the argument `server_operation_id,` shall be encoded.

11.4.2 Client Port class behavior

11.4.2.1 Specific Client Port operation restrictions based on state

In addition to and conformant with the restrictions of 11.3.2.1, the allowed operations defined on a Client Port shall be restricted as defined in Table 103. Where an operation is indicated as operational, it still shall be restricted to specific uses defined by the restrictions of 9.1.3.2.1, 9.1.3.2.2, and 9.1.3.2.3.

Table 103—`IEEE1451_ClientPort` class specific restrictions

Operation	Specified value of `rule_basis`		
	BL_UNINITIALIZED	BL_INACTIVE	BL_ACTIVE
Execute	Not operational	Operational	Operational

 Copyright © 2000 IEEE. All rights reserved.

11.4.2.2 Operation `Execute` behavior specification

The blocking behavior of the `Execute` call shall depend on the following possible values of `execute_mode`:

— `EM_RETURN_VALUE`

— `EM_NO_RETURN_VALUE`

11.4.2.2.1 Behavior specification in the `EM_RETURN_VALUE` state

The client shall be blocked, that is, execution of client code does not proceed, until the returned information has been delivered to the client, that is, the `Execute` call returns, or the `client_timeout` value for the Port has been exceeded

The Client Cached Block Cookie value shall be updated; see 11.3.2.3.

After the expiration of the client's timeout mechanism, any information returned to the client-side network infrastructure as a consequence of the `Execute` call shall be discarded. In this case, the Client Cached Block Cookie value shall remain unchanged.

11.4.2.2.2 Behavior specification in the `EM_NO_RETURN_VALUE` state

The client shall not be blocked; that is, the `Execute` call returns as soon as the operation request message is sent by the client-side network infrastructure.

Any information returned to the client-side network infrastructure as a consequence of the `Execute` call shall be discarded.

The Client Cached Block Cookie value shall remain unchanged.

11.4.2.3 Behavior of the `ClientServerReturnCode` for the `Execute` operations

`Execute` returns a `ClientServerReturnCode` to the client. The elements of the `ClientServerReturnCode`, see 7.2.3.1.2, shall be assigned values as follows.

If `execute_mode` is `EM_RETURN_VALUE,` the following values shall be used:

— `portCode`: This field shall be assigned a value from the `MajorReturnCode` enumeration subject to the restrictions of 7.2.3.3.1.

— `performCode`: This field shall be assigned the `performCode` field value of the `ClientServerReturnCode` returned by Server Object's `Perform`, or a value from the `MajorReturnCode` enumeration subject to the restrictions of 7.2.3.3.2.

— `operationMinorCode`: This field shall be assigned the `operationMinorCode` field value of the `ClientServerReturnCode` returned by `Perform.`

— `operationMajorCode`: This field shall be assigned the `operationMajorCode` field value of the `ClientServerReturnCode` returned by `Perform.`

If no return is received by the Client Port from `Perform` before the expiration of the Client Port timeout

— `portCode`: This field shall be assigned the `MajorReturnCode` enumeration value `MJ_OPERATION_TIMEOUT.`

Copyright © 2000 IEEE. All rights reserved.

— performCode and operationMajorCode: These fields shall be assigned the MajorReturnCode enumeration value MJ_NO_RETURN_CODE.

— operationMinorCode: This field shall be assigned the MinorReturnCode enumeration value MI_NO_ADDITIONAL_INFORMATION.

If execute_mode is EM_NO_RETURN_VALUE, the portCode, performCode, and operationMajorCode fields shall be assigned the value MJ_COMPLETE, and the operationMinorCode field shall be assigned the value MI_NO_ADDITIONAL_INFORMATION unless an error is detected in which case the appropriate error code shall be assigned.

11.4.2.4 Operation Execute behavior specification

The Execute operation is the client-side construct for producing network independence in a client-server communication model as illustrated in Figure 8. For details of server-side behavior, Steps 5, 6, 7, 8, 9, and 10 of the figure; see 8.2.3.2.

11.4.2.5 Client-server model: client-side Execute behavior specification

In the following the step numbers refer to the steps in Figure 8.

The behavioral steps of this clause shall be implemented consistent with the memory management specifications of Clause 15. Encoding and decoding shall follow the rules of Clause 14.

Step 1:

The client shall encode the local variables representing the input arguments of the target server operation into an Argument Array. These encoded arguments are the server_input_arguments argument of the Client Port's Execute operation.

The client shall be responsible for providing in the Execute call

— The execute_mode argument.

— The server_operation_id argument specifying the Operation ID of the target operation on the server.

— The server_input_arguments argument.

— A reference to a local Argument Array, server_output_arguments. This Array will contain the encoded output arguments from the server operation on the return from the Execute call.

Step 2:

The client invokes the Execute operation on a local Client Port that has been configured to communicate with the desired target Server Object.

Step 3:

The Client Port's Execute operation shall provide to the client-side network infrastructure

— The argument server_operation_id.

— The server_input_arguments Argument Array.

 Copyright © 2000 IEEE. All rights reserved.

— The target Server Object's Object Dispatch Address. This address was provided to the Client Port during system configuration and is the value of the Port's Client Port Properties `serverDispatchAddress` member.

— Depending on the IEEE 1451.1 implementation, the Client Cached Block Cookie; see Step 4 below.

If `execute_mode` is `EM_RETURN_VALUE`, the `Execute` operation shall

— Start the call timeout mechanism.

— Block awaiting the return from the client-side network infrastructure. If the timeout expires before Step 12 is reached, the `Execute` operation shall return to the calling client with return code as specified in 11.4.2.3. In this case it is recommended that the output Argument Array returned be bound to a zero length Array.

If `execute_mode` is `EM_NO_RETURN_VALUE`, the `Execute` operation shall return immediately to the calling client with return code as specified in 11.4.2.3. In this case, it is recommended that the output Argument Array returned be bound to a zero length Array.

Step 4:

The client-side network infrastructure shall

— Marshal the arguments `server_operation_id`, `server_input_arguments,` and the Client Cached Block Cookie into their network-specific, on-the-wire formats. Depending on the IEEE 1451.1 implementation, the Client Cached Block Cookie may be provided either by the client-side network infrastructure or by the Client Port.

— Cause the delivery of a network message containing the marshaled data to the server-side network infrastructure.

Step 11:

If `execute_mode` was `EM_RETURN_VALUE` and the `Execute` operation has not timed out`,` the network infrastructure on the client-side shall demarshal the data in the return network message sent by the server-side, network infrastructure. This data includes

— The server operation's output arguments encoded into an Argument Array

— The server-side `ClientServerReturnCode`

— The current value of the target Server Object's Block Cookie; see 8.2.3.2

Depending on the IEEE 1451.1 implementation, the client-side network infrastructure shall either cache the returned Block Cookie or it shall provide the returned Block Cookie to the Client Port for caching; see Step 4 above.

The client-side network infrastructure shall provide the demarshaled, but encoded, server operation's output arguments and the server-side `ClientServerReturnCode` to the Client Port and unblock the blocked `Execute` operation.

If `execute_mode` was `EM_NO_RETURN_VALUE`, or the `Execute` operation has timed out, the clientside network infrastructure shall discard the returned message.

Copyright © 2000 IEEE. All rights reserved.

Step 12:

In the case that the `execute_mode` was `EM_RETURN_VALUE` and the `Execute` operation has not timed out, on resumption of execution the Client Port `Execute` operation shall

— Depending on the IEEE 1451.1 implementation, cache the returned Block Cookie; see Step 11 above

— Bind the server operation's encoded output arguments to the client-supplied Argument Array referenced by `server_output_arguments`

— Augment the server-side `ClientServerReturnCode` with the appropriate `portCode` field

— Return to the client

Step 13:

The client decodes `server_output_arguments` into local variables corresponding to the output arguments of the target server operation.

NOTE—See Annex B for a detailed example of a complete client-server interaction.

11.5 Asynchronous Client Port class

Class: `IEEE1451_AsynchronousClientPort`

Parent Class: `IEEE1451_BaseClientPort`

Class ID: 1.1.3.1.1.2

Description: The Asynchronous Client Port class provides client-side functionality for an asynchronous, nonblocking, client-server communication model.

Class summary:

Local operations (see Table 104).

Table 104—`IEEE1451_AsynchronousClientPort` Local operations

Operation Name	Requirement
`ExecuteAsynchronous`	(m)
`GetAsynchronousClientResultStatus`	(m)
`GetResult`	(m)
`AbortTransaction`	(m)
`ForceAcquireTransaction`	(m)

Publications: There are no publications defined for this class.

Subscriptions: There are no subscriptions defined for this class.

 Copyright © 2000 IEEE. All rights reserved.

11.5.1 Operation specifications: Local

11.5.1.1 Local operation `ExecuteAsynchronous` specification

```
IDL: OpReturnCode ExecuteAsynchronous(
           in UInteger16 server_operation_id,
           in ArgumentArray server_input_arguments,
           out UInteger16 transaction_id);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.5.1.1.1 argument `server_operation_id` specification

The `server_operation_id` argument value shall be the Operation ID of the operation to be invoked on the Server Object.

11.5.1.2 argument `server_input_arguments` specification

The `server_input_arguments` argument value shall be the input arguments specified by the server operation's signature, corresponding to the argument `server_operation_id` encoded into an Argument Array.

11.5.1.3 argument `transaction_id` specification

The `transaction_id` argument value shall identify this instance of the invocation of `ExecuteAsynchronous` on the Port. The scope of this identification argument is this Port.

11.5.1.4 Local operation `GetAsynchronousClientResultStatus` specification

```
IDL: OpReturnCode GetAsynchronousClientResultStatus(
           in UInteger16 transaction_id,
           out UInteger8 result_status);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.5.1.4.1 argument `transaction_id` specification

The `transaction_id` argument value shall identify the transaction for which status is requested.

11.5.1.4.2 argument `result_status` specification

The `result_status` argument value shall reflect status corresponding to the `transaction_id` argument. This status indicates the availability of the results expected from the server. The allowed values of `result_status` shall be selected from the enumeration shown in Table 105.

IDL: enumeration `AsynchronousClientResultStatus;`

11.5.1.5 Local operation `GetResult` specification

```
IDL: ClientServerReturnCode GetResult(
           in UInteger16 transaction_id,
           out ArgumentArray server_output_arguments);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

Copyright © 2000 IEEE. All rights reserved.

Table 105—`AsynchronousClientResultStatus` enumeration

Enumeration	Value	Meaning
`RS_TRANSACTION_COMPLETE`	0	The invocation of the `GetResult` operation will return without blocking (may occur when the Port is in the PORT ERROR or RESULTS AVAILABLE states).
`RS_TRANSACTION_PENDING`	1	The invocation of the `GetResult` will cause `GetResult` to block (may occur when the Port is in the PENDING state).
Reserved	2–255	

11.5.1.5.1 argument `server_output_arguments` specification

This `server_output_arguments` argument value shall be the output arguments specified by the signature of the server operation (corresponding to the `server_operation_id`), encoded into an Argument Array.

11.5.1.6 Local operation `AbortTransaction` specification

```
IDL: OpReturnCode AbortTransaction(
          in UInteger16 transaction_id);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.5.1.7 Local operation `ForceAcquireTransaction` specification

```
IDL: OpReturnCode ForceAcquireTransaction(
          out UInteger16 new_transaction_id,
          out UInteger16 current_server_operation_id);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.5.1.7.1 argument `new_transaction_id` specification

The `new_transaction_id` argument value shall identify a new value for the `transaction_id` of the Port.

11.5.1.7.2 argument `current_server_operation_id` specification

The `current_server_operation_id` argument value shall be the Operation ID of the operation invoked on the Server Object during the transaction pre-empted by the invocation of this operation.

11.5.2 Asynchronous Client Port class behavior

11.5.2.1 Specific Asynchronous Client Port operation restrictions based on state

In addition to and conformant with the restrictions of 11.3.2.1, the allowed operations defined on an Asynchronous Client Port shall be restricted as defined in Table 106. Where an operation is indicated as operational, it still shall be restricted to specific uses defined by the restrictions of 9.1.3.2.1, 9.1.3.2.2, and 9.1.3.2.3.

 Copyright © 2000 IEEE. All rights reserved.

Table 106—`AsynchronousClientPort` class specific restrictions

Operation	Specified value of `rule_basis`		
	BL_UNINITIALIZED	**BL_INACTIVE**	**BL_ACTIVE**
All operations defined by this class except those in the following row	Local invocation only	Operational	Operational
`ExecuteAsynchronous`	Not operational	Operational	Operational

11.5.2.2 Concurrency behavior specification

The `AbortTransaction` and `ForceAcquireTransaction` operations shall support Precedence Concurrency.

The `GetResult` and `GetAsynchronousClientResultStatus` operations shall support Active Concurrency.

11.5.2.3 State behavior specification

The state behavior of the Asynchronous Client Port shall be as shown in Figure 26.

11.5.2.4 Transaction behavior specification

A valid `transaction_id` shall be marked as NO_TRANSACTION when any of the following occur (see also 11.5.2.7 and 11.5.2.9):

— `GetResult` returns to the client

— Successful invocation of `AbortTransaction`

The NO_TRANSACTION value shall be 0.

Two transaction IDs match if, and only if

— They have the same value **AND**

— Neither is the NO_TRANSACTION value

11.5.2.5 Operations `ExecuteAsynchronous` behavior specification

The `ExecuteAsynchronous` operation is the client-side construct for producing network independence in a client-server, asynchronous communication model as illustrated in Figure 27. The details of server-side behavior, Steps 5, 6, 7, 8, 9, and 10 on the right side of the figure are identical to those specified for the corresponding Steps in 8.2.3.2.

11.5.2.6 Client-server model: client-side `ExecuteAsynchronous` behavior specification

In the following the step numbers refer to the steps in Figure 27.

The behavioral steps of this clause shall be implemented consistently with the memory management specifications of Clause 15. Encoding and decoding shall follow the rules of Clause 14.

Copyright © 2000 IEEE. All rights reserved.

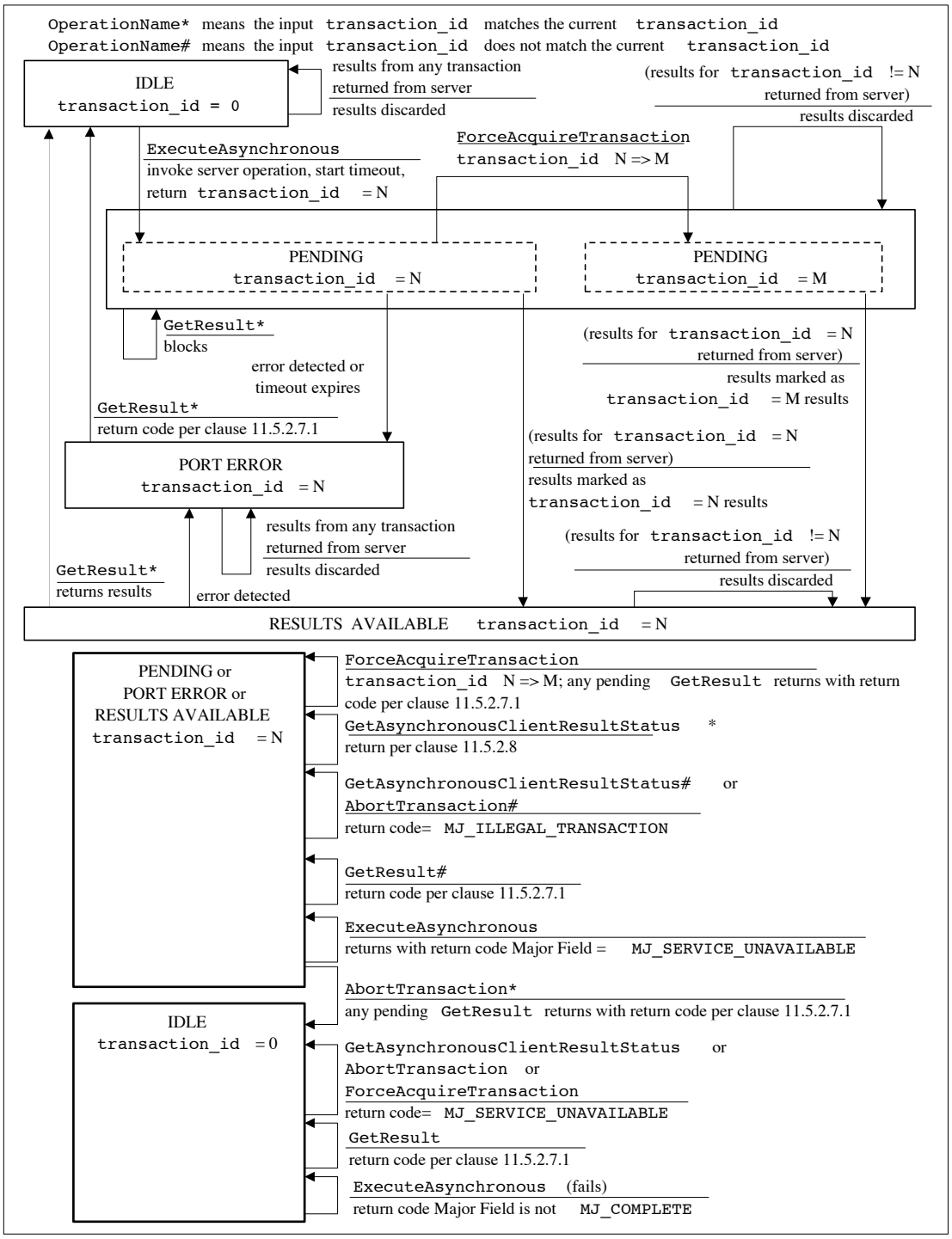

Figure 26—State machine for an Asynchronous Client Port

 Copyright © 2000 IEEE. All rights reserved.

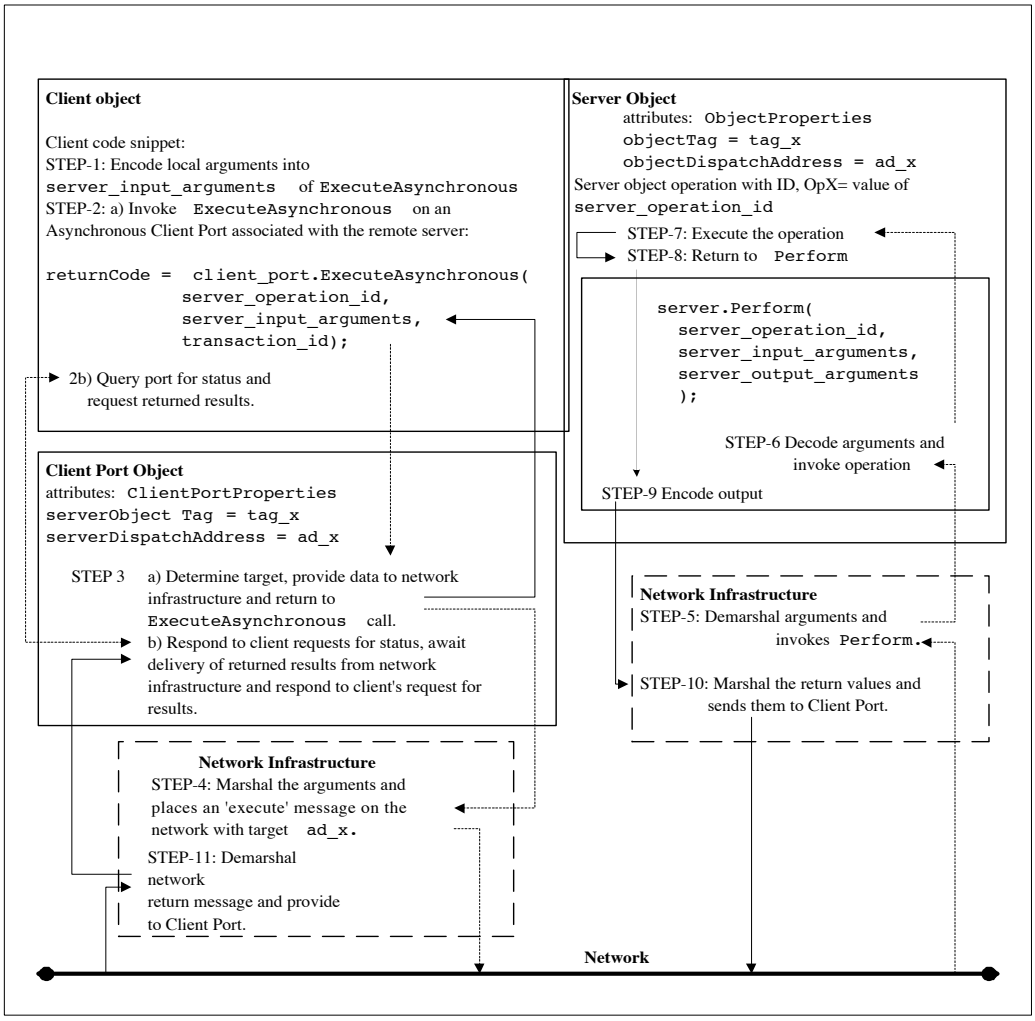

Figure 27—Asynchronous client-server communication model

Step 1:

The client shall encode the local variables representing the input arguments of the target remote server operation into an Argument Array. These encoded arguments are the `server_input_arguments` argument of the Asynchronous Client Port's `ExecuteAsynchronous` operation.

The client shall be responsible for providing in the `ExecuteAsynchronous` call

— The `server_operation_id` argument specifying the Operation ID of the target operation on the server

— The `server_input_arguments` argument

— A reference to a local variable representing the `transaction_id` argument

Copyright © 2000 IEEE. All rights reserved.

Step 2:

— The client invokes the ExecuteAsynchronous operation on a local Asynchronous Client Port that has been configured to communicate with the desired target Server Object. This call shall return as soon as the Port initiates an asynchronous client-server transaction.

— The client then proceeds with other activities. These activities may include invoking the `GetResult` or `AbortTransaction` operations on the Asynchronous Client Port.

Step 3:

a) The Asynchronous Client Port's `ExecuteAsynchronous` operation shall

— Provide to the client-side network infrastructure

— The argument `server_operation_id.`

— The `server_input_arguments` Argument Array.

— The target Server Object's Object Dispatch Address. This address was provided to the Asynchronous Client Port during system configuration and is the target Server Object's Object Dispatch Address.

— Depending on the IEEE 1451.1 implementation, the Client Cached Block Cookie; see Step 4 below.

— Request that the client-side network infrastructure deliver a network message requesting the remote service to the server-side network infrastructure.

— Start the transaction timeout mechanism.

— Return to the client with a valid `transaction_id` and an appropriate `OpReturnCode.`

b) The Asynchronous Client Port shall then go into a mode where it can

— Respond to client requests for the completion status of the server operation

— Respond to a client request for the returned results of the remote operation

— Respond to a client request to abort the transaction

— Respond to a local, third-party request to force acquire the transaction

— Monitor the transaction timeout mechanism

— Accept the returned remote operation's results from the client-side network infrastructure.

If a request that the transaction be aborted comes from a client that holds a valid `transaction_id`, then the Asynchronous Client Port shall invalidate the current `transaction_id` and return the appropriate `OpReturnCode` to the caller. In this case, the asynchronous client-server transaction is completed.

If a local third-party object invokes a `ForceAcquireTransaction` on the Port, the Port shall invalidate its current `transaction_id` and provide the caller with a new, valid `transaction_id`. In this case, the caller becomes the new client for the ongoing transaction.

If the client-side network infrastructure provides the Port with the remote operation's results and the transaction has timed out or the transaction has been aborted, then the Port shall discard the returned results. Otherwise, the Asynchronous Client Port shall augment the server-side `ClientServerReturnCode` with the appropriate `portCode` field.

If the client holds a valid `transaction_id` and requests the operation's results

 Copyright © 2000 IEEE. All rights reserved.

— If the remote operation's results have been returned to the Asynchronous Client Port, then the Port shall supply the encoded results to the calling client and the asynchronous client-server transaction is completed.

— If the server operation's results have not been returned to the Port and the transaction has not timed out or been aborted, then the request results call shall block, awaiting either the return of the results, a transaction timeout, the aborting of the transaction, or the force acquire of the transaction.

Step 4:

The client-side network infrastructure shall

— Marshal the arguments `server_operation_id` and `server_input_arguments` and the Client Cached Block Cookie into their network-specific, on-the-wire formats. Depending on the IEEE 1451.1 implementation, the Client Cached Block Cookie may be provided either by the clientside network infrastructure or by the Asynchronous Client Port.

— Cause the delivery of a network message containing the marshaled data to the server-side network infrastructure.

Step 11:

The client-side network infrastructure shall demarshal the data in the return network message sent by the server-side network infrastructure. This data includes

— The server operation's output arguments encoded into an Argument Array

— The server-side `ClientServerReturnCode`

— The current value of the target Server Object's Block Cookie

Depending on the IEEE 1451.1 implementation, the client-side network infrastructure shall either cache the returned Block Cookie or it shall provide the returned Block Cookie to the Asynchronous Client Port for caching; see Step 4 above.

The network infrastructure shall provide the demarshaled, but encoded, server operation's output arguments and the server-side `ClientServerReturnCode` to the Asynchronous Client Port.

It is possible that a result from a prior transaction will be received during the execution of a new transaction. Implementers shall ensure that the results of the prior transaction are disregarded. The mechanism for enforcing this behavior is outside the scope of this standard.

NOTE—This circumstance might appear as the result of the following sequence of events: invocation of `ExecuteAsynchronous` (with transaction_id 2); `timeout, AbortTransaction`; invocation of `ExecuteAsynchronous` (with transaction_id 3), return of results from transaction_id 2.

11.5.2.7 Operation `GetResult` behavior specification

The behavior of this operation shall depend on the Port status.

When `GetResult` returns, the Port `transaction_id` shall be marked as NO_TRANSACTION and the internal Asynchronous Client Port state corresponding to the transaction shall be released.

Copyright © 2000 IEEE. All rights reserved.

If subsequent to an invocation of `ExecuteAsynchronous` that returns `transaction_id` = N, a `ForceAcquireTransaction` successfully completes and returns a `new_transaction_id` = M, an invocation of `GetResult` with a `transaction_id` of M shall

— Follow the rules of the state machine of Figure 26.

— Return the server results corresponding to `transaction_id` = N if they become available

If `GetResult` returns as a consequence of a successful completion of the `AbortTransaction` operation, the `server_output_arguments` shall be bound to a zero length Argument Array.

11.5.2.7.1 Behavior specification of the `ClientServerReturnCode` for the operation `GetResult`

`GetResult` returns a `ClientServerReturnCode` to the client. The elements of the `ClientServerReturnCode`, see 7.2.3.1.2, shall be assigned the following values, except when the return is a result of an `AbortTransaction` or `ForceAcquireTransaction` operation

— `portCode`: This field shall be assigned a value from the `MajorReturnCode` enumeration subject to the restrictions of 7.2.3.3.1 and Table 107.

— `performCode`: This field shall be assigned the `performCode` field value of the `ClientServerReturnCode` returned to the Port by `Perform` or a value from the `MajorReturnCode` enumeration subject to the restrictions of 7.2.3.3.2 and Table 107.

— `operationMajorCode` and `operationMinorCode` fields shall be assigned values as indicated in Table 107.

Table 107—`GetResult` return codes

Port state details	portCode (performCode)	operationMajorCode (operationMinorCode)
RESULTS AVAILABLE server results available	`MJ_COMPLETE`	As returned from server
	(As returned from server)	(As returned from server)
PORT ERROR Asynchronous Client Port timeout	`MJ_OPERATION_TIMEOUT`	`MJ_NO_RETURN_CODE`
	`(MJ_NO_RETURN_CODE)`	`(MI_NO_ADDITIONAL_INFORMATION)`
PENDING	N/A	N/A
	(N/A)	(N/A)
PORT ERROR not due to timeout	`MJ_FAILED_NON_SPECIFIC` Unless more appropriate information is available	As returned from server or if no return `MJ_NO_RETURN_CODE`
	(As returned from server or if no return; `MJ_NO_RETURN_CODE`)	`(MI_NO_ADDITIONAL_INFORMATION)`
IDLE	`MJ_SERVICE_UNAVAILABLE`	`MJ_NO_RETURN_CODE`
	`(MJ_NO_RETURN_CODE)`	`(MI_NO_ADDITIONAL_INFORMATION)`
Call in any non-IDLE state with illegal transaction	`MJ_ILLEGAL_TRANSACTION`	`MJ_NO_RETURN_CODE`
	`(MJ_NO_RETURN_CODE)`	`(MI_NO_ADDITIONAL_INFORMATION)`

 Copyright © 2000 IEEE. All rights reserved.

When the return of `GetResult` is the result of an `AbortTransaction` or a `ForceAcquireTransaction` operation, the `ClientServerReturnCode` fields shall be assigned as follows:

- `portCode: MJ_OPERATION_INTERRUPTED`
- `performCode: MJ_NO_RETURN_CODE`
- `operationMajorCode: MJ_NO_RETURN_CODE`
- `operationMinorCode: MI_NO_ADDITIONAL_INFORMATION`

11.5.2.8 Operation `GetAsynchronousClientResultStatus` behavior specification

The allowed values for the `result_status` argument returned by `GetAsynchronousClientResultStatus` behavior shall be restricted based on the state of the Asynchronous Client Port, see 11.5.2.3, as indicated in Table 108.

Table 108—result_status restrictions

Enumeration	States	Meaning
`RS_TRANSACTION_COMPLETE`	RESULTS AVAILABLE	The Asynchronous Client Port has received the output arguments of the remote operation. These arguments may be retrieved by using the `GetResult` operation.
`RS_TRANSACTION_COMPLETE`	PORT ERROR	The Port is in an error state. The returned results are meaningless.
`RS_TRANSACTION_PENDING`	PENDING	The Asynchronous Client Port has not received the output arguments of the remote operation. Invocation of `GetResult` will cause `GetResult` to block.
`RS_TRANSACTION_PENDING`	Any non-IDLE state when the call has an invalid `transaction_id`	The Major Field of the `OpReturnCode` shall be `MJ_ILLEGAL_TRANSACTION.`
`RS_TRANSACTION_PENDING`	IDLE	The Major Field of the `OpReturnCode` shall be `MJ_SERVICE_UNAVAILABLE.`

11.5.2.9 Operation `AbortTransaction` behavior specification

The successful `AbortTransaction` operation shall cause any returns from the remote server operation to be discarded. The Port `transaction_id` shall be marked as NO_TRANSACTION and the corresponding transaction-state of the Asynchronous Client Port shall be released.

The `AbortTransaction` operation acts only on the Port. Any effect on the server shall be subclass implementation–dependent and outside the scope of this standard.

The return code Major Field values for an invocation `AbortTransaction` operation shall be as defined in Table 109.

Table 109—`AbortTransaction` Major Field restrictions

States	Major Field
RESULTS AVAILABLE	`MJ_COMPLETE`
PENDING PORT ERROR	`MJ_COMPLETE`
Any non-IDLE state when the call has an invalid `transaction_id`	`MJ_ILLEGAL_TRANSACTION`
IDLE	`MJ_SERVICE_UNAVAILABLE`

Copyright © 2000 IEEE. All rights reserved.

11.5.2.10 Operation `ForceAcquireTransaction` behavior specification

The `ForceAcquireTransaction` operation shall cause the current `transaction_id` to be replaced by a new value that is returned to the caller. Any server results returned as the result of the immediately preceding invocation of `ExecuteAsynchronous` shall be treated as if invoked under the new `transaction_id`. Any pending `GetResult` operations shall return as specified in 11.5.2.7.1.

This operation shall have no result if invoked in the IDLE state.

The return code Major Field values for an invocation `ForceAcquireTransaction` operation shall be as defined in Table 110.

Table 110—`ForceAcquireTransaction` Major Field restrictions

States	Major Field
RESULTS AVAILABLE PENDING PORT ERROR	`MJ_COMPLETE`
IDLE	`MJ_SERVICE_UNAVAILABLE`

NOTE—See Annex B for a detailed example of a complete client-server interaction.

11.6 Base Publisher Port abstract class

Abstract Class

Class: `IEEE1451_BasePublisherPort`

Parent Class: `IEEE1451_BasePort`

Class ID: 1.1.3.1.2

Description: The Base Publisher Port class provides basic publisher-side functionality for its subclasses.

Class summary:

Network Visible operations (see Table 111).

Table 111—`IEEE1451_BasePublisherPort` Network Visible operations

Operation Name	Requirement	Operation ID
`SetPublicationTopic`	(o)	8218
`GetPublicationTopic`	(o)	8219
`GetPublicationKey`	(m)	8220
`SetPublicationDomain`	(m)	8221
`GetPublicationDomain`	(m)	8222

Publications: There are no publications defined for this class.

Subscriptions: There are no subscriptions defined for this class.

 Copyright © 2000 IEEE. All rights reserved.

11.6.1 Operation specifications: Network Visible

11.6.1.1 Operation `SetPublicationTopic` specification

```
IDL: OpReturnCode SetPublicationTopic(
           in PublicationTopic publication_topic);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.6.1.1.1 argument `publication_topic` specification

The `publication_topic` argument value shall be the current value of the Publication Topic for the Port. Publication Topic is a semantic identifier associated with the publication. The properties and allowed values for Publication Topic shall be as specified in 12.3.2 and 12.3.4.

11.6.1.2 Operation `GetPublicationTopic` specification

```
IDL: OpReturnCode GetPublicationTopic(
           out PublicationTopic publication_topic);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.6.1.3 Operation `GetPublicationKey` specification

```
IDL: OpReturnCode GetPublicationKey(
           out UInteger8 publication_key);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.6.1.3.1 argument `publication_key` specification

The `publication_key` argument value shall be the current value of the Publication Key. The properties and allowed values for Publication Keys shall be as specified in 12.3.2 and 12.3.3.

11.6.1.4 Operation `SetPublicationDomain` specification

```
IDL: OpReturnCode SetPublicationDomain(
           in PubSubDomain publication_domain);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.6.1.4.1 argument `publication_domain` specification

The `publication_domain` argument value shall be the current Publication Domain of the publication. Publication Domains define the scope of the publication's distribution. The properties and allowed values for Publication Domains shall be as specified in 12.3.5.

11.6.1.5 Operation `GetPublicationDomain` specification

```
IDL: OpReturnCode GetPublicationDomain(
           out PubSubDomain publication_domain);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

Copyright © 2000 IEEE. All rights reserved.

11.6.2 Operation specifications: Local

11.6.2.1 Local publishing operations

Any local publishing operations are subclass specific. Any such publishing operations shall return an `OpReturnCode` Minor Field value taken from the enumeration shown in Table 112 and the restriction following:

IDL: enumeration `PublicationContentsReturn;`

Table 112—`PublicationContentsReturn` enumeration

Enumeration	Value	Meaning
`PC_NO_ADDITIONAL_INFORMATION`	0	See restriction
`PC_MEMORY_ALLOCATION_ERROR`	1	See restriction
`PC_MISSING_INPUT`	2	See restriction
`PC_INVALID_TYPE`	3	See restriction
`PC_INVALID_VALUE`	4	See restriction
`PC_COMPUTATION_ERROR`	5	See restriction
`PC_EMPTY_DOMAIN`	6	The Domain is empty.
`PC_KEY_NOT_SET`	7	The Publication Key has not been set.
`PC_TOPIC_NOT_SET`	8	The Publication Topic has not been set and the Publication Key is not terminal.
`PC_TOPIC_KEY_CONFLICT`	9	The Publication Topic has been set and the Publication Key is terminal.
`PC_ILLEGAL_KEY`	10	The Publication Key is illegal for the type of Port or for the specific publication operation invoked.
`PC_ILLEGAL_STATE`	11	The Port has detected an illegal internal state.
Reserved	12–255	

Restriction—The meaning of the first six members of the enumerations shall be identical to the meaning of the first six members of the `MinorReturnCode` enumeration; see 7.2.3.2.1.

11.6.3 Publisher Port attributes

Publication Key: Except where otherwise noted, the management of Publication Key values is outside the scope of this standard. The mechanism of setting the Publication Key is outside of this standard.

Publication Topic: Defined by the system configuration.

Publication Domain: Defined by the system configuration.

Publication Contents: The syntax of the Publication Contents is Publication Key specific. The semantics of Publication Contents is Publisher specific.

Publisher Port: Object Name; see 8.2.3.6.

 Copyright © 2000 IEEE. All rights reserved.

11.6.4 Base Publisher Port class behavior

11.6.4.1 Specific Base Publisher Port operation restrictions based on state

In addition to and conformant with the restrictions of 11.2.2.1, the allowed operations defined on a Base Publisher Port shall be restricted as defined in Table 113. Where an operation is indicated as operational, it still shall be restricted to specific uses defined by the restrictions of 9.1.3.2.1, 9.1.3.2.2, and 9.1.3.2.3.

Table 113—`IEEE1451_BasePublisherPort` class specific restrictions

Operation	Specified value of `rule_basis`		
	BL_UNINITIALIZED	**BL_INACTIVE**	**BL_ACTIVE**
`SetPublicationTopic`, `SetPublicationDomain`	Local invocation only	Operational	Not operational
`GetPublicationTopic`, `GetPublicationKey`, `GetPublicationDomain`	Local invocation only	Operational	Operational

11.6.4.2 Operations `SetPublicationTopic` and `GetPublicationTopic` behavior specification

`SetPublicationTopic` and `GetPublicationTopic` shall both have a Full Implementation for all Port Objects whose Publication Key value indicates that the Publication Key is not terminal; see 12.3.2.

`SetPublicationTopic` and `GetPublicationTopic` may both have an Interface Only Implementation if a Port Object's Publication Key value indicates that the Publication Key is terminal; see 12.3.2.

Restrictions on Publication Topics based on whether the Publication Key is or is not terminal are found in 12.3.4.

11.7 Publisher Port class

Class: `IEEE1451_PublisherPort`

Parent Class: `IEEE1451_BasePublisherPort`

Class ID: 1.1.3.1.2.1

Description: The Publisher Port class provides publisher-side functionality for a publish-subscribe communication model.

Class summary:

Local operations (see Table 114).

Table 114—`IEEE1451_PublisherPort` Local operations

Operation Name	Requirement
`Publish`	(m)

Copyright © 2000 IEEE. All rights reserved.

Publications: Publications may be defined for this class.

Subscriptions: There are no subscriptions defined for this class.

11.7.1 Operation specifications: Local

11.7.1.1 Local operation `Publish` specification

```
IDL: OpReturnCode Publish(
           in ArgumentArray publication_contents);
```

The Minor Field values of `OpReturnCode` shall be taken from the `PublicationContentsReturn` enumeration.

11.7.1.1.1 argument `publication_contents` specification

The `publication_contents` argument value shall be the Publisher-provided information to be included in the Publication Contents of the publication.

11.7.2 Publisher Port class behavior

11.7.2.1 Specific Publisher Port operation restrictions based on state

In addition to and conformant with the restrictions of 11.6.4.1, the allowed operations defined on a Publisher Port instance shall be restricted as defined in Table 115. Where an operation is indicated as operational, it still shall be restricted to specific uses defined by the restrictions of 9.1.3.2.1, 9.1.3.2.2, and 9.1.3.2.3.

Table 115—`IEEE1451_PublisherPort` class specific restrictions

Operation	Specified value of `rule_basis`		
	BL_UNINITIALIZED	**BL_INACTIVE**	**BL_ACTIVE**
`Publish`	Not operational	Operational	Operational

11.7.2.2 Publication restrictions

Publishers may use a Port of this class for Publication Key values that define publications for which information about the Publisher may not be requested by Subscribers. The allowed values of Publication Key for which this Port shall be used are specified in 12.3.3 as Class A key values. For these Publication Key values

— The publication shall be published under the terms of 12.2, if and only if the Publication Key is a member of Class A.

— If the Publication Key is not a member of Class A, the `Publish` operation shall return with a `MajorReturnCode` value of `MJ_SERVICE_UNAVAILABLE` and a `MinorReturnCode` value of `PC_ILLEGAL_KEY`.

11.7.2.3 Publication behavior specification

The Publisher shall be responsible for encoding local Publisher variables into the `publication_contents` Argument Array of the `Publish` operation.

 Copyright © 2000 IEEE. All rights reserved.

11.8 Self Identifying Publisher Port class

Class: `IEEE1451_SelfIdentifyingPublisherPort`

Parent Class: `IEEE1451_BasePublisherPort`

Class ID: 1.1.3.1.2.2

Description: The Self Identifying Publisher Port class provides publisher-side functionality for a publish-subscribe communication model with operations to allow the subscriber to establish communication with the publisher, and the publisher to notify subscribers of changes in the publication policy.

Class summary:

Network Visible operations (see Table 116).

Table 116—IEEE1451_SelfIdentifyingPublisherPort Network Visible operations

Operation Name	Requirement	Operation ID
`GetPublisherMetadata`	(m)	10253
`PublishPublisherMetadata`	(m)	10254

Local operations (see Table 117).

Table 117—IEEE1451_SelfIdentifyingPublisherPort Local operations

Operation Name	Requirement
`PublishWithIdentification`	(m)
`RegisterPublisher`	(m)
`DeregisterPublisher`	(m)

Publications: A publication may be defined for this class.

Subscriptions: There are no subscriptions defined for this class.

11.8.1 Operation specifications: Network Visible

11.8.1.1 Operation `GetPublisherMetadata` specification

```
IDL: OpReturnCode GetPublisherMetadata(
          in UInteger16 publication_id,
          in UInteger16 cached_publication_change_id,
          out ArgumentArray publisher_metadata);
```

The Minor Field enumeration for this operation shall be defined in Table 118.

Copyright © 2000 IEEE. All rights reserved.

IDL: enumeration `PublisherMetadataReturn;`

Table 118—PublisherMetadataReturn enumeration

Enumeration	Value
CS_NO_ADDITIONAL_INFORMATION	0
CS_MEMORY_ALLOCATION_ERROR	1
CS_MISSING_INPUT	2
CS_INVALID_TYPE	3
CS_INVALID_VALUE	4
CS_COMPUTATION_ERROR	5
CS_INFORMATION_NOT_AVAILABLE	6
Reserved	7–255

The meaning of the first six members of this enumeration shall be identical to the meaning of the first six members of the `MinorReturnCode` enumeration; see 7.2.3.2.1.

11.8.1.1.1 argument `publication_id` specification

The `publication_id` argument value shall be the value of the `publicationID` field of the `publisher_identification_information` Argument of any publications issued from this Self Identifying Publisher Port.

11.8.1.1.2 argument `cached_publication_change_id` specification

The `cached_publication_change_id` argument value shall be the current Subscriber cached value of the Publication Change ID received in the publication for which the `publisher_metadata` is being requested. The Publication Change ID shall be the value of the `publicationChangeID` field of the `publisher_identification_information` Argument of any publications issued from this Self Identifying Publisher Port.

11.8.1.1.3 argument `publisher_metadata` specification

The `publisher_metadata` argument value shall contain information describing the publication and the Publisher. The syntax and semantics of this information is both Publisher and publication specific.

11.8.1.2 Operation `PublishPublisherMetadata` specification

```
IDL: OpReturnCode PublishPublisherMetadata(
           in UInteger16 publication_id,
           in UInteger16 cached_publication_change_id);
```

The Minor Field enumeration for this operation shall be defined by the `PublisherMetadataReturn` enumeration.

 Copyright © 2000 IEEE. All rights reserved.

11.8.2 Operation specifications: Local

11.8.2.1 Local operation `PublishWithIdentification` specification

```
IDL: OpReturnCode PublishWithIdentification(
           in ArgumentArray publication_contents);
```

The Minor Field values of `OpReturnCode` shall be taken from the `PublicationContentsReturn` enumeration.

11.8.2.1.1 argument `publication_contents` specification

The `publication_contents` argument value shall be the Publisher provided information to be included in the Publication Contents of the publication.

11.8.2.2 Local operation `RegisterPublisher` specification

```
IDL: OpReturnCode RegisterPublisher(
           in <local operation reference>
              callback_operation_reference,
           out UInteger16 publication_id);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.8.2.2.1 argument `callback_operation_reference` specification

The `callback_operation_reference` argument value shall be a local reference to a local operation of the registering Publisher object that shall be invoked to signal a request for `publisher_metadata`. The signature of the referenced callback operation shall be

```
OpReturnCode <local operation name>(
           in UInteger16 publication_id,
           in Boolean request_client_server_flag,
           in UInteger16 cached_publication_change_id,
           out ArgumentArray publisher_metadata);
```

11.8.2.2.1.1 argument `publication_id` specification

The `publication_id` argument of the callback operation shall be the value provided as an input argument in either the `PublishPublisherMetadata` or `GetPublisherMetadata` operations invoked on this Self Identifying Publisher Port; see 11.8.3.3.

11.8.2.2.1.2 argument `request_client_server_flag` specification

The `request_client_server_flag` argument value shall define the method of response to the request for `publisher_metadata`.

— `TRUE` shall indicate that the Subscriber call was for the `GetPublisherMetadata` operation, that the metadata is expected via the client-server return, and that no publication of metadata is requested.

— `FALSE` shall indicate that the Subscriber call was for the `PublishPublisherMetadata` operation and that no data is expected via the client-server return.

Copyright © 2000 IEEE. All rights reserved.

11.8.2.2.1.3 argument `cached_publication_change_id` specification

The `cached_publication_change_id` argument value shall be the value provided by the Subscriber invoking either the `GetPublisherMetadata` or the `PublishPublisherMetadata` operations. This value may be used by the Publisher to determine the specific metadata of interest to the Subscriber. The specifics of this determination are outside the scope of this standard.

11.8.2.2.1.4 argument `publisher_metadata` specification

The `publisher_metadata` argument value shall be the metadata associated with the publication defined by the arguments `publication_id` and `cached_publication_change_id`. This metadata is the metadata that would be published by using the `PublishWithIdentification` operation with one of the following values for `content_code`:

— `PCC_METADATA`
— `PCC_BOTH_META_AND_NORMAL_DATA`
— `PCC_GROUPED_METADATA`
— `PCC_GROUPED_BOTH_META_AND_NORMAL_DATA`

11.8.2.2.2 argument `publication_id` specification

The `publication_id` output argument value of the `RegisterPublisher` operation shall be a distinct value assigned to the registering Publisher for each instance of `<local operation reference>` it registers with the Self Identifying Publisher Port.

11.8.2.3 Local operation `DeregisterPublisher` specification

```
IDL: OpReturnCode DeregisterPublisher(
           in <local operation reference>
              callback_operation_reference,
           in UInteger16 publication_id);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.8.3 Self Identifying Publisher Port class behavior

11.8.3.1 Specific Self Identifying Publisher Port operation restrictions based on state

In addition to and conformant with the restrictions of 11.6.4.1, the allowed operations defined on a Self Identifying Publisher Port instance shall be restricted as defined in Table 119. Where an operation is indicated as operational, it still shall be restricted to specific uses defined by the restrictions of 9.1.3.2.1, 9.1.3.2.2, and 9.1.3.2.3.

11.8.3.2 Publication restrictions

Publishers may use a Port of this class for Publication Key values that define publications for which additional information may be requested of the Publisher. The allowed values of Publication Key for which this Port shall be used with the `PublishWithIdentification` operation are specified in 12.3.3 as Class B key values.

 Copyright © 2000 IEEE. All rights reserved.

Table 119—`IEEE1451_SelfIdentifyingPublisherPort` class specific restrictions

Operation	Specified value of `rule_basis`		
	BL_UNINITIALIZED	**BL_INACTIVE**	**BL_ACTIVE**
`GetPublisherMetadata`, `RegisterPublisher`, `DeregisterPublisher`	Local invocation only	Operational	Operational
`PublishPublisherMetadata`, `PublishWithIdentification`	Not operational	Operational	Operational

For these Publication Key values

- The local `PublishWithIdentification` operation shall be used.
- Publishers shall be registered with the Port using the operation `RegisterPublisher.`
- The publication shall be published under the terms of 12.2, if and only if the Publisher is properly registered and the Publication Key is a member of Class B.
- Error condition 1: If the Publisher is not properly registered, the `PublishWithIdentification` operation shall return with a `MajorReturnCode` value of `MJ_SERVICE_UNAVAILABLE` and a `MinorReturnCode` value of `PC_ILLEGAL_STATE.`
- Error condition 2: If the Publication Key is not a member of Class B, the `PublishWithIdentification` operation shall return with a `MajorReturnCode` value of `MJ_SERVICE_UNAVAILABLE` and a `MinorReturnCode` value of `PC_ILLEGAL_KEY.`

If both of the previous two error conditions obtain, the `MinorReturnCode` shall be `PC_ILLEGAL_STATE.`

The Publication Key value `PSK_EVENT_NOTIFICATION` shall not be used with a Port of class `IEEE1451_SelfIdentifyingPublisherPort`. If specified in the subclass definition, the Publication Key value `PSK_EVENT_NOTIFICATION` may be used with a Port that is a subclass of `IEEE1451_SelfIdentifyingPublisherPort`.

11.8.3.3 Operations `GetPublisherMetadata` and `PublishPublisherMetadata` behavior specification

The invocation of `GetPublisherMetadata` or the `PublishPublisherMetadata` shall follow all of the specifications for client-server communication.

The invocation of the `GetPublisherMetadata` or the `PublishPublisherMetadata` operations shall result in the Self Identifying Publisher Port invoking the callback operation on the Publisher identified by the `publication_id` argument.

An invocation on the Publisher object of the callback operation shall cause the Publisher to

- Return the requested Publisher information via the out argument `publisher_metadata` of the callback operation if the `request_client_server_flag` argument is `TRUE`. The Self Identifying Publisher Port shall then return this information to the requesting client via the normal client-server mechanisms.
- Publish the Publisher information via the normal invocation of `PublishWithIdentification` if the `request_client_server_flag` argument is `FALSE`. In this case, the Argument Array

Copyright © 2000 IEEE. All rights reserved.

output argument `publisher_metadata` of the callback operation shall have as value the default value for an Argument Array.

The `MinorReturnCode` shall be `CS_INFORMATION_NOT_AVAILABLE` and the `MajorReturnCode` shall be `MJ_SERVICE_UNAVAILABLE` if the Publisher can not provide the requested information.

11.8.3.4 Operation `PublishWithIdentification` behavior specification

The Argument Array `publication_contents` shall be passed to the Self Identifying Publisher Port by the Publisher and shall be as defined in Table 120.

The Publication Contents of the publication as delivered to the network infrastructure shall be defined in Table 120.

Table 120—`publication_contents` definition

Argument Array Member	Member Argument `TypeCode` corresponding to type	Argument Name
0	`PublisherInformation`	`publisher_identification_information`
1	`UInteger8`	`content_code`
Remaining arguments	`Argument`	`application_specific_contents`

11.8.3.4.1 argument `publisher_identification_information` behavior specification

The 0th member of the Publication Contents Argument Array shall be the `PublisherInformation` structure, `publisher_identification_information`. This member shall be encoded by the Publisher into an Argument. The values of the fields of this structure shall be set as follows:

— `portObjectTag`: The Self Identifying Publisher Port shall set the encoded value of this field to the value of its Object Tag.

— `publicationID`: The Publisher shall set this field to the value of its `publication_id` established by the `RegisterPublisher` operation.

— `publicationChangeID`: The Publisher shall set this field to the value of its Publication Change ID.

11.8.3.4.2 Publication Change ID behavior specification

The Publication Change ID shall be a Publisher-defined and -maintained identifier associated with Publisher-specific metadata concerning the semantics of the publication. A change in the value of the Publication Change ID may be used to indicate to Subscribers that the metadata associated with the publication has changed. The circumstances indicating the need for such a change are outside the scope of this standard.

 Copyright © 2000 IEEE. All rights reserved.

11.8.3.4.3 argument `content_code` behavior specification

The Publication Contents semantics and format shall be defined by the `content_code`. These `content_code` dependent formats shall be specific to each publication. The value of `content_code` shall be set and encoded by the Publisher

The allowed value of the `content_code` shall be taken from the `PublicationContentCode` enumeration as shown in Table 121.

IDL: enumeration `PublicationContentCode;`

Table 121—`PublicationContentCode` enumeration

Enumeration	Value	Meaning
`PCC_NORMAL_DATA`	0	The usual contents of the publication for the defined topic.
`PCC_METADATA`	1	The information requested as a result of the invocation of the `PublishPublisherMetadata` operations.
`PCC_BOTH_META_AND_NORMAL_DATA`	2	The publication contains both the meta and normal data, defined however, for the application or publication.
`PCC_GROUPED_NORMAL_DATA`	3	Same as 0 but for a group of similarly formatted data.
`PCC_GROUPED_METADATA`	4	Same as 1 but for a group of similarly formatted metadata.
`PCC_GROUPED_BOTH_META_AND_NORMAL_DATA`	5	Same as 2 but for a group of similarly formatted meta and normal data.
Reserved values	6–127	
Open to industry	128–255	

11.8.3.4.4 argument `application_specific_contents` specification

The `application_specific_contents` argument value shall be the application specific Publisher defined information provided by the Publisher. The Publisher shall encode this information into Argument Array member 2 through the final member Argument of the Publication Contents Argument Array.

The Port shall pass the Publication Contents Argument Array to the underlying network infrastructure to be published.

11.9 Event Generator Publisher Port class

Class: `IEEE1451_EventGeneratorPublisherPort`

Parent Class: `IEEE1451_SelfIdentifyingPublisherPort`

Class ID: 1.1.3.1.2.2.1

Copyright © 2000 IEEE. All rights reserved.

Description: The Event Generator Publisher Port class may be used to allow events internal to the operation of a block to result in the publication of an event record.

Class summary:

Network Visible operations (see Table 122).

Table 122—`IEEE1451_EventGeneratorPublisherPort` Network Visible operations

Operation Name	Requirement	Operation ID
`GetEventGeneratorState`	(m)	12298
`SetEventSequenceNumber`	(m)	12299
`GetEventSequenceNumber`	(m)	12300
`SetResponseTag`	(m)	12301
`GetResponseTag`	(m)	12302
`GetLastTimestamp`	(m)	12303
`EnableEventPublication`	(m)	12304
`DisableEventPublication`	(m)	12305

Publications: A publication with Publication Key value of `PSK_EVENT_NOTIFICATION` is defined for this class.

Subscriptions: There are no subscriptions defined for this class.

11.9.1 Operation specifications: Network Visible

11.9.1.1 Operation `GetEventGeneratorState` specification

```
IDL: OpReturnCode GetEventGeneratorState(
          out UInteger8 event_generator_state);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.9.1.1.1 argument `event_generator_state` specification

The `event_generator_state` argument value shall be defined by the following enumeration shown in Table 123.

IDL: enumeration `EventGeneratorState;`

Table 123—`EventGeneratorState` enumeration

Enumeration	Value
`EG_DISABLED`	0
`EG_ENABLED`	1
Reserved	2–255

 Copyright © 2000 IEEE. All rights reserved.

11.9.1.2 Operation `SetEventSequenceNumber` specification

```
IDL: OpReturnCode SetEventSequenceNumber(
           in UInteger32 event_sequence_number);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.9.1.2.1 argument `event_sequence_number` specification

The `event_sequence_number` argument value shall order the publication of event records from this Event Generator Publisher Port.

11.9.1.3 Operation `GetEventSequenceNumber` specification

```
IDL: OpReturnCode GetEventSequenceNumber(
           out UInteger32 event_sequence_number);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.9.1.4 Operation `SetResponseTag` specification

```
IDL: OpReturnCode SetResponseTag(
           in ObjectTag response_object_tag);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.9.1.4.1 argument `response_object_tag` specification

The `response_object_tag` argument value shall be the current value of the Object Tag of the object that shall receive any response to this event by its receiving Subscribers. The management of `response_object_tag` values is outside the scope of this standard.

The purpose of the `response_object_tag` is to allow application system integrators to designate an object to handle selected application functionality required by the posting of a publication by the Event Generator Publisher Port.

11.9.1.5 Operation `GetResponseTag` specification

```
IDL: OpReturnCode GetResponseTag(
           out ObjectTag response_object_tag);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.9.1.6 Operation `GetLastTimestamp` specification

```
IDL: OpReturnCode GetLastTimestamp(
           out TimeRepresentation last_event_timestamp);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

Copyright © 2000 IEEE. All rights reserved.

11.9.1.6.1 argument `last_event_timestamp` specification

The `last_event_timestamp` argument value shall indicate the time at which the last event activating this Event Generator Publisher Port occurred, irrespective of whether the detection of the event resulted in the publication of an event record.

11.9.1.7 Operation `EnableEventPublication` specification

IDL: `OpReturnCode EnableEventPublication( );`

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.9.1.8 Operation `DisableEventPublication` specification

IDL: `OpReturnCode DisableEventPublication( );`

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.9.2 Event Generator Publisher Port class behavior

11.9.2.1 Specific Event Generator Publisher Port operation restrictions based on state

In addition to and conformant with the restrictions of 11.8.3.1, the allowed operations defined on an Event Generator Publisher Port instance shall be restricted as defined in Table 124. Where an operation is indicated as operational, it still shall be restricted to specific uses defined by the restrictions of 9.1.3.2.1, 9.1.3.2.2, and 9.1.3.2.3.

Table 124—`IEEE1451_EventGeneratorPublisherPort` class specific restrictions

Operation	Specified value of `rule_basis`		
	BL_UNINITIALIZED	BL_INACTIVE	BL_ACTIVE
All operations defined by this class except those in the following row	Local invocation only	Operational	Operational
`SetEventSequenceNumber`, `SetResponseTag`	Local invocation only	Operational	Not operational

11.9.2.2 Operational behavior specification

Implementers may cause an event record to be published containing `optional_arguments`. The specifics of what constitutes an event, and how an event is detected, are outside the scope of this standard.

The event detection mechanism, that is the Publisher, shall invoke the inherited `PublishWithIdentification` operation to cause the event record to be published by the Event Generator Publisher Port.

The implementation shall define the semantics of the underlying event and the means for detection.

The definition of `optional_arguments` is application-specific and is outside the scope of this standard.

 Copyright © 2000 IEEE. All rights reserved.

11.9.2.3 argument `event_sequence_number` behavior specification

The `event_sequence_number` shall be maintained by the Port and incremented by one each time an event record is published. The current value may be accessed by the `SetEventSequenceNumber` and `GetEventSequenceNumber` operations.

11.9.2.4 argument `response_object_tag` behavior specification

The `response_object_tag` argument shall be the Object Tag of the object that shall receive any response to the posting of a publication by the Event Generator Publisher Port. This tag value shall be maintained by the Port.

The `response_object_tag` value equal to the default value for an Object Tag shall indicate that no responses shall be accepted.

The specification of the results of a response is outside the scope of this standard.

11.9.2.5 Operations `GetLastTimestamp` behavior specification

`GetLastTimestamp` may be an Interface Only Implementation if the implementation does not provide the ability to timestamp events; otherwise, it shall be a Full Implementation.

The most recent value of the `event_detection_time` shall be available from the Port as `last_event_timestamp`. The `last_event_timestamp` shall be available via the `GetLastTimestamp` operation irrespective of the state of the Event Generator Publisher Port.

The `event_detection_time` shall be the time at which the underlying event was detected by the event detection mechanism.

11.9.2.6 argument `event_name` behavior specification

The `event_name` argument of the Publication Contents shall represent the name of the event. The management of this argument is outside the scope of this standard.

11.9.2.7 Publication semantics

The Publication Key for the publication issued by this Port shall be `PSK_EVENT_NOTIFICATION.`

The `content_code` argument of `PublishWithIdentification` shall be either `PCC_NORMAL_DATA` or shall be user-defined.

The Publication Contents for `content_code` values other than `PCC_NORMAL_DATA` is application dependent and is outside the scope of this standard.

11.9.2.7.1 argument `publication_contents` behavior specification

The Argument Array `publication_contents` for `content_code` `PCC_NORMAL_DATA` passed to the Port by the Publisher defines the Publication Contents of the publication. The form of the Publication Contents Argument Array shall be as defined in Table 125.

The `publisher_identification_information` and `content_code` members shall be provided as defined in the parent class; see 11.8.3.4.

The Publisher shall provide the values for the `event_detection_time` and `event_name`.

Copyright © 2000 IEEE. All rights reserved.

Table 125—Event Generator `publication_contents` definition

Argument Member	Member Argument `TypeCode` corresponding to type	Value
0	`PublisherInformation`	`publisher_identification_information`
1	`UInteger8`	`content_code`
2	`UInteger32`	`event_sequence_number`
3	`ObjectTag`	`response_object_tag`
4	`TimeRepresentation`	`event_detection_time`
5	`String`	`event_name`
Remaining	`Argument`	`optional_arguments`

The Port shall provide the values for the `event_sequence_number` and `response_object_tag`.

The Publisher shall provide any `optional_arguments` variables.

Encoding of the Publication Contents Argument Array shall follow the rules of Clause 14.

11.9.2.8 State behavior specification

The state behavior of an Event Generator Publisher Port shall be as defined in Figure 28.

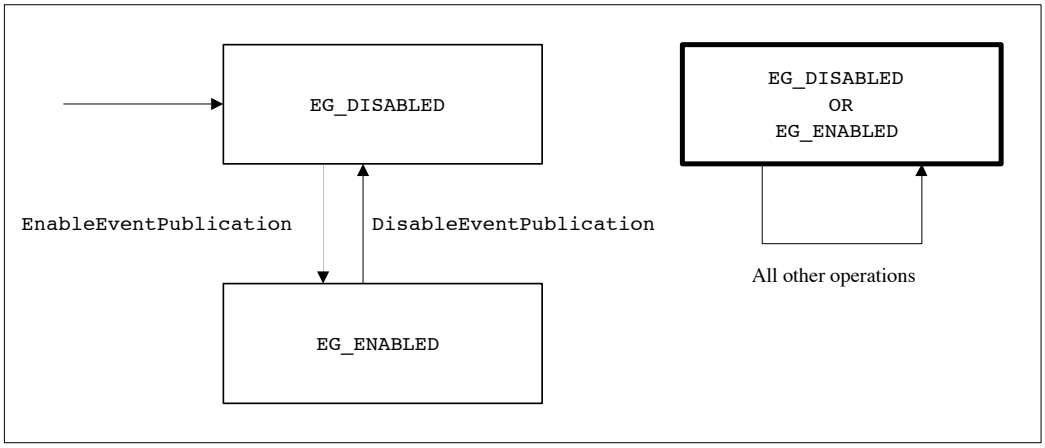

Figure 28—State machine for an Event Generator Publisher Port

The initial state of the Event Generator Publisher Port shall be the `EG_DISABLED` state as indicated by the entry transition in the state machine.

The publication of event records shall be as defined in Table 126.

Table 126—Event Generator Publisher Port state restrictions

State	Meaning
`EG_DISABLED`	No event records shall be published.
`EG_ENABLED`	Event records shall be published.

Copyright © 2000 IEEE. All rights reserved.

11.10 Subscriber Port class

Class: `IEEE1451_SubscriberPort`

Parent Class: `IEEE1451_Service`

Class ID: 1.1.3.2

Description: The Subscriber Port class provides objects with a mechanism for subscribing to publications.

Class summary:

Network Visible operations (see Table 127).

Table 127—`IEEE1451_SubscriberPort` Network Visible operations

Operation Name	Requirement	Operation ID
`SetSubscriptionQualifier`	(m)	6195
`GetSubscriptionQualifier`	(m)	6196
`GetSubscriptionKey`	(m)	6197
`SetSubscriptionDomain`	(m)	6198
`GetSubscriptionDomain`	(m)	6199

Local Operations (see Table 128).

Table 128—`IEEE1451_SubscriberPort` Local operations

Operation Name	Requirement
`AddSubscriber`	(m)
`DeleteSubscriber`	(m)

Publications: There are no publications defined for this class.

Subscriptions: Subscription may be defined for this class.

11.10.1 Operation specifications: Network Visible

11.10.1.1 Operation `SetSubscriptionQualifier` specification

```
IDL: OpReturnCode SetSubscriptionQualifier(
          in SubscriptionQualifier subscription_qualifier);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.10.1.1.1 argument `subscription_qualifier` specification

The `subscription_qualifier` argument value shall be the current value of the Port's Subscription Qualifier. The Subscription Qualifier is used in the determination of which publications are to be accepted by

Copyright © 2000 IEEE. All rights reserved.

the Port. The properties and allowed values for Subscription Qualifiers shall be as specified in 12.3.2 and 12.3.4.

11.10.1.2 Operation `GetSubscriptionQualifier` specification

```
IDL: OpReturnCode GetSubscriptionQualifier(
          out SubscriptionQualifier subscription_qualifier);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.10.1.3 Operation `GetSubscriptionKey` specification

```
IDL: OpReturnCode GetSubscriptionKey(
          out UInteger8 subscription_key);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.10.1.3.1 argument `subscription_key` specification

The `subscription_key` argument value shall be the current value of the Port's Subscription Key. The properties and allowed values for Subscription Keys shall be as specified in 12.3.2 and 12.3.3.

11.10.1.4 Operation `SetSubscriptionDomain` specification

```
IDL: OpReturnCode SetSubscriptionDomain(
          in PubSubDomain subscription_domain);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.10.1.4.1 argument `subscription_domain` specification

The `subscription_domain` argument shall be the current value of the Subscription Domain defining a set of candidate publications to be accepted by the Port; see Clause 12. The properties and allowed values for Subscription Domain shall be as specified in 12.3.5.

11.10.1.5 Operation `GetSubscriptionDomain` specification

```
IDL: OpReturnCode GetSubscriptionDomain(
          out PubSubDomain subscription_domain);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.10.2 Operation specifications: Local

11.10.2.1 Local operation `AddSubscriber` specification

```
IDL: OpReturnCode AddSubscriber(
          in <local operation reference>
             notification_operation_reference,
          in UInteger16 subscription_id);
```

The Minor Field enumeration for this operation shall be defined by the `PublicationContentsReturn` enumeration; see 11.6.2.1.

 Copyright © 2000 IEEE. All rights reserved.

11.10.2.1.1 argument `notification_operation_reference` specification

The `notification_operation_reference` argument value shall be a `<local operation reference>` to the local operation of the registering Subscriber object that shall be invoked to signal that a requested publication has been received. The signature of the operation shall be

```
void <local operation name>(
    in UInteger16 subscription_id,
    in UInteger8 publishing_port_publication_key,
    in PublicationTopic
       publishing_port_publication_topic,
    in ArgumentArray publication_contents);
```

11.10.2.1.1.1 argument `subscription_id` specification

The `subscription_id` argument value of the callback `<local operation name>` shall be the Subscriber-defined alias for the Publication Topic defined by `AddSubscriber`.

11.10.2.1.1.2 argument `publishing_port_publication_key` specification

The `publishing_port_publication_key` argument value shall be the actual Publication Key that the Publisher used when issuing the publication.

11.10.2.1.1.3 argument `publishing_port_publication_topic` specification

The `publishing_port_publication_topic` argument value shall be the actual Publication Topic that the Publisher used when issuing the publication.

11.10.2.1.1.4 argument `publication_contents` specification

The `publication_contents` argument value shall be the Publication Contents of the received publication.

11.10.2.1.2 argument `subscription_id` specification

The `subscription_id` argument value of the `AddSubscriber` operation shall be a Subscriber-defined alias for the Publication Topic.

11.10.2.2 Local operation `DeleteSubscriber` specification

```
IDL: OpReturnCode DeleteSubscriber(
           in <local operation reference>
              notification_operation_reference,
           in UInteger16 subscription_id);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.10.3 Subscriber Port class behavior

11.10.3.1 Specific Subscriber Port operation restrictions based on state

In addition to and conformant with the restrictions of 11.1.2.2, the allowed operations defined on a Subscriber Port instance shall be restricted as defined in Table 129. Where an operation is indicated as operational, it still shall be restricted to specific uses defined by the restrictions of 9.1.3.2.1, 9.1.3.2.2, and 9.1.3.2.3.

Copyright © 2000 IEEE. All rights reserved.

Table 129—`IEEE1451_SubscriberPort` class specific state

Operation	Specified value of `rule_basis`		
	BL_UNINITIALIZED	**BL_INACTIVE**	**BL_ACTIVE**
All operations defined by this class except those in the following row	Local invocation only	Operational	Not operational
`GetSubscriptionQualifier`, `GetSubscriptionKey`, `GetSubscriptionDomain`, `AddSubscriber`, `DeleteSubscriber`	Local invocation only	Operational	Operational

11.10.3.2 Subscription receipt, demarshaling, decoding and notification behavior specification

The selection of publications shall be as defined in 12.3.4.

The network infrastructure supporting the Subscriber Port shall be responsible for demarshaling the received

— Publication Contents

— Publication Topic

— Publication Key

— Publication Domain

from the network-specific, on-the-wire format, into the appropriate datatypes and delivering this information to the appropriate Subscriber Ports.

If a qualified publication is received by a Subscriber Port, all registered Subscribers shall be notified via the callback operations established by the `AddSubscriber` operation.

The Subscriber shall be responsible for decoding the Publication Contents received as an Argument Array via the callback operation into appropriate local variables.

The decoding of the Publication Contents shall be based on the format of the contents of the publication as defined by the `publishing_port_publication_topic` and `publishing_port_publication_key`.

11.10.3.3 Role of identifiers in subscribing with this Port

`portObjectTag`, `publicationID`, and `publicationChangeID` information may be present as part of the Publication Contents if the subscription is to a publication issued by a Self Identifying Publisher Port; see 11.8.

For publications from an ordinary Publisher Port, this information is not available.

An example of a publish-subscribe communication is found in Annex C.

 Copyright © 2000 IEEE. All rights reserved.

11.11 Mutex Service class

Class: `IEEE1451_MutexService`

Parent Class: `IEEE1451_Service`

Class ID: 1.1.3.3

Description: The Mutex Service class provides mutual exclusion capability.

Class summary:

Network Visible operations (see Table 130).

Table 130—`IEEE1451_MutexService` Network Visible operations

Operation Name	Requirement	Operation ID
`Lock`	(m)	6200
`Unlock`	(m)	6201
`TryLock`	(m)	6202
`IsLocked`	(m)	6203
`ForceAcquireLock`	(m)	6204

Publications: There are no publications defined for this class.

Subscriptions: There are no subscriptions defined for this class.

11.11.1 Operation specifications: Network Visible

11.11.1.1 Operation `Lock` specification

```
IDL: OpReturnCode Lock(
           in TimeRepresentation timeout,
           out UInteger16 transaction_id);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.11.1.1.1 argument `timeout` specification

The `timeout` argument value shall be the maximum time that the client shall be blocked before the operation returns.

11.11.1.1.2 argument `transaction_id` specification

The `transaction_id` argument value shall be a value generated by the Mutex Service.

11.11.1.2 Operation `Unlock` specification

```
IDL: OpReturnCode Unlock(in UInteger16 transaction_id);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

Copyright © 2000 IEEE. All rights reserved.

11.11.1.3 Operation `TryLock`

```
IDL: OpReturnCode TryLock(out UInteger16 transaction_id);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.11.1.4 Operation `IsLocked` specification

```
IDL:: OpReturnCode IsLocked(out Boolean mutex_status);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.11.1.4.1 argument `mutex_status` specification

The `mutex_status` argument value shall be `TRUE` if the Mutex Service is LOCKED, otherwise the value shall be `FALSE`.

11.11.1.5 Operation `ForceAcquireLock` specification

```
IDL: OpReturnCode ForceAcquireLock(
          out UInteger16 transaction_id);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.11.2 Mutex Service class behavior

11.11.2.1 Specific Mutex Service operation restrictions based on state

In addition to and conformant with the restrictions of 11.1.2.2, the allowed operations defined on a Mutex Service instance shall be restricted as defined in Table 131. Where an operation is indicated as operational, it still shall be restricted to specific uses defined by the restrictions of 9.1.3.2.1, 9.1.3.2.2, and 9.1.3.2.3.

Table 131—`IEEE1451_MutexService` class specific restrictions

Operation	Specified value of `rule_basis`		
	BL_UNINITIALIZED	BL_INACTIVE	BL_ACTIVE
All operations defined by this class	Local invocation only	Operational	Operational

11.11.2.2 Overview of Mutex behavior

The Mutex Service is always in one of the following two states:

— UNLOCKED

— LOCKED

11.11.2.2.1 Operation `Unlock` behavior specification

The `Unlock` operation on a Mutex Service in the LOCKED state shall unlock the Mutex Service, provided the `transaction_id` values match. The call shall return immediately. If any `Lock` operations are blocked on the formerly LOCKED Mutex Service, one and only one of them shall be unblocked and shall

 Copyright © 2000 IEEE. All rights reserved.

return with a new `transaction_id`. An attempt to unlock an already UNLOCKED Mutex Service shall return immediately.

11.11.2.2.2 Operation `Lock` behavior specification

The `Lock` operation on an UNLOCKED Mutex Service shall lock the Mutex Service and return a `transaction_id` value that shall be different from prior `transaction_id` values. If an attempt to lock a Mutex Service that is already in the LOCKED-state is made, the call shall block until the Mutex Service is UNLOCKED and the specific `Lock` operation call is the one selected to be unblocked and receive the new `transaction_id`, or the call timeout expires.

11.11.2.2.3 Behavior of the operation `TryLock`

The `TryLock` operation on an UNLOCKED Mutex Service shall lock the Mutex Service and return a `transaction_id` value that shall be different from prior `transaction_id` values. If the `TryLock` operation is invoked on a LOCKED Mutex Service, the call shall return with the return code `MJ_SERVICE_UNAVAILABLE`.

11.11.2.2.4 Operation `ForceAcquireLock` behavior specification

A `ForceAcquireLock` operation shall leave the Mutex Service LOCKED, invalidates the current `transaction_id`, and returns a `transaction_id` that shall be different from prior `transaction_id` values. An attempt to `ForceAcquireLock` the lock of an UNLOCKED Mutex Service shall return immediately.

11.11.2.3 State behavior specification

The state behavior of the Mutex Service shall be as defined Figure 29.

The initial state for a Mutex Service shall be the UNLOCKED state as indicated by the entry transition of the state machine.

All state transitions for the Mutex Service shall be atomic. Where there are multiple actions associated with the transition, the result of the transition shall be exactly as though these actions executed without interference from Concurrent Execution of any operation on any object in the process space.

Unless otherwise indicated in the state diagram, when calls return the `MajorReturnCode` field shall be `MJ_COMPLETE` and the `MinorReturnCode` field shall be `MI_NO_ADDITIONAL_INFORMATION.`

A `timeout` value of 0 shall indicate that the Mutex Service timeout never expires.

If more than a single client is blocked on a LOCKED Mutex Service when a successful `Unlock` operation is invoked, there shall be no requirement designating which client to unblock unless otherwise stated in a subclass specification.

It is recommended that, within the numerical range constraints of the datatype, `transaction_id` values not be reused.

Copyright © 2000 IEEE. All rights reserved.

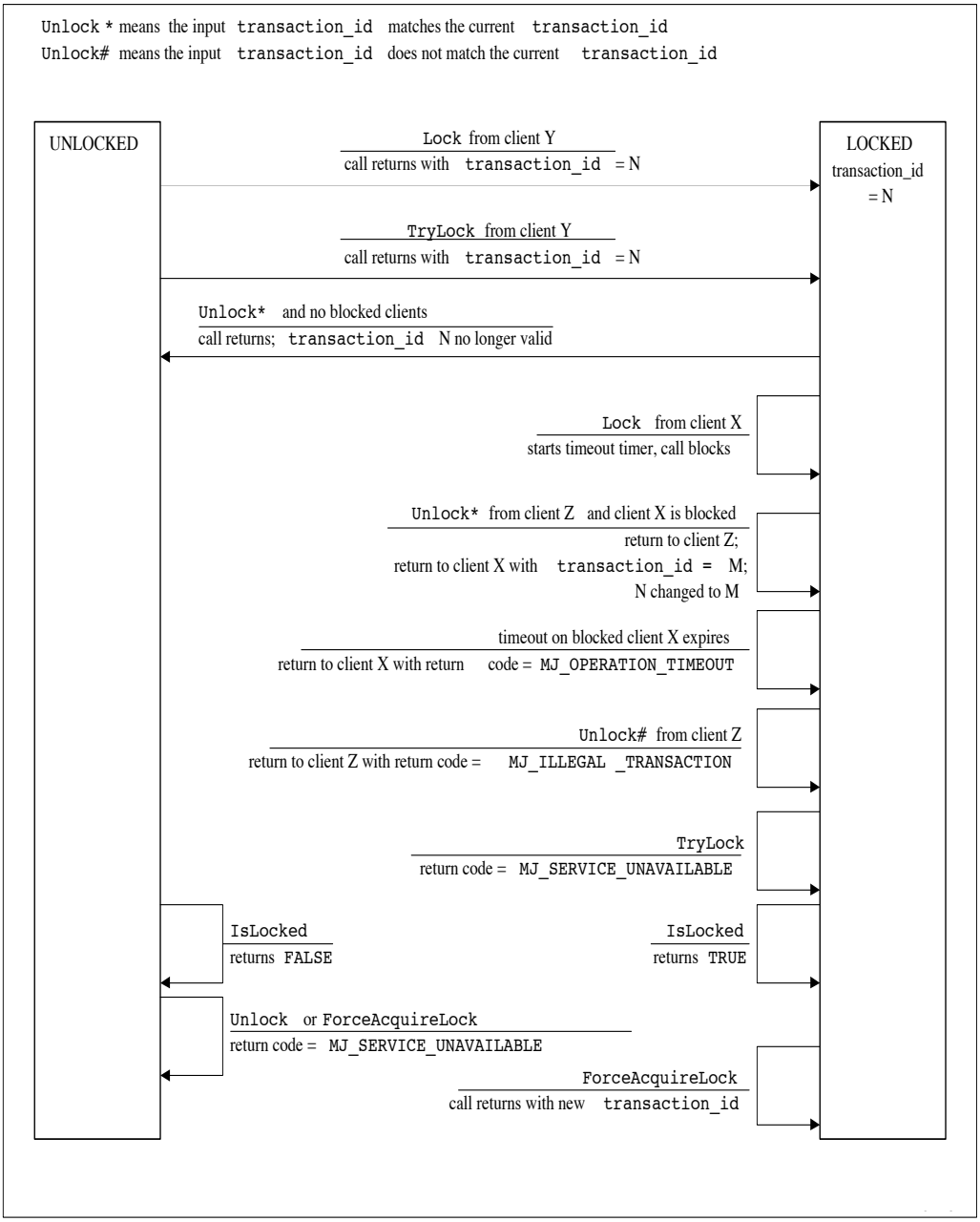

Figure 29—State machine for a Mutex Service

Copyright © 2000 IEEE. All rights reserved.

11.12 Condition Variable Service class

Class: `IEEE1451_ConditionVariableService`

Parent Class: `IEEE1451_Service`

Class ID: 1.1.3.4

Description: The Condition Variable Service class provides the capability for ordering concurrent activities.

Class summary:

Network Visible operations (see Table 132).

Publications: There are no publications defined for this class.

Subscriptions: There are no subscriptions defined for this class.

Table 132—`IEEE1451_ConditionVariableService` Network Visible operations

Operation Name	Requirement	Operation ID
`Lock`	(m)	6205
`Unlock`	(m)	6206
`TryLock`	(m)	6207
`SetPredicateState`	(m)	6208
`Wait`	(m)	6209
`SignalOne`	(m)	6210
`SignalAll`	(m)	6211
`IsLocked`	(m)	6212
`GetPredicateState`	(m)	6213
`ForceAcquireLock`	(m)	6214
`ClearAllPendingWaits`	(m)	6215

11.12.1 Operation specifications: Network Visible

11.12.1.1 Operation `Lock` specification

```
IDL: OpReturnCode Lock(
          in TimeRepresentation lock_timeout,
          out UInteger16 transaction_id);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.12.1.1.1 argument `lock_timeout` specification

The `lock_timeout` argument value shall be the maximum time that the client shall be blocked before a return.

11.12.1.1.2 argument `transaction_id` specification

The `transaction_id` argument value shall be a value generated by the Condition Variable Service.

Copyright © 2000 IEEE. All rights reserved.

11.12.1.2 Operation `Unlock` specification

```
IDL: OpReturnCode Unlock(in UInteger16 transaction_id);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.12.1.3 Operation `TryLock` specification

```
IDL: OpReturnCode TryLock(out UInteger16 transaction_id);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.12.1.4 Operation `SetPredicateState` specification

```
IDL: OpReturnCode SetPredicateState(
           in Boolean predicate_state,
           in UInteger16 transaction_id);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.12.1.4.1 argument `predicate_state` specification

The `predicate_state` argument value shall be the desired state of the Condition Variable Service's predicate.

11.12.1.5 Operation `Wait` specification

```
IDL: OpReturnCode Wait(
           in TimeRepresentation wait_timeout,
           out UInteger16 transaction_id);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.12.1.5.1 argument `wait_timeout` specification

The `wait_timeout` argument value shall be the maximum time that the client shall be blocked before a return.

11.12.1.6 Operation `SignalOne` specification

```
IDL: OpReturnCode SignalOne(in UInteger16 transaction_id);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.12.1.7 Operation `SignalAll` specification

```
IDL: OpReturnCode SignalAll(in UInteger16 transaction_id);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.12.1.8 Operation `IsLocked` specification

```
IDL: OpReturnCode IsLocked(out Boolean mutex_status);
```

 Copyright © 2000 IEEE. All rights reserved.

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.12.1.8.1 argument `mutex_status` specification

The `mutex_status` argument value shall indicate the status of the mutex protecting the Condition Variable Service.

11.12.1.9 Operation `GetPredicateState` specification

```
IDL: OpReturnCode GetPredicateState(
          out Boolean predicate_state);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.12.1.9.1 argument `predicate_state` specification

The `predicate_state` argument value shall be the value of the current state of the Condition Variable Service's predicate.

11.12.1.10 Operation `ForceAcquireLock` specification

```
IDL: OpReturnCode ForceAcquireLock(
          out UInteger16 transaction_id);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.12.1.11 Operation `ClearAllPendingWaits` specification

```
IDL: OpReturnCode ClearAllPendingWaits(
          in UInteger16 transaction_id);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

11.12.2 Condition Variable Service class behavior

11.12.2.1 Specific Condition Variable Service operation restrictions based on state

In addition to and conformant with the restrictions of 11.1.2.2, the allowed operations defined on a Condition Variable Service instance shall be restricted as defined in Table 133. Where an operation is indicated as operational, it still shall be restricted to specific uses defined by the restrictions of 9.1.3.2.1, 9.1.3.2.2, and 9.1.3.2.3.

Table 133—`IEEE1451_ConditionVariableService` class specific restrictions

Operation	Specified value of `rule_basis`		
	BL_UNINITIALIZED	BL_INACTIVE	BL_ACTIVE
All operations defined by this class	Local invocation only	Operational	Operational

Copyright © 2000 IEEE. All rights reserved.

11.12.2.2 Overview of Condition Variable Service behavior

This Service is the application interface to a mechanism for controlling the ordering of activities between code segments executing in distinct NCAP processes. This mechanism consists of a Boolean state variable, termed the predicate, a mutex, and an internal mechanism for supporting a collection of blocked `Wait` calls. This internal mechanism is termed the "collection" in this clause.

The Condition Variable Service's predicate is a Boolean with states `TRUE` and `FALSE`. The state of the predicate shall be returned by the `GetPredicateState` operation.

The behavior of the mutex portion is exactly as for the Mutex Service with respect to `Lock`, `Unlock`, `TryLock`, and `ForceAcquireLock`. A client in possession of the lock `transaction_id` may change the predicate's state by using the `SetPredicateState` operation.

Clients can invoke the `Wait` operation on the Service. Conceptually, when a `Wait` is invoked, a state machine shall be created in the collection to manage the execution of the `Wait` operation. There shall be one such conceptual state machine for every invocation of `Wait`.

Each of these state machines, when invoked, shall attempt to secure the mutex lock and if successful shall check the state of the predicate. If the predicate is `TRUE` then the `Wait` call shall return with a new `transaction_id` and its state machine shall be removed from the collection. If the predicate is `FALSE` then the `Wait` call shall block.

A `Wait` state machine that is in this blocked state shall remain so until it receives a signal or a `ClearAllPendingWaits` operation is invoked.

Signals may be issued in two ways

— By a client in possession of the `transaction_id` via the `SignalOne` call. This operation shall signal one of the state machines in the collection. There shall be no requirement for ordering the selection of which state machine to apply the `SignalOne` call.

— By a client in possession of the `transaction_id` via the `SignalAll` call. This operation shall signal all of the state machines in the collection. There shall be no requirement for ordering the signaling of the members of the collection.

When a blocked state machine in the collection is signaled it shall attempt to lock the mutex. If it is successful, it shall check the state of the predicate and act as follows:

— If the predicate is `TRUE` then the `Wait` call shall return with a new `transaction_id` and its state machine shall be removed from the collection.

— If the predicate is `FALSE` then the `Wait` call shall reblock.

If the `Lock` call unblocks with an abnormal return code on the mutex, the `Wait` call shall return with the appropriate return code. Whenever `Wait` returns to the client, its state machine shall be removed from the collection.

A client may force the acquisition of the mutex lock by the invocation of `ForceAcquireLock`. This operation shall return a new `transaction_id` to the client and shall cause any outstanding `transaction_id` to be invalid.

A client holding a valid `transaction_id` may invoke the `ClearAllPendingWaits` operation. This operation shall cause all pending `Wait` calls to unblock and return to their respective clients.

 Copyright © 2000 IEEE. All rights reserved.

The collection manages timeouts such that the `wait_timeout` value supplied by each `Wait` call shall be the maximum timeout before `Wait` returns with the appropriate return code and its state machine is removed from the collection. The state machine of each `Wait` coordinates the `lock_timeout` values provided in requests by the state machine to secure the lock on the mutex in such a way as to produce the following overall timeout behavior. `lock_timeout` values shall be selected such that the value is less than any unexpired time remaining on the `wait_timeout`.

11.12.2.3 State behavior specification

The state behavior of the mutex portion of the Condition Variable Service mechanism shall be as defined in Figure 30, for those mutex operations not dependent on the state of the predicate.

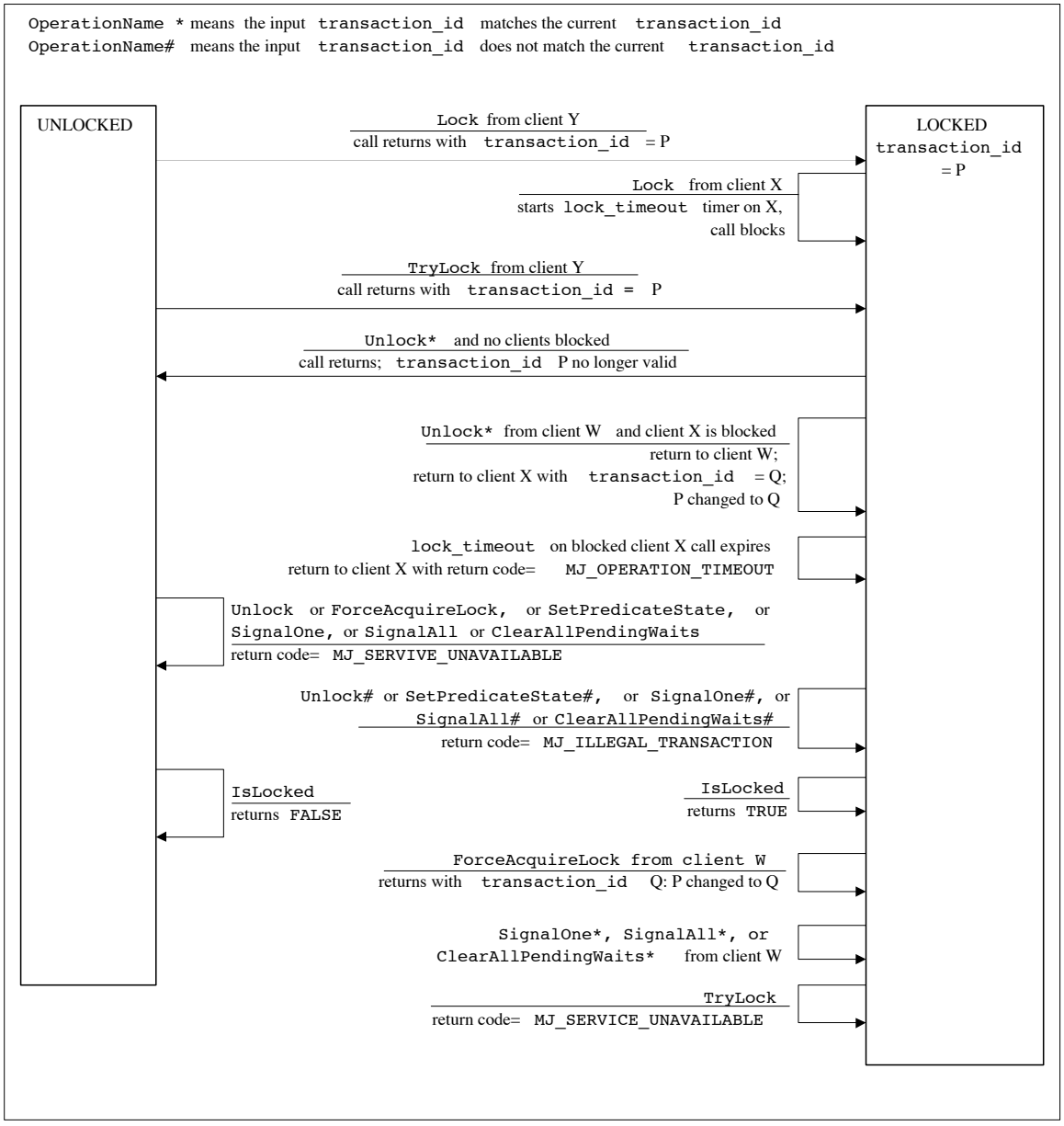

Figure 30—Predicate independent mutex state machine for a Condition Variable Service

Copyright © 2000 IEEE. All rights reserved.

The state behavior of the mutex portion of the Condition Variable Service mechanism shall be as defined in Figure 31, for those mutex operations that are dependent on the state of the predicate.

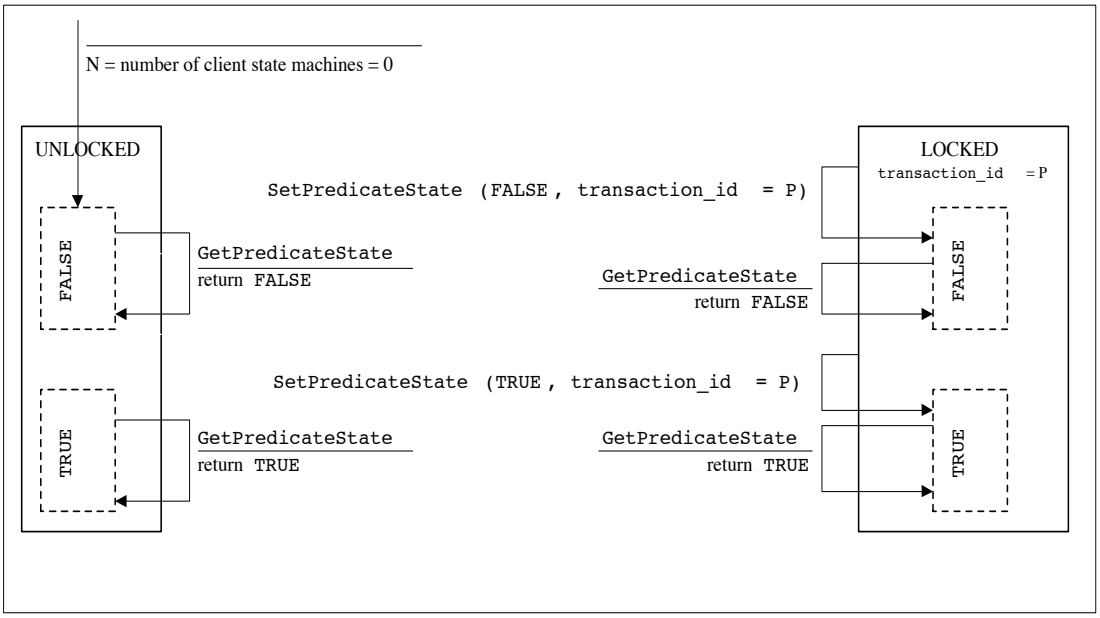

Figure 31—Predicate dependent mutex state machine for a Condition Variable Service

When a client invokes the `Wait` operation on the Condition Variable Service, a thread of control defined by the state machine in Figure 32 shall be established as a member of the collection. This thread of control makes `Lock`, `Unlock`, and `GetPredicateState` calls on the mutex associated with the Condition Variable Service.

There may be N such state machines in the collection running concurrently where N is implementation-specific. When the Condition Variable Service Object is instantiated, the default predicate state shall be UNLOCKED and there shall be no pending `Wait` state machines in the collection.

The state machines that are members of the collection, each of which represents a `Wait`, call on the Service shall behave as defined in Figure 32.

Unless otherwise indicated in the state diagram, when calls return the `MajorReturnCode` field shall be `MJ_COMPLETE` and the `MinorReturnCode` field shall be `MI_NO_ADDITIONAL_INFORMATION`.

All state transitions for the `IEEE1451_ConditionVariableService` shall be atomic. Where there are multiple actions associated with the transition, the result of the transition shall be exactly as though these actions executed without interference from Concurrent Execution of any operation on any object in the process space.

 Copyright © 2000 IEEE. All rights reserved.

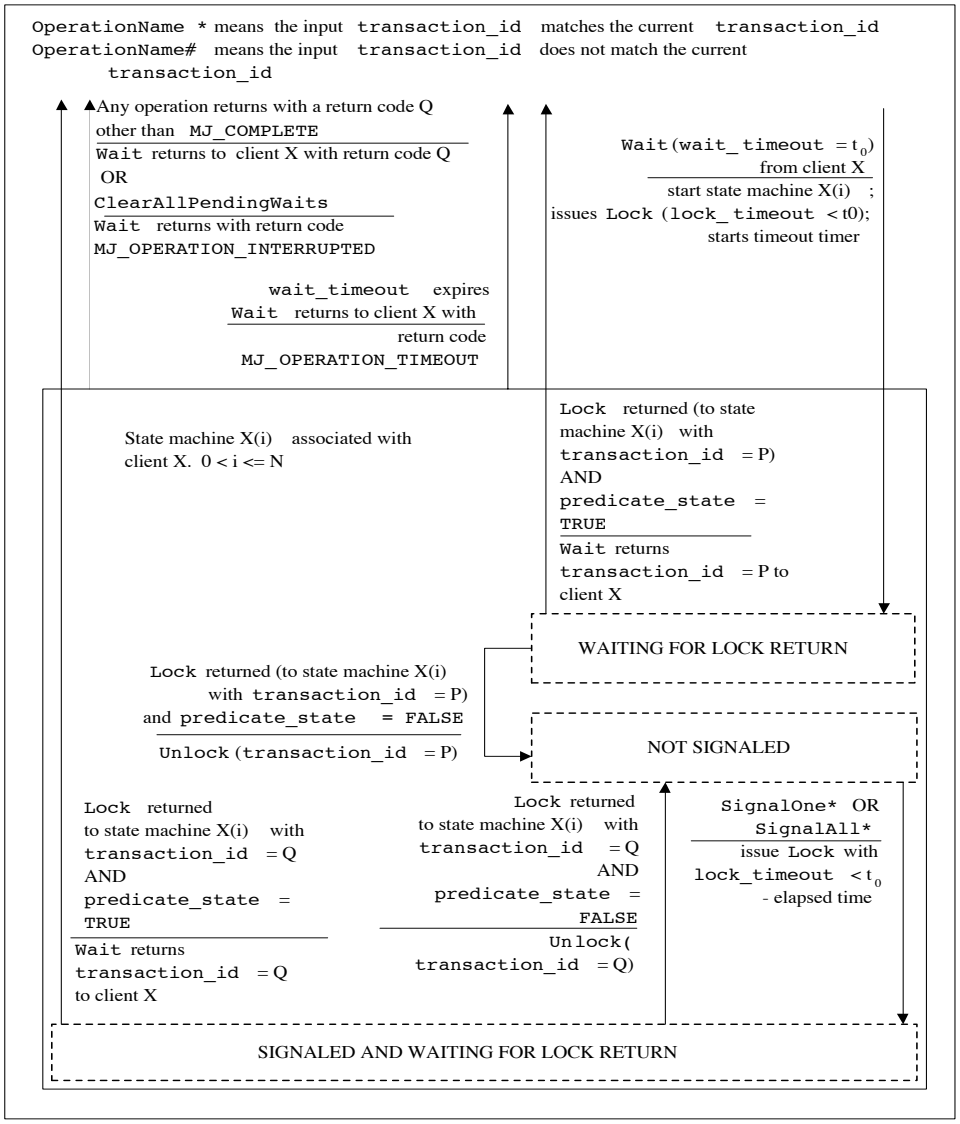

Figure 32—Wait state machine for a Condition Variable Service

12. Publication semantics

12.1 Overview

12.1.1 Main components of publications

Publications have four main components

— A Publication Key that may be used to define syntax and semantics of the publication

— A Publication Topic that may be used to further define syntax and semantics or may be used to define the semantics of the interaction between Publishers and Subscribers with respect to acceptance of publications

Copyright © 2000 IEEE. All rights reserved.

— Publication Contents that may be used to contain application or class-specific information

— A Publication Domain that defines the distribution scope for the publication

12.1.2 Selection of publications by subscribers

Subscriber Ports select publications on the basis of three main selectors

— A Subscription Key that is used to select against Publication Key values

— A Subscription Qualifier that may be used to select against Publication Topic values

— A Subscription Domain that defines the acceptance scope for receiving publications

12.2 Publishing operational behavior

The behavioral steps of this clause shall be implemented consistent with the memory management specifications of Clause 15. Encoding and decoding shall follow the rules of Clause 14.

The Argument Array, `publication_contents` passed to the Port by the Publisher shall be of the correct length to properly accommodate the contents. The number, type, and semantics of the member Arguments of the Argument Array `publication_contents` are publication-specific.

For Self Identifying Publisher Ports, the Port augments the received `publication_contents` Argument Array with Port-specific information to complete the encoding of the Publication Contents.

The Port shall pass the Publication Contents Argument Array to the underlying network infrastructure to be published.

The network infrastructure supporting the Publisher Port shall be responsible for marshaling

— The Publication Contents

— The Publication Topic

— The Publication Key

— The Publication Domain

into the on-the-wire representation appropriate for the underlying network communications.

The issuance of publications shall depend on the values of the Publication Domain, the Publication Key, and possibly the Publication Topic, according to the rules specified in Table 134. The return code columns indicate the values for the fields of the `OpReturnCode` of the publish semantics operation initiating the publication.

If permitted by the above constraints, the invocation of a local operation with publish semantics shall cause the issuance of a publication with the Publication Topic and Publication Key defined for the Port and Publication Contents defined by the caller. The publication shall be published into the Publication Domain defined for the Port.

 Copyright © 2000 IEEE. All rights reserved.

Table 134—Publication rules

Publication Domain	Publication Key	Publication Topic	Publication shall be issued	Major Return Code (Minor Return Code)
EMPTY_DOMAIN	Any value	Any value	NO	MJ_SERVICE_UNAVAILABLE (PC_EMPTY_DOMAIN)
Not the EMPTY_DOMAIN	PSK_NOT_SET	Any value	NO	MJ_SERVICE_UNAVAILABLE (PC_KEY_NOT_SET)
Not the EMPTY_DOMAIN	Any "terminal = no" value except PSK_NOT_SET	Any value except NOT_SET	YES	MJ_COMPLETE or as appropriate. (MI_NO_ADDITIONAL_INFORMATION or as appropriate.)
Not the EMPTY_DOMAIN	Any "terminal = no" value except PSK_NOT_SET	NOT_SET	NO	MJ_SERVICE_UNAVAILABLE (PC_TOPIC_NOT_SET)
Not the EMPTY_DOMAIN	Any "terminal = yes" value	Any value except NOT_SET	NO	MJ_SERVICE_UNAVAILABLE (PC_TOPIC_NOT_SET)
Not the EMPTY_DOMAIN	Any "terminal = yes" value	NOT_SET	YES	MJ_COMPLETE or as appropriate. (MI_NO_ADDITIONAL_INFORMATION or as appropriate)

12.3 Structure of publications

12.3.1 Publication Contents

The contents of a publication shall be of type `ArgumentArray`. The `TypeCode` and value of each member Argument of the Argument Array are specific to each publication.

12.3.2 Publication Keys and Subscription Keys

Values of Publication Key and Subscription Key shall be taken from the enumeration `PubSubKey`.

Values of Publication Key and Subscription Key indicated in the `PubSubKey` enumeration shown in Table 135 as Fixed Key = no, may

— Be set when the Publisher or Subscriber Port is instantiated

— Be set by Publishers or Subscribers via a implementation-specific local calls to the Publisher or Subscriber Ports, respectively

— Be modified by Publishers or Subscribers via implementation-specific local calls to the Publisher or Subscriber Ports, respectively

The behavior of systems where Publication Key or Subscription Key values have been modified after a system has been configured is outside the scope of this standard. Such modification is not recommended.

Values of Publication Key or Subscription Key, indicated in the `PubSubKey` enumeration shown in Table 135 as Fixed Key = yes, shall

— Be set when the Publisher or Subscriber Ports, respectively, are instantiated

— Not be modified

Publication Key and Subscription Key shall have a default value: `PSK_NOT_SET`.

The Publication Key value `PSK_ANY_KEY` shall not be used.

Copyright © 2000 IEEE. All rights reserved.

The Subscription Key value `PSK_ANY_KEY` should be used with caution. The behavior of systems where this value is used is outside the scope of this standard.

IDL: enumeration `PubSubKey;`

Table 135—`PubSubKey` enumeration

Enumeration	Value	Fixed Key	Terminal
PSK_NOT_SET	0	No	No
PSK_USER_DEFINED_PUBLICATION_TOPIC	1	No	No
PSK_NCAPBLOCK_ANNOUNCEMENT	2	Yes	Yes
PSK_NETWORK_VISIBLE_SERVER_OBJECT_PROPERTIES	3	Yes	Yes
PSK_REQUEST_NCAPBLOCK_ANNOUNCEMENT	4	Yes	Yes
PSK_FORCE_NCAPBLOCK_ANNOUNCEMENT	5	Yes	Yes
PSK_REQUEST_NETWORK_VISIBLE_SERVER_OBJECT_PROPERTIES	6	Yes	Yes
PSK_NCAPBlOCK_GO_ACTIVE	7	Yes	No
PSK_TIMER_TICK	8	No	No
PSK_REQUEST_TIMER_PROPERTIES	9	No	No
PSK_TIMER_PROPERTIES	10	No	No
PSK_PHYSICAL_PARAMETRIC_DATA	11	No	No
Reserved	12–63		
PSK_USER_DEFINED_PUBLISHER_IDENTIFYING_PUBLICATION	64	No	No
PSK_EVENT_NOTIFICATION	65	No	No
PSK_PHYSICAL_PARAMETRIC_DATA_PUBLISHER_IDENTIFYING	66	No	No
Reserved	67–127		
Open to industry	128–191		
Open to industry	192–254		
PSK_ANY_KEY	255	Yes	No

12.3.3 Publisher Port restrictions based on Publication Keys

The class of Publisher Port used for each Publication Key shall be defined by Table 136.

Publications with Publication Key values of Port class A in this table shall use ports of class `IEEE1451_PublisherPort` or a subclass thereof.

Publications with Publication Key values of Port class B in this table shall use ports of class `IEEE1451_SelfIdentifyingPublisherPort` or a subclass thereof.

12.3.4 Publication Topics and Subscription Qualifiers

Both Publication Topic and Subscription Qualifier are of type `OctetArray`.

 Copyright © 2000 IEEE. All rights reserved.

Table 136—Publisher Key, Port Class relationship

PubSubKey enumeration	Value	Port class
PSK_NOT_SET	0	N/A
PSK_USER_DEFINED_PUBLICATION_TOPIC	1	A
PSK_NCAPBLOCK_ANNOUNCEMENT	2	
PSK_NETWORK_VISIBLE_SERVER_OBJECT_PROPERTIES	3	
PSK_REQUEST_NCAPBLOCK_ANNOUNCEMENT	4	
PSK_FORCE_NCAPBLOCK_ANNOUNCEMENT	5	
PSK_REQUEST_NETWORK_VISIBLE_SERVER_OBJECT_PROPERTIES	6	
PSK_NCAPBLOCK_GO_ACTIVE	7	
PSK_TIMER_TICK	8	
PSK_REQUEST_TIMER_PROPERTIES	9	
PSK_TIMER_PROPERTIES	10	
PSK_PHYSICAL_PARAMETRIC_DATA	11	
Reserved	12–63	
PSK_USER_DEFINED_PUBLISHER_IDENTIFYING_PUBLICATION	64	B
PSK_EVENT_NOTIFICATION	65	
PSK_PHYSICAL_PARAMETRIC_DATA_PUBLISHER_IDENTIFYING	66	
Reserved	67–127	
Open to industry	128–191	A
Open to industry	192–254	B
PSK_ANY_KEY	255	N/A

The default value for the Publication Topic and the Subscription Qualifier shall be the zero length Octet Array. This default value shall be interpreted as the NOT_SET value.

Publisher Ports and Subscriber Ports that have keys indicated as “terminal = yes” in the `PubSubKey` enumeration shown in Table 135 shall restrict the values of their Publication Topic and Subscription Qualifier to the NOT_SET value.

For those that have keys indicated as “terminal = no”, Publication Topic and Subscription Qualifier shall be available for definition and use by IEEE 1451.1 component and application developers.

The values for each octet in a Publication Topic or Subscription Qualifier may assume any value.

Publication Topic and Subscription Qualifier octet values selected from the `Wildcard` enumeration shown in Table 137 have special meaning with respect to the matching criteria.

IDL: enumeration `Wildcard;`

Copyright © 2000 IEEE. All rights reserved.

Table 137—Wildcard enumeration

Enumeration	Value	Meaning
WC_ANY_OCTET	254	Wildcard: match to any octet.
WC_ANY_OCTET_ARRAY	255	Wildcard: match to any Octet Array.
Reserved	0–253	

12.3.4.1 Matching criteria used to select publications

A Subscriber Port shall accept a publication if and only if

— The Domain Match = MATCH

AND

— The Key Match = MATCH

AND

— The Topic Match = MATCH

The value of Domain Match is defined in 12.3.5.

The value of Key Match shall be as defined in the Key Match Table 138.

Table 138—Key Match

Subscription Key value	Key Match
PSK_NOT_SET	NO MATCH
Not the same value as the Publication Key.	NO MATCH
Same value as the Publication Key, but not PSK_NOT_SET.	MATCH
PSK_ANY_KEY	MATCH

An octet O_P in a Publication Topic shall match an octet O_S in a Subscription Qualifier as indicated in Table 139.

Table 139—Topic, Qualifier match table

$O_S \backslash O_P$	WC_ANY_OCTET	WC_ANY_ OCTET_ARRAY	Octet does not exist	Any other value
WC_ANY_OCTET	MATCH	MATCH	NO MATCH	MATCH
WC_ANY_OCTET_ARRAY	MATCH	MATCH	MATCH	MATCH
Octet does not exist	NO MATCH	MATCH	MATCH	NO MATCH
Any other value	MATCH	MATCH	NO MATCH	MATCH iff $O_P = O_S$

For the default Subscription Qualifier value, NOT_SET, the “Octet does not exist” condition shall apply to all octets.

The value of the “Topic Match” condition shall be defined by the following algorithm:

 Copyright © 2000 IEEE. All rights reserved.

Step 1: Start at the 0th Octet of both the Publication Topic and the Subscription Qualifier, do an Octet-to-Octet match.

Step 2: Continue matching until

— There is a nonmatch. In this case, the overall topic-to-qualifier match fails

— There is a match and at least one of the Octets is `WC_ANY_OCTET_ARRAY`. In this case, the overall topic-to-qualifier match succeeds

— All of the Octets match pairwise. In this case, the overall topic-to-qualifier match succeeds.

NOTE—The combinations of Publication Key and Publication Topic values in 12.2 preclude the issuance of a publication that is of no concern to the subscription acceptance algorithm.

Table 140 illustrates the operation of the topic-matching algorithm.

Table 140—Match example

Topic (or qualifier)	**ABC**	**ABCD**	**ABCD**	*****	**ABCD**	**ABCD**	**ABCD**	**NOT_SET**	**AB**	**AB**
Qualifier (or topic)	ABC	ABC	A??D	Any value	A*	B*	???	A?*	ABC*	AB*
Match ?	YES	NO	YES	YES	YES	NO	NO	NO	NO	YES

* = `WC_ANY_OCTET_ARRAY`
? = `WC_ANY_OCTET`
“Topic” refers to a Publication Topic and “Qualifier” refers to a Subscription Qualifier.

12.3.4.2 Restrictions on Publication Topic and Subscription Qualifier interpretation

There shall be no dependency on the interpretation of each octet of the Octet Array assumed by any operation or publication form under this standard outside those interpretations defined in Clause 12.

Any “print form” interpretation of the octet values of a Publication Topic or Subscription Qualifier is outside the scope of this standard.

NOTE—Publication Topics and Subscription Qualifiers are used in matching Publishers and Subscribers to establish the configurable aspects of publication. Configuration processes will need to compare or specify the Publication Topic of Publisher Ports and the Subscription Qualifier of Subscriber Ports. Users and configuration tool developers are cautioned to be sure that any implementation mappings of any human-readable forms of the Publication Topic or Subscription Qualifier onto the underlying Octet Array do not introduce ambiguities in the matching criteria. Of particular concern are the multiple ways of representing the same print form in some of the ISO 10646 multiple octet character sets [ISO/IEC DIS 10646-1: 1993 and ISO/IEC 10646-2: 1993].

12.3.5 Domains

Publication Domain and Subscription Domain values are of type `PubSubDomain` that is represented by 64 bit fields. Each of these bits shall correspond to a single logical Domain. The management of Domains is outside the scope of this standard.

The FULL_DOMAIN shall correspond to all bits set to `TRUE`. An EMPTY_DOMAIN shall correspond to all bits set to `FALSE`. The FULL_DOMAIN shall be the default value for Domains.

Copyright © 2000 IEEE. All rights reserved.

The match condition “Domain Match” between a Publication Domain and a Subscription Domain shall be MATCH if, and only if, the bitwise AND operation on the two Domains is `TRUE`. That is if, and only if, the logical intersection of the publication and subscription Domains is not the EMPTY_DOMAIN.

The purpose of Domains is to provide a scope for publications.

13. Defined publications

The following are publications not otherwise defined in connection with a class.

13.1 Publication `Timer_Tick` specification

Description: The `Timer_Tick` publication is used to distribute the occurrence of time intervals measured on the clock of the publisher to other objects in the system.

Publication Key: `PSK_TIMER_TICK`

Publication Topic: User-defined

Publisher Port Object Name: It is recommended that the Port Object Name be `Timer_Tick`

If the Port’s name is `Timer_Tick` or `Timer_TickX`, where X is an ordinal number, the Publisher Port’s Object Name shall be represented in the IEEE 1451.1 character set defined in Annex F.

13.1.1 Publication Contents

The Arguments of the Publication Contents Argument Array structure shall be as shown in Table 141.

Table 141—`Timer_Tick` Publication Contents

Argument Array Member	Member Argument `TypeCode` corresponding to type	Argument Name
0	`UInteger32`	`major_tick_number`
1	`UInteger32`	`minor_tick_number`
2	`TimeRepresentation`	(Optional) `last_major_tick_timestamp`

13.1.1.1 argument `major_tick_number` specification

The value of `major_tick_number` shall order events defining major time intervals.

13.1.1.2 argument `minor_tick_number` specification

The value of `minor_tick_number` shall order events defining minor time intervals.

 Copyright © 2000 IEEE. All rights reserved.

13.1.1.3 argument `last_major_tick_timestamp` specification

The value of the optional `last_major_tick_timestamp` shall specify the time of occurrence of the major time interval designated by the current `major_tick_number.` The `last_major_tick_timestamp` shall

— Specify the time of occurrence of the major time interval designated by the current `major_tick_number`

— Be relative to the `epoch`; see 13.2.1

13.1.2 Publication `Timer_Tick` behavior

This `Timer_Tick` publication shall be used if a pattern of major and minor time intervals is to be established within a Publication Domain. Such a pattern shall consist of ordered events separated by a time interval `duration_of_major_tick`. The specification of means for generating such time intervals is outside the scope of this standard.

The major time interval shall be divided into `number_of_minor_ticks_per_major_tick` equally spaced ordered events. The `minor_tick_number` shall be zero for the minor tick interval which starts at a major tick event.

NOTE—The largest `minor_tick_number` is therefore one less than the `number_of_minor_ticks_per_major_tick`.

13.2 Publication `Timer_Properties` specification

Description: The `Timer_Properties` publication is used to distribute the additional data, describing the properties of the time intervals distributed by the `Timer_Tick` publication.

Publication Key: `PSK_TIMER_PROPERTIES`

Publisher Port Object Name: It is recommended that the Port's Object Name be `Timer_Properties`

If the name is `Timer_Properties` or `Timer_PropertiesX,` where X is an ordinal number, the Publisher Port's Object Name shall be represented in the IEEE 1451.1 character set defined in Annex F.

13.2.1 Publication Contents

The Argument members of the Publication Contents Argument Array shall be shown in Table 142.

13.2.1.1 argument `major_tick_number` specification

The value of `major_tick_number` shall order events defining major time intervals.

13.2.1.2 argument `minor_tick_number` specification

The value of `minor_tick_number` shall order events defining minor time intervals.

13.2.1.3 argument `last_major_tick_timestamp` specification

The value of `last_major_tick_timestamp` shall be the value of the time of occurrence of the last major tick event. If the time of occurrence is not available, the default value for the

Copyright © 2000 IEEE. All rights reserved.

Table 142—`Timer_Properties` Publication Contents

Argument Array Member	Member Argument `TypeCode` corresponding to type	Argument Name
0	`UInteger32`	`major_tick_number`
1	`UInteger32`	`minor_tick_number`
2	`TimeRepresentation`	`last_major_tick_timestamp`
3	`TimeRepresentation`	`duration_of_major_tick`
4	`UInteger32`	`number_of_minor_ticks_per_major_tick`
5	`UInteger8`	`epoch_representation`
6	`TimeRepresentation`	`epoch`
7	`Uncertainty`	`time_uncertainty`

`TimeRepresentation` type encoded into the argument `last_major_tick_timestamp` shall be used.

13.2.1.4 argument `duration_of_major_tick` specification

The value of `duration_of_major_tick` shall be the time interval between major ticks.

13.2.1.5 argument `number_of_minor_ticks_per_major_tick` specification

The value of `number_of_minor_ticks_per_major_tick` shall be the number of minor ticks between each major tick.

13.2.1.6 argument `epoch_representation` specification

The allowed values for the value of `epoch_representation` shall be defined by the `TimeEpoch` enumeration shown in Table 143. This enumeration defines the method of specifying the epoch (origin of the timescale) on which the values of `last_major_tick_timestamp` are based.

IDL: enumeration `TimeEpoch;`

Table 143—TimeEpoch enumeration

Enumeration	Value	Meaning
`TE_NOT_DEFINED`	0	The epoch is either not defined or not known.
`TE_UTC`	1	The epoch shall be the UTC defined epoch.
`TE_USER_DEFINED`	2	The epoch shall be defined by the user.
Reserved	3–127	
Open to industry	128–255	

 Copyright © 2000 IEEE. All rights reserved.

13.2.1.7 argument `epoch` specification

The value of `epoch` shall be the origin of the timescale on which the values of `last_major_tick_timestamp` are based.

For an `epoch_representation` value of `TE_UTC`, the epoch is 0h MJD 40587.0, which corresponds to the time 0h on 1 January 1970 according to the Gregorian calendar [B7].

13.2.1.8 argument `time_uncertainty` specification

The value of `time_uncertainty` shall be an `Uncertainty` data structure and shall represent the error bounds on the all times associated with `Timer_Tick` publications. The `time_uncertainty` shall be based on time expressed in seconds.

13.2.2 Publication `Timer_Properties` behavior

The `Timer_Properties` publication shall be issued once for each receipt via a suitable Subscriber Port of the publication `Request_Timer_Properties`.

13.3 Publication `Request_Timer_Properties` specification

Description: The `Request_Timer_Properties` publication is used to request the publication of `Timer_Properties`.

Publication Key: `PSK_REQUEST_TIMER_PROPERTIES`

Publication Topic: User-defined

Publisher Port Object Name: It is recommended that the Port's Object Name be `Request_Timer_Properties`

If the name is `Request_Timer_Properties` or `Request_Timer_PropertiesX` where X is an ordinal number, the Publisher Port's Object Name shall be represented in the IEEE 1451.1 character set defined in Annex F.

13.3.1 Publication Contents

This publication shall not have any contents.

13.3.2 Publication `Request_Timer_Properties` behavior

The `Timer_Properties` publication shall be issued once for each receipt via a suitable Subscriber Port of the publication `Request_Timer_Properties`.

13.4 Publication `Physical_Parametric_Data` specification

Description: The `Physical_Parametric_Data` publication is used to distribute data that, in a client-server model, would be represented by an Object whose class is a subclass of `IEEE1451_PhysicalParameter` or a group of such Objects.

Publication Key: `PSK_PHYSICAL_PARAMETRIC_DATA`

Publication Topic: User-defined

Copyright © 2000 IEEE. All rights reserved.

Publisher Port Object Name: It is recommended that the Port's Object Name be `Physical_Parametric_Data`

If the name is `Physical_Parametric_Data` or `Physical_Parametric_DataX`, where X is an ordinal number, the Publisher Port's Object Name shall be represented in the IEEE 1451.1 character set defined in Annex F.

13.4.1 Publication Contents for a single Parameter

The Argument elements of the Publication Contents Argument Array structure shall be shown in Table 144.

Table 144—`PHYSICAL_PARAMETRIC_DATA` Single parameter Publication Contents

Argument Array Member	Member Argument `TypeCode` corresponding to type	Argument Name
0	`UInteger8`	`content_code`
1	`PhysicalParameterData`	`physical_parameter_data`
2	`PhysicalParameterMetadata`	`physical_parameter_metadata`

13.4.1.1 argument `content_code` specification

The value of `content_code` shall be a member of the following subset of the `PublicationContentCode` enumeration (see 11.8.3.4.3):

— `PCC_NORMAL_DATA`

— `PCC_METADATA`

— `PCC_BOTH_META_AND_NORMAL_DATA`

13.4.1.2 argument `physical_parameter_data` specification

The value of `physical_parameter_data` shall be a `PhysicalParameterData` union structure containing the published value of the Parameter. This union has as discriminant the `PhysicalParameterType` of the Physical Parameter whose information is being published. The value of the union is either a `PhysicalParameterSingletonData` structure or a `PhysicalParameterSeriesData` structure.

The value of `physical_parameter_data` shall be the `PhysicalParameterData` union structure returned as the 0th member of the output Argument Array returned by a `Read` operation on the Parameter whose value is being published.

This member Argument of the publication shall be the Argument with `TypeCode` `EMPTY_TC` in the case where the `content_code` Argument value is `PCC_METADATA`.

13.4.1.3 argument `physical_parameter_metadata` specification

The value of `physical_parameter_metadata` shall be a `PhysicalParameterMetadata` union structure containing the published value of the Parameter's metadata. This union has as discriminant the `PhysicalParameterType` of the Physical Parameter whose information is being published. The

 Copyright © 2000 IEEE. All rights reserved.

published physical_parameter_metadata shall be the same as that obtained by a GetMetadata on the Physical Parameter.

This member Argument of the publication shall not be present in the case where the content_code is PCC_NORMAL_DATA. That is, in this case the publication's content consists of a two element Argument Array whose first element contains the content_code PCC_NORMAL_DATA and whose second element contains the physical_parameter_data for the Parameter.

13.4.2 Publication Contents for multiple Parameters

The Argument elements of the Publication Contents Argument Array structure shall be shown in Table 145.

Table 145—Physical_Parameter_Data multiple parameter Publication Contents

Argument Array Member	Member Argument TypeCode corresponding to type	Argument Name
0	UInteger8	content_code
1	ArgumentArray	physical_parameter_datums
2	PhysicalParameterMetadataArray	physical_parameter_metadatums

13.4.2.1 argument content_code specification

The value of content_code shall be a member of the following subset of the PublicationContentCode enumeration (see 11.8.3.4.3):

— PCC_GROUPED_NORMAL_DATA
— PCC_GROUPED_METADATA
— PCC_GROUPED_BOTH_META_AND_NORMAL_DATA

13.4.2.2 argument physical_parameter_datums specification

The value of physical_parameter_datums shall be an Argument Array each element of which has as a value, a PhysicalParameterData union structure containing the published value of one of the Parameters in the group. The order of the Arguments in the Argument Array, that is the order of the group members' values, shall be the same for all publications of the specific Parameter group. This order is determined by the Publishers.

Each such PhysicalParameterData union structure contains the published value of a Parameter. This union has as discriminant the PhysicalParameterType of the Physical Parameter whose information is being published. The value of the union is either a PhysicalParameterSingletonData structure or a PhysicalParameterSeriesData structure.

The value of each such PhysicalParameterData union structure shall be the PhysicalParameterData union structure returned as the 0th member of the output Argument Array returned by a Read operation on the Parameter whose value is being published as this member Argument.

This member Argument of the publication shall be the Argument with TypeCode EMPTY_TC in the case where the content_code is PCC_GROUPED_METADATA.

Copyright © 2000 IEEE. All rights reserved.

13.4.2.2.1 argument `physical_parameter_metadatums` specification

The value of `physical_parameter_metadatums` shall be a `PhysicalParameterMetadataArray.` Each element of the Array is a `PhysicalParameterMetadata` union structure containing the metadata of one of the Parameters in the group. The ith element of `physical_parameter_metadatums` and the ith element of `physical_parameter_datums` shall correspond to the same group member Parameter. The contents of each element of the Array shall be the same as that obtained by a `GetMetadata` operation on the associated Physical Parameter.

This member Argument of the publication shall not be present in the case where the `content_code` is `PCC_GROUPED_NORMAL_DATA`.

13.5 Publication `Physical_Parametric_ Data_Publisher_Identifying` specification

Description: The `Physical_Parametric_Data_Publisher_Identifying` publication is used to distribute data that, in a client-server model, would be represented by an Object whose class is a subclass of `IEEE1451_PhysicalParameter` or a group of such Objects.

Publication Key:

`PSK_PHYSICAL_PARAMETRIC_DATA_PUBLISHER_IDENTIFYING`

Publication Topic: User-defined

Publisher Port Object Name: It is recommended that the Port's Object Name be `Physical_Parametric_Data_Publisher_Identifying.`

If the name is `Physical_Parametric_Data_Publisher_Identifying` or `Physical_Parametric_Data_Publisher_IdentifyingX`, where X is an ordinal number, the Publisher Port's Object Name shall be represented in the IEEE 1451.1 character set defined in Annex F.

13.5.1 Publication Contents for a single Parameter

The Arguments of the Publication Contents Argument Array structure shall be shown in Table 146.

Table 146—`Physical_Parametric_Data_Publisher_Identifying` single parameter Publication Contents

Argument Array Member	Argument Member `TypeCode` corresponding to type	Argument Name
0	`PublisherInformation`	`publisher_identification_information`
1	`UInteger8`	`content_code`
2	`PhysicalParameterData`	`physical_parameter_data`
3	`PhysicalParameterMetadata`	`physical_parameter_metadata`

 Copyright © 2000 IEEE. All rights reserved.

The publisher_identification_information shall have value a PublisherInformation structure as defined in 11.8.3.4.1. The Array elements 1, 2, and 3 shall be the same as elements 0, 1, and 2 of a Physical_Parametric_Data publication, respectively, as defined in 13.4.1.

13.5.2 Publication Contents for multiple Parameters

The Arguments of the Publication Contents Argument Array structure shall be shown in Table 147.

Table 147—Physical_Parametric_Data_Publisher_Identifying multiple parameter Publication Contents

Argument Array Member	Argument Member TypeCode corresponding to type	Argument Name
0	PublisherInformation	publisher_identification_information
1	UInteger8	content_code
2	ArgumentArray	physical_parameter_datums
3	PhysicalParameterMetadataArray	physical_parameter_metadatums

The publisher_identification_information shall have a value PublisherInformation structure as defined in 11.8.3.4.1. The Array elements 1, 2, and 3 shall be the same as elements 0, 1, and 2 of a Physical_Parametric_Data publication, respectively, as defined in 13.4.2.

14. Encode and decode rules

14.1 Argument encoding and decoding

14.1.1 Client responsibilities

In an IEEE 1451.1 system client-server communication, the client shall be responsible for

— Encoding the input arguments appearing in the requested Server Object operation's signature. These input arguments shall be encoded from their primitive or derived datatypes into the Argument Array used as the input argument for the Execute or ExecuteAsynchronous call on the Client Port or Asynchronous Client Port, respectively, associated with the target Server Object

— Decoding the Argument Array returned from the Execute or GetResult call on the Client Port or Asynchronous Client Port, respectively. The returned Argument Array shall be decoded into the primitive or derived datatypes of the output arguments appearing in the target Server Object operation's signature

14.1.2 Server Object responsibilities

In an IEEE 1451.1 system client-server communication, the Server Object shall be responsible for

— Decoding the input Argument Array used in the call to its Perform operation into the primitive or derived datatypes appearing in the requested server operation's signature

— Encoding the output arguments appearing in the requested server operation's signature from their

Copyright © 2000 IEEE. All rights reserved.

primitive or derived datatypes into the Argument Array used as the output arguments in the `Perform` call

14.1.3 Publisher responsibilities

In an IEEE 1451.1 system publish-subscribe communication, the Publisher shall be responsible for encoding a publication from its composition of primitive or derived datatypes to the Argument Array that is the `publication_contents` argument of a `Publish` or `PublishWithIdentification` operation.

14.1.4 Subscriber responsibilities

IEEE 1451.1 Application Subscribers shall be responsible for decoding a publication from the Argument Array `publication_contents` delivered as an argument of the callback operation to registered Subscribers back into the primitive or derived datatypes comprising the Publication Contents.

In order to support interoperability of IEEE 1451.1 NCAP nodes implemented in different processors, operating systems, and/or programming languages, the use of the algorithms specified in this clause to perform the encode/decode of the arguments appearing in the server operations and the publications specified in the IEEE 1451.1 standard shall be required.

The encode and decode algorithms for the arguments and publications specified in this standard are very simple since the datatypes of all arguments appearing in the signatures of IEEE 1451.1-defined operations and the members of all IEEE 1451.1-specified publications are IEEE 1451.1 datatypes.

For an IEEE 1451.1 subclass that defines operation signatures or publications using datatypes other than the IEEE 1451.1-specified primitive or derived datatypes, the subclass definition shall include the encoding and decoding algorithms for those signatures or publications. These algorithms shall encode or decode between the class-specific datatypes appearing in the signatures or publication and IEEE 1451.1 datatypes represented in the Argument Array used in client-server and publish-subscribe communications. If such a class-specific datatype includes as a member an IEEE 1451.1-specified primitive or derived datatype, the encoding and decoding algorithms for the class-specific datatype shall agree in their encoding and decoding of this member with the rules specified in Clause 14.

In Clause 14, the term “value of the Argument” shall be interpreted according to 6.2.14.

The declaration, storage allocation, and storage deallocation for all of datums related to argument and publication encoding and decoding shall follow the memory management rules of Clause 15.

14.2 Client-server argument encoding and decoding

NOTE—In implementing the encode/decode rules, note that 6.2.1 mandates that implementations provide a mechanism for knowing the length of arrays. This information will be needed in implementing the encode/decode mechanisms.

14.2.1 Client-side encoding and decoding rules

The signature of the remote operation being invoked by the client is of the following form:

```
OpReturnCode <Operation>(
           in type_0_in input_argument_0,
           ...,
           in type_i_in input_argument_i,
           ...,
           in type_last_in input_argument_last,
```

 Copyright © 2000 IEEE. All rights reserved.

```
            out type_0_out output_argument_0,
            ...,
            out type_j_out output_argument_j,
            ...,
            out type_last_out output_argument_last);
```

The signatures of `Execute`, `ExecuteAsynchronous`, and `GetResult` are as follows:

```
ClientServerReturnCode Execute(
            in UInteger8 execute_mode,
            in UInteger16 server_operation_id,
            in ArgumentArray server_input_arguments,
            out ArgumentArray server_output_arguments);

OpReturnCode ExecuteAsynchronous(
            in UInteger16 server_operation_id,
            in ArgumentArray server_input_arguments,
            out UInteger16 transaction_id);

ClientServerReturnCode GetResult(
            in UInteger16 transaction_id,
            out ArgumentArray server_output_arguments);
```

Encoding:

Prior to the invocation by a client of the `Execute` or `ExecuteAsynchronous` operation on a Client Port or Asynchronous Client Port, respectively, the client shall use the following encoding algorithm to encode the input arguments for the requested server operation into the Argument Array, `server_input_arguments`.

Starting with `input_argument_0`

— The `TypeCode` field of the 0th member Argument of `server_input_arguments` shall be assigned the `TypeCode` corresponding to `type_0_in`

— The value of the 0th member Argument of `server_input_arguments` shall be assigned the value of `input_argument_0`

— Repeat in turn for each input Argument

Decoding:

Following the return to the client from the `Execute` or `GetResult` operation on the Client Port or Asynchronous Client Port, respectively, the client shall use the following decoding algorithm to decode the returned Argument Array, `server_output_arguments` into the server operation's output arguments.

Starting with the 0th member Argument of `server_output_arguments`

— The value of `output_argument_0` shall be assigned the value of that Argument. The value shall be represented in the datatype `type_0_out.`

— Repeat in turn for each output Argument.

Copyright © 2000 IEEE. All rights reserved.

14.2.2 Server-side encoding and decoding rules

The signature of the server operation being invoked by the client is that defined in 14.2.1.

The signature of `Perform` is of the following form:

```
ClientServerReturnCode Perform(
           in UInteger16 server_operation_id,
           in ArgumentArray server_input_arguments,
           out ArgumentArray server_output_arguments);
```

Decoding:

Following the call to `Perform` from the underlying network infrastructure, `Perform` shall use the following decoding algorithm to decode the Argument Array, `server_input_arguments`, passed to `Perform` into the target operations' input arguments.

Starting with the 0th member Argument of `server_input_arguments`

— The value of `input_argument_0` shall be assigned the value of that Argument. This value shall be represented in the datatype `type_0_in`

— Repeat in turn for each member Argument of `server_input_arguments`

Encoding:

Following the return to `Perform` from the server operation, `Perform` shall use the following encoding algorithm to encode the operations output arguments into the Argument Array, `server_output_arguments` passed to `Perform` by the underlying network infrastructure.

Starting with 0th member Argument of `server_output_arguments`

— The `TypeCode` field of the 0th member Argument of `server_output_arguments` shall be assigned the `TypeCode` corresponding to `type_0_out`

— The value of the 0th member Argument of `server_output_arguments` shall be assigned the value of `output_argument_0`

— Repeat in turn for each output Argument

14.3 Publish-side argument encoding

14.3.1 Publisher-side encoding rules

There are two publishing operations

— `Publish` on an Publisher Port

— `PublishWithIdentification` on a Self Identifying Publisher Port

Argument encoding for these two operations shall be as specified in the following clauses.

 Copyright © 2000 IEEE. All rights reserved.

14.3.1.1 Publishing using the `Publish` operation

The contents of a publication published using the `Publish` operation on a Publisher Port Object consists of a sequence of publication members where each member is an IEEE 1451.1 primitive or derived datatype

```
type_0 publication_member_0,
...,
type_i publication_member_i,
...,
type_last publication_member_last
```

Here, `type_i` is the datatype of the ith publication member.

The signature of the `Publish` operation is as follows:

```
OpReturnCode Publish(in ArgumentArray publication_contents);
```

Prior to invoking the `Publish` operation, the Publisher shall be responsible for encoding the publication's publication members into the member Arguments of the Argument Array, `publication_contents`. The Publisher shall use the following algorithm to encode the Argument Array.

Encoding:

Starting with `publication_member_0`

- The `TypeCode` field of the 0th member Argument of the Argument Array shall be assigned the `TypeCode` corresponding to `type_0`
- The value of the 0th member Argument of the Argument Array shall be assigned the value of `publication_member_0`
- Repeat in turn for each publication member

14.3.1.2 Publishing using the `PublishWithIdentification` operation on a Self-Identifying Publisher Port

The contents of a publication published with the `PublishWithIdentification` operation on a Self-Identifying Publisher Port Object consists of (see 11.8.3.4)

- A datum of type `PublisherInformation` followed by
- A datum of type `UInteger8` representing a member of the `PublicationContentCode` enumeration giving the publication's `content_code` followed by
- A sequence of publication members where each member is an IEEE 1451.1 primitive or derived datatype

```
type_0 publication_member_0,
...,
type_i publication_member_i,
...,
type_last publication_member_last
```

Here, `type_i` is the datatype of the ith publication member.

The signature of the `PublishWithIdentification` operation on a Self Identifying Publisher Port is as follows:

```
OpReturnCode PublishWithIdentification(
        in ArgumentArray publication_contents);
```

Copyright © 2000 IEEE. All rights reserved.

Encoding:

Prior to invoking the PublishWithIdentification operation, the Publisher shall be responsible for encoding the publication's publication members into the member Arguments of the Argument Array, publication_contents. The Publisher shall use the following algorithm to encode the Argument Array:

Step 1: The Publisher defined fields, publicationID and publicationChangeID of the publisher_identification_information structure shall be assigned values as defined in 11.8.3.4.

Step 2: The TypeCode field of member Argument 0 of publication_contents shall be assigned the value PUBLISHER_INFORMATION_TC.

Step 3: The value of member Argument 0 of publication_contents shall be assigned the value of publisher_identification_information.

Step 4: The TypeCode field of member Argument 1 of publication_contents shall be assigned the value UINTEGER8_TC.

Step 5: The value of member Argument 1 of publication_contents shall be assigned the value of content_code.

Step 6: Starting with publication_member_0:

Step 7: The TypeCode field of the 2nd member Argument of publication_contents shall be assigned the TypeCode corresponding to type_0.

Step 8: The value of the 2nd member Argument of publication_contents shall be assigned the value of publication_member_0.

Step 9: Repeat Steps 7 and 8 in turn for each remaining publication member.

The Port shall be responsible for assigning and encoding the value of the portObjectTag member field of publisher_identification_information into the corresponding member field of member Argument 0 of the publication_contents Argument Array.

14.3.1.3 Publishing using the PublishWithIdentification operation on an Event Generator Publisher Port

The contents of a publication published with the PublishWithIdentification operation on an Event Generator Publisher Port Object (see 11.9.2.7.1) consists of

- A datum of type PublisherInformation
- A datum of type UInteger8 representing a member of the PublicationContentCode enumeration giving the publication's content_code
- A datum of type UInteger32 representing an event_sequence_number
- A datum of type ObjectTag representing the response_object_tag
- A datum of type TimeRepresentation representing the event_detection_time
- A datum of type String representing the event_name
- A sequence of zero or more optional publication members where each member is an IEEE 1451.1 primitive or derived datatype

```
type_0 optional_publication_member_0,
...,
```

 Copyright © 2000 IEEE. All rights reserved.

```
            type_i optional_publication_member_i,
            ...,
            type_last optional_publication_member_last
```

Here, `type_i` is the datatype of the ith optional publication member.

The signatures of the `PublishWithIdentification` operation is as follows:

```
OpReturnCode PublishWithIdentification(

            in ArgumentArray publication_contents);
```

Encoding:

Prior to invoking the `PublishWithIdentification` operation, the Publisher shall be responsible for encoding the publication members into the member Arguments of the Argument Array, `publication_contents`. The Publisher shall use the following algorithm to encode the Argument Array.

Step 1: The Publisher defined fields, `publicationID` and `publicationChangeID` of the `publisher_identification_information` structure, shall be assigned values according to 11.8.3.4.

Step 2: The `TypeCode` field of member Argument 0 of `publication_contents` shall be assigned the value `PUBLISHER_INFORMATION_TC.`

Step 3: The value of member Argument 0 of `publication_contents` shall be assigned the value of `publisher_identification_information.`

Step 4: The `TypeCode` field of member Argument 1 of `publication_contents` shall be assigned the value `UINTEGER8_TC.`

Step 5: The value of member Argument 1 of `publication_contents` shall be assigned the value of `content_code`.

Step 6: The `TypeCode` field of member Argument 2 of `publication_contents` shall be assigned the value `UINTEGER32_TC.`

Step 7: The value of member Argument 2 of `publication_contents` shall be assigned by the publishing Port.

Step 8: The `TypeCode` field of member Argument 3 of `publication_contents` shall be assigned the value `OBJECT_TAG_TC.`

Step 9: The value of member Argument 3 of `publication_contents` shall be assigned by the publishing Port.

Step 10: The `TypeCode` field of member Argument number 4 `publication_contents` shall be assigned the value `TIME_REPRESENTATION_TC.`

Step 11: The value of member Argument 4 of `publication_contents` shall be assigned the value `event_detection_time.`

Step 12: The `TypeCode` field of member Argument number 5 of `publication_contents` shall be assigned the value `STRING_TC`.

Step 13: The value of member Argument 5 of `publication_contents` shall be assigned the value `event_name.`

Copyright © 2000 IEEE. All rights reserved.

Step 14: Starting with `optional_publication_member_0:`

Step 15: The `TypeCode` field of member Argument 6 of `publication_contents` shall be assigned the `TypeCode` corresponding to `type_0`.

Step 16: The value of member Argument 6 of `publication_contents` shall be assigned the value of `publication_member_0`.

Step 17: Repeat Steps 15 and 16 in turn for each remaining `optional_publication_member`.

The publishing Port shall be responsible for assigning and encoding the value of

— The `objectTag` member field of `publisher_identification_information` into the corresponding member field of member Argument 0 of the `publication_contents` Argument Array received from the Publisher.

— The `event_sequence_number` as the value of member Argument 2 of the `publication_contents` Argument Array received from the Publisher.

— The `response_object_tag` as the value of member Argument 3 of the `publication_contents` Argument Array received from the Publisher.

14.4 Subscriber-side argument decoding

The Subscriber declares a set of local variables for the received publication, identified by its Publication Key and Publication Topic. The publication members are

```
type_0 publication_member_0,
...,
type_i publication_member_i,
...,
type_last publication_member_last
```

Here, `type_i` is the `datatype` of the ith member in the publication.

The signature of the `local subscription callback` operations is

```
void(
  in UInteger16 subscription_id,
  in UInteger8 publishing_port_publication_key,
  in PublicationTopic
     publishing_port_publication_topic,
  in ArgumentArray publication_contents);
```

The number and the datatypes of publication members encoded in the Argument Array, `publication_contents`, is identified by

— The `publishing_port_publication_key`

— The `publishing_port_publication_topic`

The three publication types defined by this standard are publications published using

— A Publisher Port

— A Self Identifying Publisher Port

— An Event Generator Publisher Port

 Copyright © 2000 IEEE. All rights reserved.

14.4.1 Decoding publications published from a Publisher Port

For publications published using a Publisher Port, `publication_contents` is an Argument Array with each member corresponding directly to the publication members.

14.4.2 Decoding publication published from a Self Identifying Publisher Port

For publications published using an Self Identifying Publisher Port, the Argument members of the Argument Array `publication_contents are` defined in Table 148 (see 11.8.3.4):

Table 148—Self Identifying Publication Contents

Argument Array Member	Member TypeCode corresponding to type	Argument Name
0	`PublisherInformation`	`publisher_identification_information`
1	`UInteger8`	`content_code`
Remaining members	Datatype of the corresponding publication member.	`application_specific_contents`

14.4.3 Decoding publications published from an Event Generator Publisher Port

For publications published using an Event Generator Publisher Port, the Argument members of the `publication_contents` Argument Array are defined in Table 149; see 11.9.2.7.1.

Table 149—Event Generator Publication Contents

Argument Member	Member TypeCode corresponding to type	Argument Name
0	`PublisherInformation`	`publisher_identification_information`
1	`UInteger8`	`content_code`
2	`UInteger32`	`event_sequence_number`
3	`ObjectTag`	`response_object_tag`
4	`TimeRepresentation`	`event_detection_time`
5	`String`	`event_name`
Remaining members	Datatype of the corresponding publication member.	`optional_arguments`

Decoding:

In all three cases, the Subscriber shall be responsible for decoding the Argument Array `publication_contents`.

Case 1:

If the value of the input argument, `publishing_port_publication_key`, of the Subscriber callback operation signifies that the publication was published via the `Publish` operation, the Subscriber shall use the following decoding algorithm to decode the Argument Array, `publication_contents`:

Starting with the 0th member Argument of `publication_contents:`

Copyright © 2000 IEEE. All rights reserved.

Step 1: The value of publication_member_0 shall be assigned the value of the Argument. This value shall be represented in the datatype corresponding to type_0.

Step 2: Repeat in turn for each Argument member of publication_contents.

Case 2:

If the value of the input argument, publishing_port_publication_key, of the subscriber callback operation signifies that the publication was published via the PublishWithIdentification operation, using an Self Identifying Publisher Port, the Subscriber shall use the following decoding algorithm to decode the Argument Array, publication_contents:

Step 1: The value of a local variable of datatype PublisherInformation representing the Publisher's identification information shall be assigned the value of the 0th member Argument of publication_contents.

Step 2: The value of a local variable of datatype UInteger8 representing the value of the publication's content_code shall be assigned the value of the 1st member Argument of publication_contents.

Step 3: Starting with the 2nd member Argument of publication_contents:

Step 3a: The value of publication_member_0 shall be assigned the value of the Argument. This value shall be represented in the datatype type_0.

Step 3b: Repeat Step 3a in turn for each of the remaining member Arguments of publication_contents.

Case 3:

If the value of the input argument, publishing_port_publication_key, of the subscriber callback operation signifies that the publication was published via the PublishWithIdentification operation using an Event Generator Publisher Port, the Subscriber shall use the following decoding algorithm to decode the Argument Array, publication_contents:

Step 1: The value of a local variable of datatype PublisherInformation representing the publisher_identification_information shall be assigned the value of the 0th member Argument of publication_contents.

Step 2: The value of a local variable of datatype UInteger8 representing the value of the publication's content_code shall be assigned the value of the 1st member Argument of publication_contents.

Step 3: The value of a local variable of datatype UInteger32 representing the event_sequence_number shall be assigned the value of the 2nd member Argument of publication_contents.

Step 4: The value of a local variable of datatype ObjectTag representing the value of the response_object_tag shall be assigned the value of the 3rd member Argument of publication_contents.

Step 5: The value of a local variable of datatype TimeRepresentation representing the event_detection_time shall be assigned the value of the 4th member Argument of publication_contents.

Step 6: The value of a local variable of datatype String representing the value of the event_name shall be assigned the value of the 5th member Argument of publication_contents.

Step 7: Starting with the 6th member Argument of publication_contents:

 Copyright © 2000 IEEE. All rights reserved.

Step 7a: The value of `optional_publication_member_0` shall be assigned the value of the Argument. This value shall be represented in the `datatype type_0.`

Step 7b: Repeat Step 7a in turn for each of the remaining member Arguments of `publication_contents`.

15. Memory management rules

15.1 Applicability of memory management rules

An implementation of IEEE 1451.1 in a programming language where the program is responsible for allocating and deallocating memory for nonprimitive datatypes shall adhere to the memory management rules defined in this clause.

15.2 In process memory management

Memory management for component interactions within a single NCAP process shall be the responsibility of IEEE 1451.1 implementers and IEEE 1451.1 application developers. The remainder of Clause 15 defines the memory management rules for network communication–related operations.

15.3 Simple datatype memory management

For the IEEE 1451.1 simple primitive datatypes defined in 6.1.1, any necessary memory management shall be the responsibility of the code block in which they are declared.

15.4 Nonsimple datatype memory management

The store for the nonsimple IEEE 1451.1 datatypes used in network communications shall be managed by the various components that participate in such communications.

15.4.1 Client-server memory management

The flow of data in a client-server network communication is illustrated in Figure 33.

The memory management responsibility assignments for components involved in client-server communications shall be as defined in Table 150. The responsibility assignments are the same for both synchronous and asynchronous client-server communications.

The key numbers are those given in Figure 33.

15.4.2 Publish-subscribe memory management

Copyright © 2000 IEEE. All rights reserved.

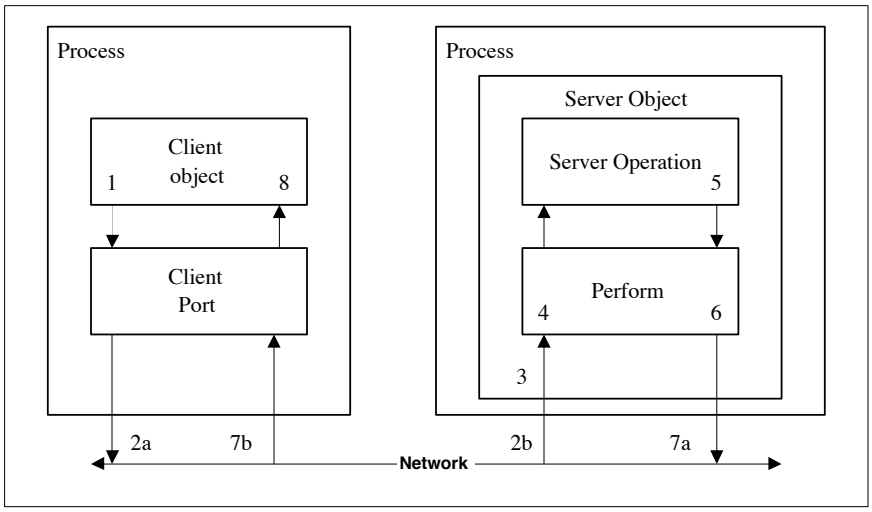

Figure 33—Client-server communication

The flow of data in a publish-subscribe network communication is illustrated in Figure 34.

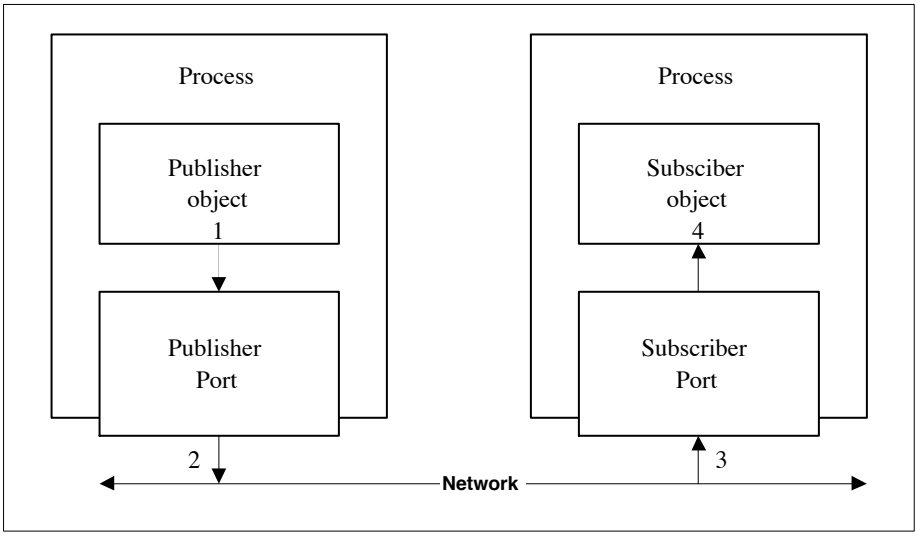

Figure 34—Publish-subscribe communication

The memory management responsibility assignments for components involved in publish-subscribe communications shall be as defined in Table 151. Table 151 applies to publications made using either ports of class `IEEE1451_PublisherPort`, or `IEEE1451_SelfIdentifyingPublisherPort` or subclasses thereof.

In the case of a Self Identifying Publisher Port, Table 151 specifies the memory management of the publication aspects of the communication. Any use of the self-identifying features use client-server communications and shall follow the memory management rules for such communications.

The key numbers are those given in Figure 34.

 Copyright © 2000 IEEE. All rights reserved.

Table 150—Client-server memory management

Key #	Operation	Declaration responsibility	Store allocation responsibility	Store deallocation responsibility
1	Client: Invoke the Client Port's or Asynchronous Client Port's `Execute` or `ExecuteAsynchronous` operation, respectively	—Argument Array `server_input_arguments` —Member Arguments of `server_input_arguments` —Reference to Argument Array `server_output_arguments`	—Argument Array `server_input_arguments` —Member Arguments of `server_input_arguments`	
2a	Client-side Network Infrastructure: Marshal onto network	Intermediate store for marshaling data onto the network	Intermediate store for marshaling data onto the network	Intermediate store for marshaling data onto the network
2b	Server-side Network Infrastructure: Demarshal from network	Intermediate store for demarshaling data from the network	Intermediate store for demarshaling data from the network	Intermediate store for demarshaling data from the network
3	Server-side Network infrastructure: Invoke the Server Object's `Perform`	—Argument Array `server_input_arguments` —Member Arguments of `server_input_arguments` —Reference to Argument Array `server_output_arguments`	—Argument Array `server_input_arguments` —Member Arguments of `server_input_arguments`	
4	`Perform`: Invokes server operation	—Server operation input arguments —References to server operation output arguments	Server operation input arguments	
5	Server operation executes and return to `Perform`	Intermediate store for operation	Intermediate store for operation. Server operation output arguments if needed (e.g., argument not a simple datatype)	Intermediate store for operation
6	`Perform` returns to server-side network infrastructure	Member Arguments of `server_output_arguments`	—Argument Array `server_output_argument` —Member Arguments of `server_output_arguments`	—Server operation input arguments —Server operation output arguments (may delegate deallocation responsibility for heap allocated server output arguments to the Network Infrastructure in 7a)
7a	Server-side Network Infrastructure: Marshal onto network	Intermediate store for marshaling output data onto the network	Intermediate store for marshaling output data onto the network	—Argument Array `server_input_arguments` —Member Arguments of `server_input_arguments` —Intermediate store for marshaling output data onto the network —Argument Array `server_output_arguments` —Member Arguments of `server_output_arguments`
7b	Client-side Network Infrastructure: Demarshal from network	—Intermediate store for demarshaling output data from the network —Member Arguments of `server_output_arguments`	—Intermediate store for demarshaling output data from the network —Argument Array `server_output_arguments` —Member Arguments of `server_output_arguments`	Intermediate store for demarshaling data from the network
8	Client Port: Return to client			—Argument Array `server_input_arguments` —Member Arguments of `server_input_arguments` —Argument Array `server_output_arguments` —Member Arguments of `server_output_arguments`

Copyright © 2000 IEEE. All rights reserved.

Table 151—Publish-subscribe memory management

Key #	Operation	Responsibility
1	Publisher invokes `Publish` or `PublishWithIdentification` on the Publishing Port	The Publisher declares, allocates, and deallocates all store for the publication's Publication Contents Argument Array.
2	Publishing side Network Infrastructure: Marshal onto network	The Publication side Network Infrastructure declares, allocates, and deallocates any intermediate store used in marshaling the publication onto the network.
3	Subscriber-side Network Infrastructure: Demarshal from network	The Subscriber-side Network Infrastructure declares, allocates, and deallocates any intermediate store used in demarshaling the publication from the network.
4	Subscriber Port: Callback Subscribers	The Subscriber Port declares and allocates the store for the received publication and invokes its subscribers' callback operations. It deallocates the store for the received publication, but not until all of the callbacks to its Subscribers have returned.

15.4.3 Mutex and Condition Variable Service memory management

Mutex Service and Condition Variable Service Objects are application interfaces to IEEE 1451.1 implementation provided mutex and condition variable functionality, respectively. The management of the store for these Mutex Service and Condition Variable Services is outside of the scope of this standard.

16. Conformance

16.1 Conformance objective

The philosophy underlying the conformance requirements of this clause is to provide the structure necessary to raise the level of interoperability of systems built to this standard, while leaving opportunity open for continued technical improvement and differentiation.

An IEEE 1451.1-conformant system provides the following benefits, not usually found in nonconforming systems:

— A uniform design model for system implementation

— Portable Application System designs

— A uniform and network independent set of operations for system configuration

— Network-independent models for system communication

— A network-independent abstraction layer and encode decode rules that isolate applications from the details of network communications

— Network-independent models for implementing application functionality

— A uniform information model for representing physical parametric data

— Uniform models for managing and representing event data, parametric data, and bulk data

— Uniform models for managing and representing time

 Copyright © 2000 IEEE. All rights reserved.

— Uniform models for intranode concurrency management and Components to manage internode concurrency

— Uniform models for memory management

In order not to unduly constrain application developers, in particular when integrating legacy systems with IEEE 1451.1 systems, IEEE 1451.1 conformance is defined for subsystems. Such a subsystem can constitute an entire system.

16.2 Definition of conformance terms

The following definitions for the following terms shall apply to all conformance clauses:

— IEEE 1451.1 class or subclass

— IEEE 1451.1 Object

— Subsystem

— Application subsystem

— Exists in the application subsystem

— Support

— IEEE 1451.X Transducer

— Communications

— IEEE 1451.1 Communication

— Network Communications

— Interact with the physical world

16.2.1 Conformance definition of the term IEEE 1451.1 class

The term “IEEE 1451.1 class” shall be interpreted as the class `IEEE 1451_Root` or any subclass thereof whether specified by this standard or by users.

16.2.2 Conformance definition of the term IEEE 1451.1 Object

The term “IEEE 1451.1 Object” shall mean an object whose class is an IEEE 1451.1 class.

16.2.3 Conformance definition of the term subsystem

The term “subsystem” of a system shall mean a subset of the system. Only a portion of the subsystem, possibly a null portion, need interact with other system entities. The subsystem may be the entire system.

16.2.4 Conformance definition of the term application subsystem

The term “application subsystem” shall mean all vendor- and user-supplied hardware, firmware, and software that implements or supports

— The end user application and application communications

— The configuration of the application and its application communications

— The maintenance of the application and its application communications

Copyright © 2000 IEEE. All rights reserved.

An application subsystem may include objects whose classes are user-defined subclasses of the IEEE 1451.1-specified classes as well as non-IEEE 1451.1 objects.

16.2.5 Conformance definition of the term exists in the application subsystem

For an IEEE 1451.1 Object in an application subsystem, the term "exists in the application subsystem" shall mean that

— The object is, or can be made operational in the application subsystem, see 5.1.3, via some sequence of operations

OR

— The object is or can be made operational in the application subsystem via a sequence of operations or procedures outside of the scope of this standard. The operations or procedures to be considered shall include but are not limited to power cycling of the device, the introduction of code in any form via any means, interaction with other objects, interaction with transducers, or by use of device-specific operator interfaces

16.2.6 Conformance definition of the term support

Given an application subsystem, the term "support" shall mean that for every IEEE 1451.1 Object that exists in the application subsystem all of the following conditions shall hold for the object's class:

— The implementation of the class shall include
  — All of the IEEE 1451.1-required behavior for the class including all required behavior inherited from parent classes following the class hierarchy up to and including `IEEE1451_Root`
  — All of the IEEE 1451.1-required operations for the class including all operations inherited from parent classes following the class hierarchy up to and including `IEEE 1451_Root`
  — All of the IEEE 1451.1-required publications or subscriptions for the class including all publications or subscriptions inherited from parent classes following the class hierarchy up to and including `IEEE 1451_Root`
  — All non-class-specific requirements of the IEEE 1451.1 standard
— The IEEE 1451.1 datatypes specified in the signature of any Network Visible operation of the class or in the format of any publication processed by the class shall be encoded and decoded following the rules of Clause 14.
— Any user-defined datatypes that appear in the signature of a Network Visible operation of the class or in the format of any publication processed by the class shall be encoded and decoded to and from a sequence of datums of type `Argument`. They shall follow rules that shall be included as part of the class specification. This sequence of datums shall be assigned to specific elements of the appropriate Argument Array appearing in the signature of `Perform`, `Execute`, `ExecuteAsynchronous`, `GetResult,` or in the Publication Contents of a publication
— The implementation shall follow the Memory Management requirements of Clause 15.

There is no requirement that an implementation provide support for a class if no instance exists in the application subsystem.

 Copyright © 2000 IEEE. All rights reserved.

16.2.7 Conformance definition of the term IEEE 1451.X Transducer

An "IEEE 1451.X Transducer" is a transducer based on one of the IEEE 1451 family of transducer interface standards, for example, [IEEE 1451.2]. The X refers to the suffix of the relevant standard.

16.2.8 Conformance definition of the term Communication

The term "Communication" shall mean the transfer of information between two objects in a system. The mechanism that provides this information transfer shall be termed the Communication infrastructure.

16.2.9 Conformance definition of the term IEEE 1451.1 Communication

The term "IEEE 1451.1 Communication" shall mean a Communication using mechanisms that shall be correctly interpreted by

— The `Perform` operation of an Object acting as a server whose class is a subclass of `IEEE1451_Entity`

— An Object whose class is a subclass of `IEEE1451_BaseClientPort`

— An Object whose class is a subclass of `IEEE1451_BasePublisherPort`

— An Object whose class is a subclass of `IEEE1451_SubscriberPort`

16.2.10 Conformance definition of the term IEEE 1451.1 Network Communications

The term "IEEE 1451.1 Network Communication" shall mean an IEEE 1451.1 Communication between objects in distinct process spaces either within a single NCAP or over the network between two NCAPs.

An IEEE 1451.1 Communication does not have to be an IEEE 1451.1 Network Communication, if both communication endpoints reside in the same process space; see 5.2.

An IEEE 1451.1 Communication between two process spaces in the same NCAP may use techniques such as shared memory not normally regarded as "networks." For purposes of conformance to this standard, all such techniques shall be regarded as networks and shall meet all of the conformance requirements associated with IEEE 1451.1 Network Communication.

16.2.11 Conformance definition of the term interaction with the physical world

The phrase "interaction with the physical world" shall be interpreted as a communication, in either direction, between an object in a process executing on an NCAP and a transducer, physically connected to that NCAP.

16.3 Conformance requirements

This clause defines the requirements that shall be met by a subsystem for it to be IEEE 1451.1 conformant.

16.3.1 Requirements for an IEEE 1451.1 subsystem

The overall requirements are stated in this clause and additional more detailed requirements are stated in 16.3.2, 16.3.3, 16.3.4, 16.3.5, 16.3.6, and 16.3.7.

A subsystem shall be deemed IEEE 1451.1-conformant if and only if

— The construction requirements of 16.3.2 are met

Copyright © 2000 IEEE. All rights reserved.

— The subsystem is constructed such that all objects in the subsystem Directly or Indirectly Owned by NCAP Block objects execute on NCAPs that are IEEE 1451.1 conformant according to 16.3.3

— All Network Communications involving the subsystem are implemented with IEEE 1451.1 conformant network implementations according to 16.3.7

— All networks used by the subsystem for IEEE 1451.1 Communications and that use a common network protocol shall also use the same on-the-wire format as the basis for marshaling and demarshaling of IEEE 1451.1 Communications

16.3.2 Construction requirements for IEEE 1451.1 conformance

An IEEE 1451.1 conformant subsystem shall meet the requirements of 16.3.2.1, 16.3.2.2, and 16.3.2.3.

16.3.2.1 Object ownership requirements

— Each subsystem code module shall execute in an NCAP process where the process contains precisely one instance of an `IEEE1451_NCAPBlock` class or subclass thereof. These NCAP Block instances shall be a part of the conformant subsystem.

— Any IEEE 1451.1 Object that is a part of the conformant subsystem and any non-IEEE 1451.1 object that is both part of the conformant subsystem and that participates in an IEEE 1451.1 Communication shall each be Directly or Indirectly Owned by the unique instance of an NCAP Block in each object's process space.

— Any object that is part of the conformant subsystem that interacts with the physical world shall be Directly or Indirectly Owned by the unique instance of an NCAP Block in each object's process space.

16.3.2.2 Interactions with the physical world

All conformant subsystem interactions with the physical world shall meet one of the following requirements:

— The interaction shall be between a transducer connected to an NCAP and an instance of a subclass of `IEEE1451_BaseTransducerBlock` executing in a process on the NCAP. The Transducer Block Object shall be a part of the conformant subsystem. This clause is intended for interactions with that portion of the physical world comprising those parts of the application plant being measured or controlled.

— The interaction shall be between a transducer connected to an NCAP and an instance of a subclass of `IEEE1451_Block` executing in a process on the NCAP. This instance Object shall be a part of the conformant subsystem. This clause is intended for interactions between a Function Block or an NCAP Block and that portion of the physical world comprising HMI transducers—e.g., keyboards, displays, printers, files—or other input or output devices not normally considered part of the plant.

16.3.2.3 Communications

All Communications involving an object that is part of a conformant subsystem shall meet the following requirements:

— If the object participates in an IEEE 1451.1 Communication then it shall be one of the following:

  — An instance of a subclass of `IEEE1451_Entity` acting as a server via its `Perform` operation
  OR

 Copyright © 2000 IEEE. All rights reserved.

  - An instance of a subclass of `IEEE1451_BaseClientPort`
    OR
  - An instance of a subclass of `IEEE1451_BasePublisherPort`
    OR
  - An instance of the `IEEE1451_SubscriberPort` class or subclass thereof

- If the Communication is between the object that is part of the conformant subsystem and an object that is not part of the conformant subsystem, the Communication shall be an IEEE 1451.1 Communication
- If the Communication is between two objects each of which is a part of the conformant subsystem and the two objects reside in different processes, then the communication shall be an IEEE 1451.1 Network Communication

16.3.3 Requirements for IEEE 1451 NCAPs

An NCAP shall be deemed conformant to IEEE 1451.1 provided

- It supports each IEEE 1451.1 class for which an instance exists in the application subsystem.
- If any transducer implementing one of the IEEE 1451 family of transducer standards is used with the application subsystem, for example, [IEEE 1451.2], the application subsystem shall conform to the requirements of 16.3.6.
- It provides conformance documentation that meets the requirements of 16.3.4 for all supported classes.
- It supports the conformance requirements of 16.3.5 for the minimum configuration of an IEEE 1451.1 NCAP.

16.3.4 Conformance documentation requirements for supported classes

An implementation of the standard is always in the context of

- A specific NCAP with its operating system
- A specific network
- A specific programming language

The documentation requirements are given with respect to this implementation context.

In this clause, the implementation's programming language is denoted by $\mathcal{P}$.

16.3.4.1 Datatype and reference documentation

The datatype documentation requirements for a conformant IEEE 1451.1 implementation are

- For each primitive datatype defined in the standard, the mapping of that datatype to a datatype in $\mathcal{P}$ shall be documented.
- For each derived datatype defined in the standard, the mapping of that datatype to constructs in $\mathcal{P}$ shall be documented.
- For each application developer visible datatype used in the implementation which is not an IEEE 1451.1-specified datatype, the following documentation shall be provided:
  - The IDL specification of that datatype.

Copyright © 2000 IEEE. All rights reserved.

  - The mapping of the datatype to constructs in $\mathcal{P}$.
  - The encoding and decoding rules for the datatype. That is, the rules by which the datatype is encoded/decoded to/from a sequence of Argument datatypes.
- For all application-developer-visible derived datatypes defined in the implementation
  - If $\mathcal{P}$ uses header files (for example, $\mathcal{P}$ is C or C++), the implementation shall provide header files defining the derived datatypes.
  - If $\mathcal{P}$ does not use header files but does use some type of interface construct for defining types (for example, $\mathcal{P}$ is Java), the implementation shall provide interface definitions defining the derived datatypes.
- The implementation shall document how an application accesses the members of an IEEE 1451.1 Array datatype and obtain the Array's length.
- The implementation shall document how an application determines prior to access, that an attempt to access the value of the IEEE 1451.1 discriminated union datatype will fail in the event that no value has been bound to the union's discriminate and value.
- Unless $\mathcal{P}$ directly provides a mechanism for accessing the members of a derived datatype, the implementation shall describe how an application does access such members

The documentation requirements for references in a conformant IEEE 1451.1 implementation are

- Documentation shall be provided for the syntax and the semantics of the implementation's network Object Dispatch Address reference type.
- The mapping of the reference `<local object reference>` to a construct in $\mathcal{P}$ shall be documented.
- The mapping of the reference `<local operation reference>` to a construct, or constructs, in $\mathcal{P}$ shall be documented.

16.3.4.2 Object class documentation

The object class documentation requirements for an IEEE 1451.1 implementation are

- The implementation shall provide a list of all supported classes, both IEEE 1451.1-specified classes and implementation-provided specializations of those classes.
- The implementation shall provide a list of all IEEE 1451.1-specified classes that are not supported.
- If $\mathcal{P}$ supports either header files or some type of interface construct, the implementation shall provide header files or interface definitions for each application-developer-visible object class supported by the implementation. This includes, in particular, the IEEE 1451.1-specified classes.
- If $\mathcal{P}$ supports neither header files or interface constructs, but does support other than pass-by-value calling semantics, then the implementation shall provide for each operation of each application-developer-visible object class supported by the implementation, the representation of the operation's arguments as they occur in an actual call to the operation.
- For each implementation-provided specialization of an IEEE 1451.1-specified class, the implementation shall provide an IDL specification of the specialized class. For each implementation-provided specialization of an IEEE 1451.1-specified class, if the specialized class has operations in addition to those inherited from the parent class, the implementation shall provide a behavioral description for each such operation.
- For each IEEE 1451.1-specified class that has optional operations, which of these optional operations have an Interface Only Implementation and which have Full Implementations shall be documented.

 Copyright © 2000 IEEE. All rights reserved.

— For each implementation provided specialization of an IEEE 1451.1-specified class that has a defined state machine and the specialization substates one of the parent class's states, the implementation shall provide a state machine specification for the specialized class.

— For each implementation provided specialization of an IEEE 1451.1-specified class, a behavior visible directly or indirectly at the interface shall be documented.

— All operation return code assignments not specified in this standard shall be documented.

— If any Objects of class `IEEE1451_File` or `IEEE1451_PartitionedFile` exist in the implementation, the implementation shall provide descriptions of the syntax and the semantics of the contents of all such Objects.

16.3.4.3 Publications

If the implementation includes any application-accessible publications beyond those specified in the IEEE 1451.1 standard, the implementation shall document the syntax and the semantics of these publications.

16.3.4.4 Object instantiation

Unless $\mathcal{P}$ directly provides a mechanism for creating instance objects of the supported IEEE 1451.1 Object classes and the implementation-provided specialization of these classes, the implementation shall document how an application creates such instance objects.

16.3.4.5 Local operation invocation

Unless it is immediately apparent from the mapping of the IEEE 1451.1 Object classes and the implementation provided specialization of these classes to $\mathcal{P}$, the implementation shall document how an application invokes local operations on instances of the classes.

NOTE—This could be the case if $\mathcal{P}$ is not an object-oriented language.

16.3.4.6 Enumerations

If the implementation defines any additional members of the IEEE 1451.1-specified enumerations that admit user extension; the implementation shall document the assignment of `UInteger8` values to these added members. In particular, the implementation shall document any extensions to the various IEEE 1451.1 return code enumerations.

If the implementation defines any enumerations beyond those specified in the IEEE 1451.1 standard, the implementation shall document the assignment of `UInteger8` values to the members of each such enumeration.

16.3.4.7 On the wire format

The implementation shall provide the specification of the network on-the-wire-format used by the network infrastructure. This format shall be specified by a reference to an industry standard for this format and/or by an explicit byte-level specification of the format.

16.3.4.8 Boot-up and configuration tools

If the implementation provides any IEEE 1451.1 system boot-up and/or system configuration tools, the implementation shall include documentation for using these tools. The documentation shall include the syntax and the semantics of the tools' programmatic interfaces. If the tools include any HMIs, the

Copyright © 2000 IEEE. All rights reserved.

documentation shall be complete enough so that the HMIs are usable by someone who is not conversant with the implementation's programming language, *P*.

16.3.5 Requirements for the minimum configuration of an IEEE 1451 NCAP

An NCAP implementation conformant to IEEE 1451.1, that is, an IEEE 1451 NCAP, shall include an NCAP Block. This NCAP Block shall

— Fully implement all of the mandatory operations and behaviors specified for the `IEEE1451_NCAPBlock` class

— Support the publications and subscriptions defined for the `IEEE1451_NCAPBlock` class along with the required behaviors

16.3.6 Requirements for application subsystems using IEEE 1451.X Transducers

An application subsystem that uses transducers implementing one of the IEEE 1451.X transducer standards shall represent the transducer by an `IEEE1451_DotXTransducerBlock` or a subclass thereof. The X stands for the suffix of the relevant IEEE 1451 transducer interface standard.

An application subsystem using transducers implementing one of the IEEE 1451.X Transducer standards shall be deemed IEEE 1451.X-conformant provided

— It is IEEE 1451.1 conformant

AND one of the following requirements is met:

— If the IEEE 1451.X standard defined–physical interface is implemented, the interface fully conforms in all respects to the relevant IEEE 1451.X standard.

— If the IEEE 1451.X standard–defined physical interface is not implemented or if the IEEE 1451.X transducer can not be removed or replaced except as a factory operation, then when invoked from the network, all of the operations on the DotX Transducer Block shall operate in all respects as if a full IEEE 1451.X-conformant physical interface was present. That is, it shall be impossible from the network to determine whether the physical interface to the transducer is actually implemented or not.

NCAPs that include transducers as an integral part of the device without exposing a separable interface are permitted.

16.3.7 Requirements for a network implementation for an IEEE 1451.1 subsystem

A network implementation providing IEEE 1451.1 Network Communication to a conformant IEEE 1451.1 NCAP shall be IEEE 1451.1-conformant provided

— It supports both the IEEE 1451.1-defined client-server and the publish-subscribe models and mechanisms of communication.

— It supports the marshaling and demarshaling of all `Argument` types defined in 6.2.14, in accordance with the specifications for the underlying network on-the-wire format.

— It supports Object Dispatch Addresses as defined in IEEE 1451.1 for the underlying network.

 Copyright © 2000 IEEE. All rights reserved.

Annex A

(informative)

Using the object model

A.1 Overview of application annexes

This annex, Annex B, Annex C, and Annex D provide an outline of a typical use of this standard in implementing systems. The example presented in this annex is used throughout the discussions in these annexes.

A.1.1 Context

The IEEE 1451.1 object model provides an extensible framework for the design and implementation of software for distributed measurement and control systems, consisting of networked NCAPs hosting smart transducers. Application System software is attained by mapping the required system functionality onto IEEE 1451.1 object types. The IEEE 1451.1 standard is network and NCAP neutral. Consequently, the same object-level design of an Application System can be realized on a multiplicity of networks and NCAPs.

The IEEE 1451.1 standard provides a set of object classes whose operations have completely specified interfaces and behaviors. Extensions of the design and implementation framework provided by these classes are realized by class specialization.

Vendors are expected to provide implementations of the standard as object libraries, targeted for a specific NCAP, network, and programming language. Application developers will use these products to build Application Systems. The standard provides a communications software abstraction layer. Using this abstraction layer, it is possible from different manufacturer's NCAPs, each hosting software developed from using different vendor's libraries, to interoperate. Annex E gives the necessary and sufficient conditions for such interoperation.

The software in an Application System consists of a collection of Block Objects, their Owned objects, and optionally other, non-Network Visible objects. Each NCAP process contains precisely one NCAP Block Object that provides information about and services to the resources managed and made available by the NCAP. In particular, an NCAP Block provides the interfaces necessary to configure the communication patterns between objects in a system.

If an NCAP hosts transducers, either sensors and/or actuators, the transducers are modeled by using Transducer Block Objects and Physical Parameter Objects Owned by the Transducer Blocks. The Transducer Block and Physical Parameter interfaces usually abstract the details of the actual physical interface to the transducer.

Application-specific functionality is realized in one or more Function Block Objects.

The interfaces to a Block may include the interfaces of IEEE 1451.1 Components or Service Objects Owned by the Block. These Components and Services model commonly used interactions and behaviors. This includes

— Parameter Objects for reading and writing measurement and actuator variables
— Action Objects for physical activities taking place over an extended time period
— File Objects that provide network-accessible data and code storage
— Port Objects for network communication

Copyright © 2000 IEEE. All rights reserved.

A.1.2 Analysis of system requirements

An implementation of an Application System typically consists of the following steps:

— The system requirements are analyzed to determine the various data flows and transformations necessary to meet these requirements.

— The system requirements and their implied data flows and transformations are used to generate a partitioning of the system into functional components and an allocation of these components to hardware and software.

— The hardware functional components are then mapped onto hardware devices resulting in a system hardware design identifying the various sensors, actuators, controllers, and NCAPs of the system.

— An appropriate networking technology is selected.

— Software functional components are mapped onto the object types provided by a vendor's implementation of IEEE 1451.1. This mapping results in an object-based design of the system's software. From this design, code is implemented, if necessary, to augment the object types' provided functionality in order to meet the system's requirements.

It is expected that vendors will provide IEEE 1451.1 system development tools to assist application developers in the design and implementation of IEEE 1451.1 systems.

A.2 Illustration of using the object model

To illustrate the use of the object model, an example of a mapping of the software functionality of a simple wastewater processing system onto IEEE 1451.1 Objects is presented here. The example is for illustrative purposes only.

The purpose of the wastewater processing system is to monitor the pH of industrial wastewater and raise it to a predetermined level by mixing in a sodium hydroxide (NaOH) solution before discharging the water into the city sewer. An abstract layout of the system is given in Figure A.1.

The pH of the water in the tank is controlled by a proportional integral derivative (PID) function that takes as input periodic output of the pH sensor and outputs a control signal to the pump speed control based on the pH measurements and pH setpoints.

The mixer control is signaled to on or to off when the pump speed rises above a given setpoint and drops below another given setpoint, respectively. The water level in the tank is controlled by a PID function that takes as input periodic output of the level sensor and outputs a control signal to the valve control based on a given setpoint (see Bryan and Bryan [B2]).

Both the pH sensor and the level sensor measurement values are supplied to the Operator Station where they are plotted on a display. These measurement values are also logged to a database. Alarm events are signaled if the pH drops below a given alarm setpoint or the water level exceeds a given alarm setpoint. At any time, the system operator can switch the system to manual mode, overriding the automatic control of the pump, the valve, and the mixer.

Figure A.2 shows a hardware mapping for the wastewater treatment system.

As shown in Figure A.2, the major application functionality is partitioned among three NCAPs. The Operator Station is implemented on a PC NCAP as it hosts the system's human-machine interface and a

 Copyright © 2000 IEEE. All rights reserved.

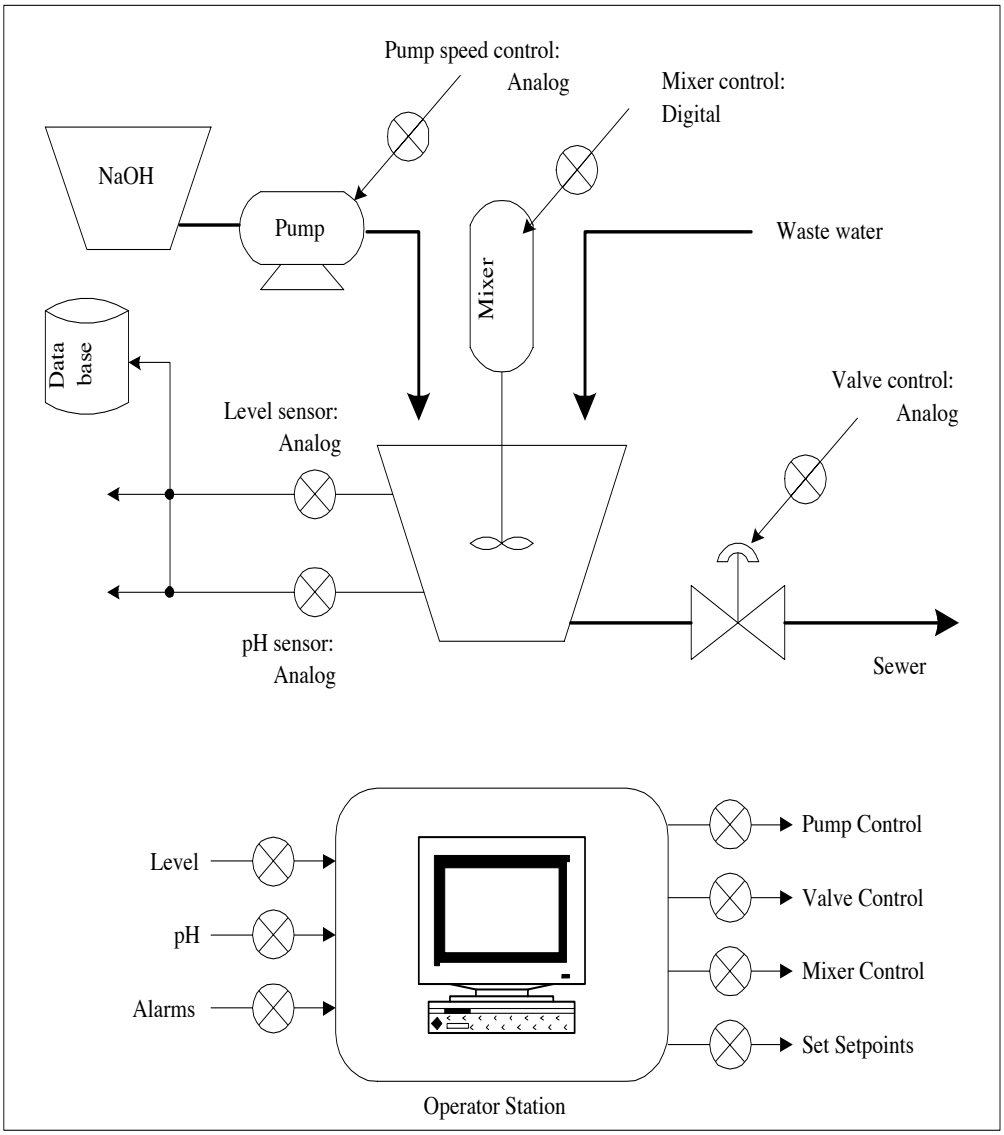

Figure A.1—Wastewater treatment system

database management system. The other system functionality is implemented on simpler NCAPs. The pH control and the level control are implemented on separate NCAPs primarily to satisfy safety requirements.

A.2.1 pH Control

Figure A.3 gives a mapping of the system's software functionality for pH control onto IEEE 1451.1 Objects.

A.2.1.1 pH Dot2 Transducer Block

The three transducers hosted by the pH Control NCAP are all IEEE 1451.2-conformant transducers. Consequently, the pH Dot2 Transducer Block is an instance of the vendor supplied, Dot2 Transducer Block class.

That is, it is not a specialization of that class. In general, an instance of the class Dot2 Transducer Block and its Owned Parameter Objects can provide the software interface for any IEEE 1451.2-conformant transducers without the need for class specialization.

Copyright © 2000 IEEE. All rights reserved.

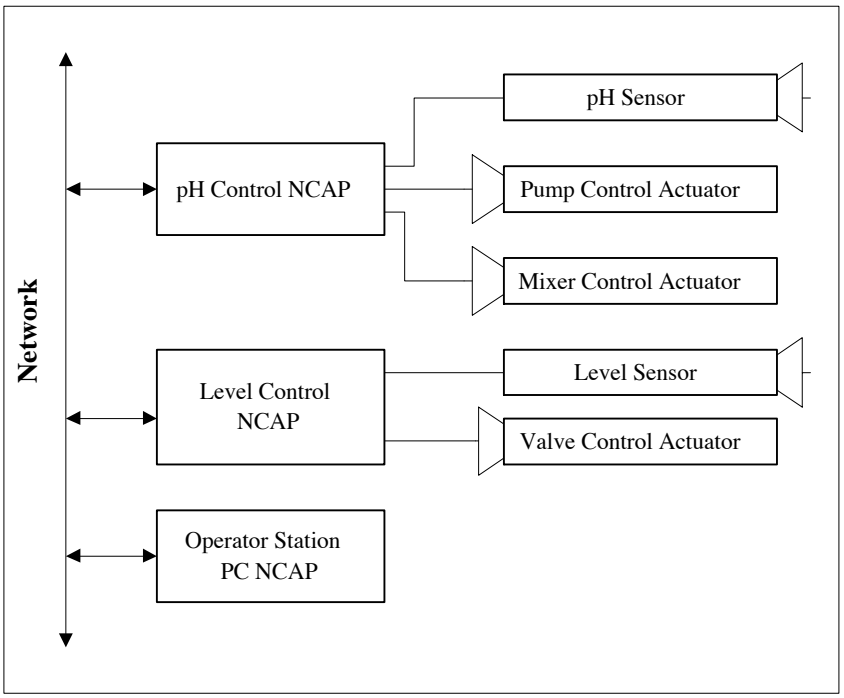

Figure A.2—Hardware components

The pH Dot2 Transducer Block Owns three Parameter Objects

— pH Sensor Parameter

— Pump Speed Control Parameter

— Mixer Control Parameter

These Parameter Objects provide application interfaces to the pH sensor, the pump control, and the mixer control, respectively. These objects are not Network Visible, but they are locally accessible to the pH Function Block.

A.2.1.2 pH Function Block

The pH Function Block is an instance of an application-specific subclass of the class `IEEE1451_FunctionBlock`.

Because there are multiple consumers of the pH measurements, the measurements are published via the pH Measurement Publisher Port Owned by the pH Function Block. The primary reason for having the pH Measurement Publisher Port owned by the pH Function Block rather than the pH Dot2 Transducer Block is so that a "generic," self-configuring Dot2 Transducer Block class could be used as the class of the Transducer Block (see Annex H). Publishing directly from a Dot2 Transducer Block would require altering the interface of the Transducer Block by adding a Publisher Port via class specialization. In addition, if some signal processing of the pH measurement is desired before publication, the pH Function Block is the appropriate place to do this rather than the transducer block. The remainder of the pH Control subsystem's application-specific remote communications are client-server.

The pH PID object provides the PID functionality for controlling the speed of the pump. The object obtains pH measurements from the pH sensor via the pH Sensor Parameter using the Parameter's `UpdateAndRead` operation, and it controls the pump speed and the mixer via the Pump Speed Control

 Copyright © 2000 IEEE. All rights reserved.

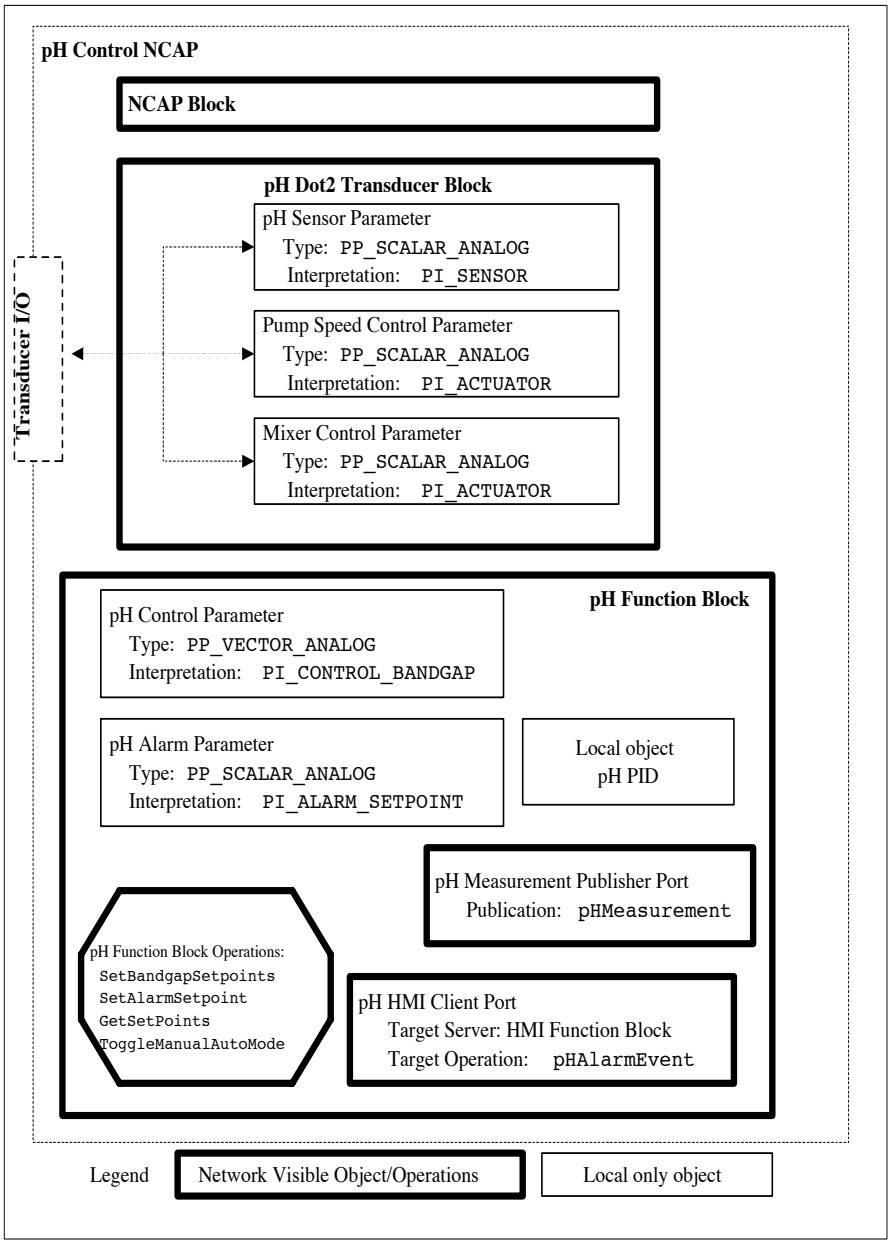

Figure A.3—pH control software functionality to object mapping

Parameter and the Mixer Control Parameter, respectively. It does so by invoking these latter Parameter Objects' `WriteAndUpdate` operations. The pH PID object is not directly exposed to the network, and it need not be an IEEE 1451.1 Object. It is not an IEEE 1451.1 requirement that all objects in an IEEE 1451 system be IEEE 1451.1 Objects. In particular, a Block Object can Own non-IEEE 1451.1 objects.

The pH Function Block Owns two Parameter Objects

— pH Control Parameter (representing the bandgap setpoints for the pH level)

— pH Alarm Parameter (representing the pH alarm setpoint)

In the example system, the pH Function Block provides Network Visible operators to set and get the values of the pH Control Parameter and pH Alarm Parameter. The two Parameter Objects are not Network Visible. An alternative approach would be to make the two Parameters Network Visible. In this case, the pH setpoints

Copyright © 2000 IEEE. All rights reserved.

could be directly accessed by the Parameter operations `Read` and `Write`. Whether an Owned Object is made Network Visible and therefore directly exposes its operations to remote clients, or whether the object is encapsulated by its Owning Block and is accessible only via operations on that Block, is a system design tradeoff.

If the wastewater's pH drops below a given setpoint, the pH PID object of the pH Control subsystem remotely invokes the `pHAlarmEvent` operation of the Operator Station's HMI Function Block Object. Consequently, the pH Function Block Owns a Client Port, HMI Client Port, to support that invocation.

The Operator Station remotely invokes operations in the pH Function Block to set the pH setpoints and to toggle the pump and mixer controls between automatic and manual operation. Accordingly, the pH Function Block has four application-specific Network Visible operations that provide the requisite functionality

— `SetBandgapSetpoints`

— `SetAlarmSetpoint`

— `GetSetpoints`

— `ToggleManualAutoMode`

The Block Major State, `BL_ACTIVE,` of the pH Function Block is substated. The substates are `PH_AUTOMATIC` and `PH_MANUAL`. The operation `ToggleManualAutoMode` causes a transition between these substates. When the Function Block is in the substate `PH_AUTOMATIC`, the pH PID object controls the pump speed and the mixer operation. When it is in the substate `PH_MANUAL`, the system operator controls the pump speed and the mixer operation.

A.2.2 Level control

NCAP2 in the wastewater treatment system hosts the water level control subsystem. A mapping of that subsystem's functionality to IEEE 1451.1 Objects is given in Figure A.4. This mapping is completely analogous to the mapping of the software functionality for pH control.

A.2.3 Operation station

Figure A.5gives a mapping of the system's software functionality for the Operator Station onto IEEE 1451.1 Objects.

The classes of Measurement Log Function Block and HMI Function Block are application-specific specializations of `IEEE1451_FunctionBlock`.

Both the Measurement Log Function Block and HMI Function Block use the pH measurements and the level measurements.

A.2.3.1 Measurement Log Function Block

The Measurement Log Function Block Owns two Subscriber Ports

— Log pH Measurement Subscriber Port for the pH measurement publications

— Log Level Measurement Subscriber Port for the level publications

 Copyright © 2000 IEEE. All rights reserved.

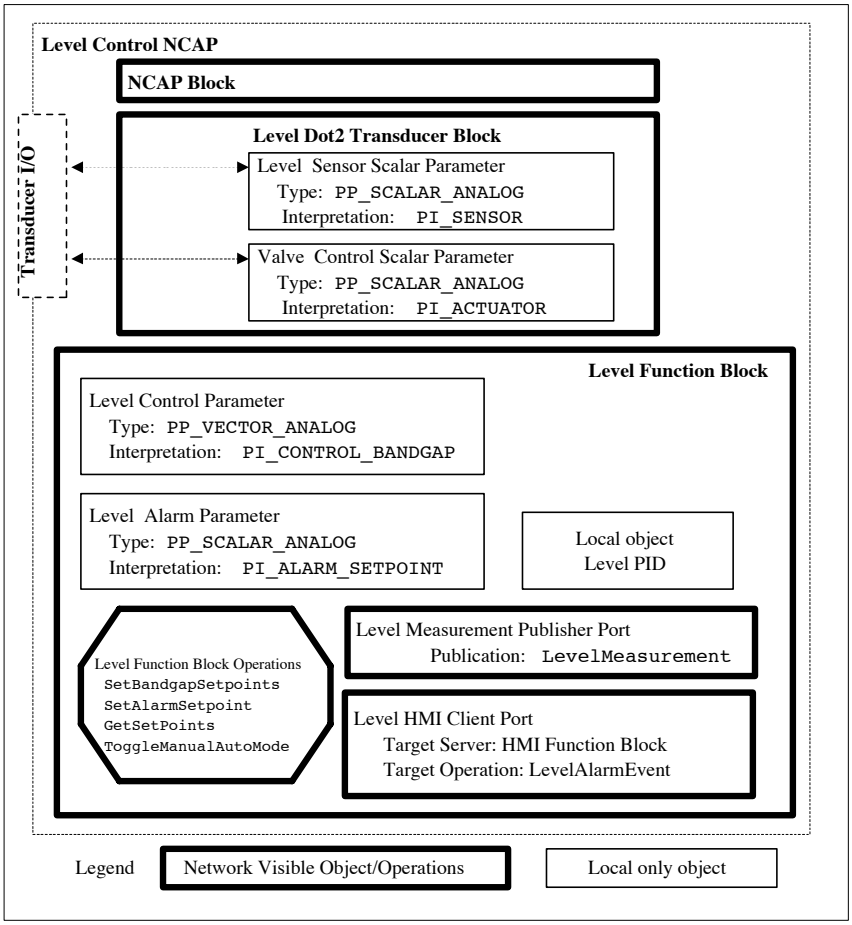

Figure A.4—Level control software functionality to object mapping

A.2.3.2 HMI Function Block

The HMI Function Block Owns two Subscriber Ports

— HMI pH Measurement Subscriber Port

— HMI Level Measurement Subscriber Port

The ports subscribe to the topics `pHMeasurement` and `LevelMeasurement`, respectively. Both the local Log Accessor object and the Human-Machine Interface object are registered Subscribers with the two Subscriber Ports. Whether to have the Measurement Log Function Block only Own two Subscriber Ports that are shared by the Subscribers, or to have the HMI Function Block only Own two Subscriber Ports that are shared by the Subscribers, or to provide the two Subscribers each with their own set of two Subscriber Ports is a design tradeoff. In this example, the latter choice is taken for illustrative purposes only. A more pragmatic choice would be for both the Database Accessor object and the Human-Machine Interface object to be Subscribers to shared Subscriber Ports.

The HMI Function Block has two Network Visible operations to handle pH and wastewater level alarm events

— `pHAlarmEvent` (remotely invoked by the client pH PID object of the pH Function Block)

— `LevelAlarmEvent` (remotely invoked by the client Level PID object of the Level Function Block)

Copyright © 2000 IEEE. All rights reserved.

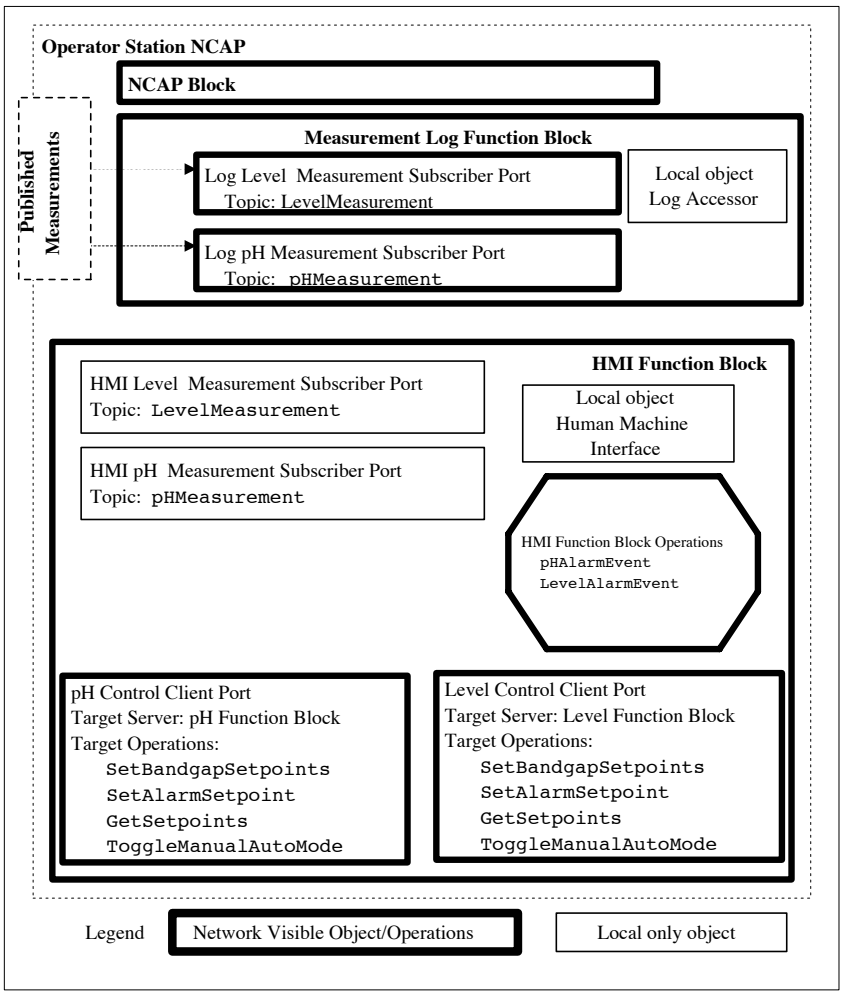

Figure A.5—Operator Station software functionality to object mapping

A PID object does the remote invocation via an `Execute` call to its local HMI Client Port. Here, both HMI Client Ports have their `serverDispatchAddress` members bound to the Object Dispatch Address of the HMI Function Block Object.

The operations for the setting and getting of setpoints and the toggling between automatic and manual control use point-to-point, client-server communication. Accordingly, the HMI Function Block Owns two Client Ports

— pH Control Client Port

— Level Control Client Port

These ports provide remote access to operations on the pH Function Block and the Level Function Block Objects, respectively. The level of granularity of Network Visibility in IEEE 1451.1 is an object rather than an operation. However, a Network Visible Server Object need not make all of its operations accessible to all clients. Such accessibility is controlled by the `Perform` operation of the Server Object and/or the server operations themselves, see 7.1.2.3 and 8.2.3.2, Step 6. Selective exposure of Server Object operations can enhance the robustness of an IEEE 1451.1 system.

NOTE—In this example, the HMI functionality is encapsulated in a Function Block. However, it is not an IEEE 1451.1 requirement that all non-IEEE 1451.1 objects in an IEEE 1451.1 system be contained in IEEE 1451.1 Blocks.

 Copyright © 2000 IEEE. All rights reserved.

The wastewater treatment system and the mapping of its software functionality to IEEE 1451.1 Objects presented in this annex will also be used as an illustrative example in Annex B, Annex C, and Annex D.

Copyright © 2000 IEEE. All rights reserved.

Annex B

(informative)

Client-server example

B.1 Example overview

This annex presents an example client-server interaction. The example is for illustrative purposes only.

NCAP2 in the wastewater treatment system example of Annex A hosts the wastewater Level Control subsystem. A mapping of that subsystem's functionality to IEEE 1451.1 Objects is given in Annex A.

The Level PID object in the Level Function Block of the Level Control subsystem notifies the Operator Station if the level of the wastewater exceeds a given setpoint. It does so by remotely invoking, via a client-server communication, the `LevelAlarmEvent` operation on the Operator Station's HMI Function Block object. The sequence of activities that results in this remote invocation is given below. The steps in the sequence are illustrated with sample pseudo C++ code fragments.

The signature of the `LevelAlarmEvent` operation is

```
OpReturnCode LevelAlarmEvent(
           /*in*/ Float32 level,
           /*out*/ Boolean resume_flag);
```

For the purpose of this example only, it is assumed that the IEEE 1451.1 implementation used provides the following operation on the NCAP Block class. This operation is a part of the client-side network infrastructure for client-server communications.

```
ClientServerReturnCode MarshalAndTransmitClientMessage(
           /*in*/ ObjectDispatchAddress &server_dispatch_address,
           /*in*/ UInteger16 remote_operation_id,
           /*in*/ UInteger16 client_cached_block_cookie,
           /*in*/ ArgumentArray *input_arg_array_p,
           /*out*/ UInteger16 server_associated_block_cookie,
           /*out*/ ArgumentArray *output_arg_array_p);
```

This operation first marshals a client's remote operation invocation request into its on-the-wire format. It then requests the network to deliver the message based on the Object Dispatch Address of the Server Object.

The operation is both network and IEEE 1451.1 implementation specific.

It is also assumed in the example that the IEEE 1451.1 implementation used realizes the IEEE 1451.1 types `ArgumentArray` and `Argument` as object classes. The constructor for `ArgumentArray` takes the size of the Array as a `UInteger16` input parameter. `ArgumentArray` provides an `Encode` operation and a `GetValue` operation.

The signature of the `Encode` operation is

```
OpReturnCode ArgumentArray::Encode(
           Boolean new_members_flag, TypeCode tc_1,...);
```

 Copyright © 2000 IEEE. All rights reserved.

If `new_members_flag` is `TRUE`, then a new Argument will be created for each member in the Array.

If `new_members_flag` is `FALSE`, then it is assumed that the Array consists of the correct number of Argument members, and each member's `TypeCode` and value will be overwritten with the new `TypeCode` and value.

The second argument, `tc_1`, will be used as the `TypeCode` of the 0th Argument in the Array. The third argument is a variable that is to be used as the value of the 0th Argument in the Array. The remaining arguments are alternating pairs {`TypeCode`, value}.

The signature of the `GetValue` operation is

```
OpReturnCode Argument::GetValue(...);
```

The single argument to `GetValue` is a variable whose type has `TypeCode` the discriminant of the Argument. On return, the input variable will be bound to the value of the Argument.

These two operations, `Encode` and `GetValue`, are not type-safe, and they must be used with care.

B.2 Level alarm event remote invocation activity sequence

Step 1: During the creation of the Level Control subsystem, several objects are instantiated that are used during the remote invocation of the `LevelAlarmEvent` operation by the Level PID object.

The initial instantiation of some of the objects within the Level Control NCAP, see Figure A.4, is illustrated by the following pseudo code.

```
// Instantiate Level Dot2 Transducer Block
IEEE1451_Dot2TransducerBlock *level_transducer_block_p =
          new IEEE1451_Dot2TransducerBlock(...);

// Instantiate Level Function Block
IEEE1451_LevelFunctionBlock *level_function_block_p =
          new IEEE1451_LevelFunctionBlock(...);

// Instantiate Level Sensor Scalar Parameter
IEEE1451_ScalarParameter *level_sensor_p =
          new IEEE1451_ScalarParameter(...);

// Instantiate Level Alarm Scalar Parameter
IEEE1451_ScalarParameter *level_alarm_setpoint_p =
          new IEEE1451_ScalarParameter(...);

// Instantiate Level HMI Client Port
IEEE1451_ClientPort *level_hmi_client_port_p =
          new IEEE1451_ClientPort(...);

// Instantiate Level PID passing in to constructor pointers to
// level sensor and level HMI Client Port
LevelPID *level_pid_p =
          new LevelPID(level_sensor_p, level_hmi_client_port_p);
```

Copyright © 2000 IEEE. All rights reserved.

Step 2: Register the Level HMI Client Port with its Local NCAP Block, making it Network Visible. This visibility is necessary because the Level HMI Client Port must be remotely accessible for system configuration.

```
// Register HMI Client Port
level_control_ncap_block.RegisterObject(
           /*in*/ level_hmi_client_port_p,
           /*in*/ level_function_block_p,
           /*out*/ dispatch_address);
```

The `dispatch_address` returned by the `RegisterObject` operation, depending on the implementation, may be needed for implementing the various operations identifying Network Visible Objects. For a Port Object, the dispatch address is usually of interest only to system configuration tools. These tools typically use an operation on the Local NCAP Block, such as the publication `PSK_NETWORK_VISIBLE_SERVER_OBJECT_PROPERTIES,` to gain this information.

Step 3: During the creation of the Operator Station subsystem, the HMI Function Block Object is instantiated and registered with its Local NCAP Block.

```
// Instantiate and register HMI Function Block
HMIFunctionBlock *hmi_function_block_p =
           new HMIFunctionBlock (...);

operator_station_ncap_block.RegisterObject(
           /*in*/ hmi_function_block_p,
           /*in*/ &operator_station_ncap_block,
           /*out*/ dispatch_address);
```

NOTE—The specification of `RegisterObject`, see 9.2.2.1, calls for the two input arguments to be of type `<local object reference>.` From 7.2.2, the form of a `<local Object reference>` is dependent on the language and implementation. For the examples in these annexes, a `<local object reference>` will be specified as either

- A pointer to the object, for example, `hmi_function_block_p`
- A reference to the object, for example, `&operator_station_ncap_block`

Step 4: During system configuration, the `serverDispatchAddress` of the Level HMI Client Port Owned by the Level Function Block is bound to the Object Dispatch Address of the HMI Function Block of the system's Operator Station.

Step 5: During system operation, the Level PID object periodically obtains level measurements from the level sensor by locally invoking the Level Sensor Scalar Parameter Object's `UpdateAndRead` operation. Whenever the Level PID object obtains a level measurement, it compares the measurement's value with the current value of the Level Alarm Parameter. If the measurement value exceeds the Level Alarm Parameter Object's value, an alarm event is raised.

```
// Variables declared during Level PID object initialization
Float32 water_level;
Float32 level_alarm_setpoint;
ArgumentArray *level_arg_array_p = new ArgumentArray(1);
Argument *level_arg_p = new Argument( );
level_arg_array_p[0] = *level_arg_p;

ArgumentArray *level_alarm_setpoint_arg_array_p =
           new ArgumentArray(1);
```

 Copyright © 2000 IEEE. All rights reserved.

```
Argument *level_alarm_setpoint_arg_p = new Argument( );
level_alarm_setpoint_arg_array_p[0] =
           *level_alarm_setpoint_arg_p;

// Read level sensor
op_return_code =
           level_sensor_p->UpdateAndRead(level_arg_array_p);
```

NOTE—The specification of `UpdateAndRead`, see 10.3.1.1, calls for the output argument to be of type `ArgumentArray`. In 4.5, it is noted that the method of passing argument values of operations may be both language- and implementation-dependent. For the `UpdateAndRead` operation in this example, the value is passed by reference.

```
// Check if an error has occurred
if (op_return_code != 0) { ...

// Extract level value from the output Argument Array and
// check for error
op_return_code = level_arg_array[0]->GetValue( water_level);
if (op_return_code != 0) { ...

// Get the current level alarm setpoint value and
// check for errors
op_return_code =
           level_alarm_setpoint_p->
              Read(level_alarm_setpoint_arg_array_p);
if (op_return_code != 0) {...

op_return_code =
           level_alarm_setpoint_arg_array_p[0]->
              GetValue( level_alarm_setpoint);
if (op_return_code != 0) { ...

// Check level against setpoint
if (water_level > level_alarm_setpoint) {...
           // Process alarm condition (see Step-6)

// Carryout PID computations

// Publish Level Measurement
```

Step 6: To raise the level alarm at the Operator Station, the Level PID object invokes the `Execute` operation on the Level HMI Client Port. The call passes in

— The `execute_mode` which, in this example, is bound to `EM_RETURN_VALUE`

— The enumeration member, `LEVEL_ALARM_EVENT,` which is the Operation ID for the `LevelAlarmEvent` operation on the HMI Function Block

— A pointer to the Argument Array `server_input_arguments` containing the encoded input argument to `LevelAlarmEvent`

— A pointer to the `server_output_arguments` Argument Array, which, on return, will contain the `LevelAlarmEvent` output argument

```
// Variables declared during Level PID object initialization
```

Copyright © 2000 IEEE. All rights reserved.

```
ArgumentArray *server_input_arg_array_p =
          new ArgumentArray(1);
Argument *level_arg_p = new Argument( );
server_input_arg_array_p[0] = *level_arg_p;

ArgumentArray *server_output_arg_array_p =
          new ArgumentArray(1);
// Encode the input Argument checking for error
op_return_code = server_input_arg_array_p->
          Encode(FALSE, FLOAT32_TC, water_level);
if (op_return_code != 0) { ...

// Invoke the remote operation via the Level HMI Client Port's
// Execute operation and check for error
client_server_return_code = level_hmi_client_port_p->
          Execute(
             EM_RETURN_VALUE,
             LEVEL_ALARM_EVENT,
             server_input_arg_array_p,
             server_output_arg_array_p);
if (client_server_return_code != 0) {...
```

Step 7: Since the `execute_mode` argument to `Execute` is `EM_RETURN_VALUE`, the client Level PID object blocks awaiting the return of the call. If it was the case that

— It was important that the Level PID object not be blocked by the `Execute` call

OR

— It was not required that the Level PID object be aware if the remote call to `LevelAlarmEvent` was successfully completed

OR

— The `LevelAlarmEvent` operation had no output arguments

then an `execute_mode` of `EM_NO_RETURN_VALUE` could have been used. With this mode, the `Execute` call returns as soon as the Client Port puts the remote call on the wire. If both nonblocking behavior and a value returned to the Client object are required, then an Asynchronous Client Port should be used rather than a Client Port.

Step 8: The Level HMI Client Port uses the network infrastructure operation `MarshalAndTransmitClientMessage`, on its Local NCAP Block to marshal the remote call into its on-the-wire message format and then request that the network transmit the message to the Operator Station subsystem's HMI Function Block Object, using the passed in Object Dispatch Address. Since the `execute_mode` is `EM_RETURN_VALUE`, the `MarshalAndTransmitClientMessage` blocks awaiting a return from the network infrastructure.

```
// Marshal and transmit remote server operation request
client_server_return_code =
          level_control_ncap_block.MarshalAndTransmitClientMessage(
             hmi_function_block_dispatch_address,
             LEVEL_ALARM_EVENT,
             client_cached_block_cookie,
             server_input_arg_array_p,
             server_associated_block_cookie,
             server_output_arg_array_p);
```

 Copyright © 2000 IEEE. All rights reserved.

Step 9: The network infrastructure delivers the message to the Operator Station's NCAP.

Step 10: The server-side network infrastructure

- Demarshals the message's contents.
- Compares the passed in Client Cached Block Cookie with the Associated Block Cookie for the HMI Function Block Object. This latter cookie's value is updated whenever the NCAP Block is requested to `ChangeBlockCookie` for the HMI Function Block.
- If the cookies do not match, the message is rejected and the network infrastructure returns to the Level HMI Client Port's `MarshalAndTransmitClientMessage` call. In this case, the performCode field of the `client_server_return_code` has value `MJ_BLOCK_COOKIE_MISMATCH`.
- If the cookies match, the server-side network infrastructure invokes the `Perform` operation of the HMI Function Block Object passing in the arguments
  - server_operation_id
  - LEVEL_ALARM_EVENT
  - A local pointer, `server_input_arguments_p`, to an Argument Array whose contents is the same as that of `server_input_arguments` passed to the `Execute` operation on the Level HMI Client Port Object
  - A local pointer, `server_output_arguments_p`

```
// Invoke perform on Level HMI Client Port
client_server_return_code = hmi_function_block_p->
           Perform(
              LEVEL_ALARM_EVENT,
              input_arg_array_p,
              output_arg_array_p);
```

The network delivery, demarshaling and `Perform` invocation mechanisms described here are network and IEEE 1451.1 implementation specific.

Step 11: The `Perform` operation of the HMI Function Block Object decodes the input Argument Array back into the input argument for the `LevelAlarmEvent` operation and declares the output argument of `LevelAlarmEvent`. It then invokes `LevelAlarmEvent,` passing in the operation's required input argument and a reference to the operation's output argument. `Perform` blocks awaiting return of `LevelAlarmEvent`.

```
// Variables declared during HMI Function Block Object
// initialization
Float32 water_level;
Boolean resume_flag;

// Decode the input argument and check for errors
op_return_code = input_arg_array_p[0]->
           GetValue(water_level);
if (op_return_code != 0) { ...
```

Copyright © 2000 IEEE. All rights reserved.

```
// Invoke the LevelAlarmEvent operation on the
// HMI Function Block Object
switch(server_operation_id)
{
           ...
           case LEVEL_ALARM_EVENT:
           level_alarm_op_return_code = this->
            LevelAlarmEvent(water_level, &resume_flag);
           break;
           ...
};
```

Step 12: After completing its execution, `LevelAlarmEvent` returns to `Perform` with its output argument bound. `Perform` then

- — Encodes the output argument of `LevelAlarmEvent` into the Argument Array, `server_output_arguments`, passed in to it by the call from the server-side network infrastructure
- — Constructs the client-server return code consisting of the return code of `LevelAlarmEvent` and its own local return code
- — Returns to the server-side network infrastructure

```
// Encode output and check for errors
op_return_code = output_arg_array_p->Encode(TRUE, resume_flag);
if (op_return_code != 0) { ...

// Construct return code and return
client_server_return_code =
           (UInteger32)level_alarm_op_return_code ^
              (perform_op_return_code << 16);
return client_server_return_code;
```

Step 13: When `Perform` returns to the server-side network infrastructure, the infrastructure

- — Marshals the output Argument Array and the `client_server_return_code` of `Perform` into an on-the-wire message
- — Transmits the message back to the Level Control NCAP

Step 14: The client-side network infrastructure

- — Demarshals the returned output argument into the `server_output_arguments` Argument Array. This Array was passed to the client-side network infrastructure in the `MarshalAndTransmitClientMessage` operation invoked on the Local NCAP Block by the Level Alarm Client Port
- — Demarshals the client-server return value returned from `Perform` and returns it to the Level HMI Client Port as the return from its call to the `MarshalAndTransmitClientMessage` operation on the Local NCAP Block

Step 15: On the return from the client-side network infrastructure to the `Execute` call on the Level HMI Client Port, `Execute` augments the call's client-server return code with the major return code of its own operation. The `Execute` call then returns to its invoker, the Level PID object.

 Copyright © 2000 IEEE. All rights reserved.

```
// Construct return code and return to Level PID Client object
client_server_return_code =
           (execute_major_return_code << 24) ^
              client_server_return_code;
return client_server_return_code;
```

Step 16: The Level PID object checks the return value of its call to `Execute`. If there are no errors, it decodes the `server_output_arguments` Argument Array into the output argument of `LevelAlarmEvent`, `resume_flag`, completing the client-server call from the Level PID object to the `LevelAlarmEvent` operation of the Operator Station's HMI Function Block Object.

```
// Check if a remote invocation error has occurred
if (client_server_return_code != 0) {...
// Decode output argument
op_return_code = server_output_arguments[0]->
           GetValue(resume_flag);
// Check if an error has occurred
if (op_return_code != 0) { ...
```

Step 17: The client Level PID object continues execution or pauses according to the value of the output `resume_flag.`

Step 18: During any of the activities in Step 6 through Step 13, see 8.2.3.2 and 11.4.2.5, the `Execute` operation on the Level HMI Client Port could time out, provided the Port's timeout value is greater than 0. In this case, the `portCode` field of the client-server return code has value `MJ_OPERATION_TIMEOUT`.

Some of the steps in a client-server remote operation invocation have a form that is independent of which client, which server, and which server operation are involved. An implementation of IEEE 1451.1 could have library routines that perform these boilerplate steps. A development tool for IEEE 1451.1 systems could also provide a wizard to help with code for client-server interactions.

Copyright © 2000 IEEE. All rights reserved.

Annex C

(informative)

Publish-subscribe example

C.1 Example overview

This annex presents an example publish-subscribe interaction. The example is for illustrative purposes only.

The pH Control subsystem of the wastewater treatment system presented in Annex A controls the pH level of the wastewater. A mapping of the subsystem's functionality to IEEE 1451.1 Objects is given in Annex A.

The pH PID Object in the pH Function Block Object of the pH Control subsystem periodically obtains wastewater pH measurements by invoking the `UpdateAndRead` operation on the pH Sensor Parameter Object. The pH Sensor Parameter Object is Owned by the pH Dot2 Transducer Block Object. The Transducer Block provides the logical interface between the physical pH sensor and the pH Sensor Parameter Object.

There are three consumers of the pH measurements.

— The measurements are used by the pH PID Object in the closed-loop control of the NaOH solution pump.

— The measurements are displayed in real time on the Operator Station's display.

— The measurements are logged in the wastewater system's long-term database hosted by the Measurement Log Function Block Object Operator Station subsystem.

The pH PID object publishes each pH measurement it obtains. The object uses the pH Measurement Publisher Port Object Owned by the pH Function Block Object for the publishing.

There are two Subscribers to the pH measurement publications in the Operator Station subsystem

— The Database Accessor object, Owned by the Measurement Log Function Block Object

— The Human-Machine Interface object, Owned by the HMI Function Block Object

Each Subscriber has a callback operation named `pHMeasurementPublication`, that is called by the Log pH Measurement Subscriber Port, and HMI pH Measurement Subscriber Port `Objects`, respectively, whenever the Port receives a `pHMeasurement` publication. As specified by the IEEE 1451.1 standard, the signature of each `pHMeasurementPublication` operation is

```
OpReturnCode pHMeasurementPublication(
           /*in*/ UInteger16 subscription_id,
           /*in*/ UInteger8 publication_port_publication_key,
           /*in*/ PublicationTopic
              &publishing_port_publication_topic,
           /*in*/ ArgumentArray &publication_contents);
```

The sequence of activities that results in a publication of a pH measurement and receipt of that measurement publication by the Database Accessor and Human Machine Interface Objects is described below. The steps in the sequence are illustrated with sample pseudo C++ code fragments.

 Copyright © 2000 IEEE. All rights reserved.

For the purpose of this example only, it is assumed that the IEEE 1451.1 implementation used provides the following publisher-side network infrastructure operations on the NCAP Block class:

```
OpReturnCode MarshalAndPublishPublication(
           /*in*/ PubSubDomain &domain,
           /*in*/ UInteger8 pub_sub_key,
           /*in*/ PublicationTopic &topic,
           /*in*/ ArgumentArray &publication_contents);
```

and the following operation on the Subscriber Port class:

```
void FilterPublicationAndCallbackSubscribers(
           /*in*/ PubSubDomain &domain,
           /*in*/ UInteger8 pub_sub_key,
           /*in*/ PublicationTopic &topic,
           /*in*/ ArgumentArray &publication_contents);
```

The first operation is invoked by a Publisher Port when it wants to publish a publication onto the network. The second operation is invoked by the subscriber-side network infrastructure on all Subscriber Ports in the system whenever a publication message is delivered to these NCAPs.

These two operations are both network and IEEE 1451.1 implementation specific.

It is also assumed in the publish-subscribe example that, as in Annex B, `ArgumentArray` and `Argument` are implemented as object classes with the operations `Encode` and `GetValue`, respectively.

C.2 pH measurement publish-subscribe sequence

The example publish-subscribe sequence is simpler than the client-server sequence described in Annex B. This is because a publisher does not need to know its subscribers while a client does need to know its server.

Step 1: During the creation of the pH Control subsystem, several objects are instantiated that are used during the publication of a pH measurement by the pH PID object.

```
// Instantiate objects
IEEE1451_Dot2TransducerBlock *ph_transducer_block_p =
           new IEEE1451_Dot2TransducerBlock(...);
pHFunctionBlock *ph_function_block_p = new pHFunctionBlock(...);
IEEE1451_ScalarParameter *ph_sensor_p =
           new IEEE1451_ScalarParameter(...);
IEEE1451_PublisherPort *ph_publisher_port_p =
           new IEEE1451_PublisherPort(
           ...,"pHMeasurementPublisherPort", ...);
              // The name of the Publisher Port,
              // pHMeasurementPublisherPort, is passed
              // to the constructor
pHPID *ph_pid_p = new pHPID(
           ph_sensor_p,
           ph_publisher_port_p,
           ph_hmi_client_port_p);
              // Pointers to needed local objects are passed
              // to the constructor
```

Copyright © 2000 IEEE. All rights reserved.

Step 2: The pH Measurement Publisher Port Object is registered with its Local NCAP Block, making it Network Visible. This visibility is necessary because the pH Measurement Publisher Port Object must be remotely accessible for system configuration.

```
// Register Publisher Port
ph_control_ncap_block.RegisterObject(
           /*in*/ ph_publisher_port_p,
           /*in*/ ph_function_block_p,
           /*out*/ dispatch_address);
```

Step 3: During the creation of the Operator Station subsystem, the Log pH Measurement Subscriber Port and HMI pH Measurement Subscriber Port Objects are instantiated.

Step 4: During the instantiation of the Database Accessor Object and Human-Machine Interface Object, a pointer to the Log pH Measurement Subscriber Port Object and HMI pH Measurement Subscriber Port Object, respectively, is passed as a constructor argument. This is so the Database Accessor and Human Machine Interface Objects can subscribe to receive pH measurement publications via the respective Subscriber Ports. The two Subscriber Ports are registered with the Local NCAP Block so that they are Network Visible for system configuration purposes.

```
//Instantiate and register Subscriber Ports
IEEE1451_SubscriberPort *log_ph_subscriber_port_p =
           new IEEE1451_SubscriberPort(Ö);
IEEE1451_SubscriberPort *hmi_ph_subscriber_port_p =
           new IEEE1451_SubscriberPort(Ö);
operator_station_ncap_block.RegisterObject(
           /*in*/  log_ph_subscriber_port_p,
           /*in*/  measurement_log_function_block_p,
           /*out*/ dispatch_address1);
operator_station_ncap_block.RegisterObject(
           /*in*/  hmi_ph_subscriber_port_p,
           /*in*/  hmi_function_block_p,
           /*out*/ dispatch_address2);
// Instantiate database accessor
DatabaseAccessor *database_accessor_p =
           new DatabaseAccessor (...,
             log_ph_subscriber_port_p, ...);
// Instantiate HMI Function Block
HumanMachineInterface *human_machine_interface_p =
           new HumanMachineInterface (...,
              hmi_ph_subscriber_port_p, ...);
              // Pass in pointer to Subscriber Port
              // constructor so block can register
              // interest in publications
```

When the pH Measurement Publisher Port Object is initialized, its Publication Domain member is bound to an 8-element Octet Array reference, with only the low-order bit of the 0th octet turned on (i.e., its Publication Domain is 1). As specified in IEEE 1451.1, its Publication Key member is set to `PSK_PHYSICAL_PARAMETRIC_DATA`. The Subscription Domain and Subscription Key members of both the Log pH Measurement Subscriber Port and the HMI pH Measurement Subscriber Port Objects are bound to the same values, respectively.

 Copyright © 2000 IEEE. All rights reserved.

C.3 System configuration

During system configuration, the Publication Topic member of the pH Measurement Publisher Port Object is bound to the Octet Array that, when interpreted as an 8-bit ASCII String, is `pHMeasurement`.

The Subscription Qualifier member of both the Log pH Measurement Subscriber Port and the HMI pH Measurement Subscriber Port Objects are bound to the same value.

C.4 System initialization

During system initialization, both the Database Accessor Object and the Human Machine Interface Object of the Measurement Log Function Block Object and HMI Function Block Object, respectively, subscribe to the `pHMeasurement` publication by notifying the Log pH Measurement Subscriber Port Object and HMI pH Measurement Subscriber Port Object, respectively, of their interest in such publications. They do this by a call to the Ports' `AddSubscriber` operations. The `AddSubscriber` operation on a Subscriber Port takes as one of its arguments a reference to a callback operation on the Subscriber, in this case the `pHMeasurementPublication` operation, that is invoked by the Port whenever a publication with topic `pHMeasurement` and Domain and key matching those of the Port is received by the Port from the subscriber-side network infrastructure.

```
// Subscribe to publications
OpReturnCode = log_ph_subscriber_port_p->
           AddSubscriber(
              database_accessor_p,
              &DatabaseAccessor::pHMeasurementPublication,
              117);// The subscription id is 117
OpReturnCode =  hmi_ph_subscriber_port_p->
            AddSubscriber(
              human_machine_interface_p,
              &HumanMachineInterface::pHMeasurementPublication,
              223);// The subscription id is 223
```

C.5 System operation

During system operation, the pH PID object periodically obtains pH measurements from the pH sensor by locally invoking the pH Sensor Parameter's `UpdateAndRead` operation. Whenever the pH PID object obtains a pH measurement, it first uses the measurement to update the PID loop. It then encodes the measurement into the IEEE 1451.1 required Publication Contents for physical parametric data, and it requests the pH Measurement Publisher Port Object to publish the pH measurement publication.

```
// Variables declared during pH PID object initialization
// Variables used in making the measurement
ArgumentArray *ph_measurement_arg_array_p = new ArgumentArray(2);
Argument *ph_arg_p = new Argument( );
ph_measurement_arg_array_p[0] = *ph_arg_p;
// Variables used in constructing the publication
ArgumentArray *ph_publication_arg_array_p =
           new ArgumentArray(2); // publication_contents
Argument *ph_publication_content_code_arg_p =
           new Argument( );  // content_code
```

Copyright © 2000 IEEE. All rights reserved.

```
Argument ph_publication_arg_array_p [0] =
          *ph_publication_content_code_arg_p;
Argument *ph_physical_parameter_data_arg_p =
          new Argument( );  // physical_parameter_data
Argument ph_publication_arg_array_p [1] =
          *ph_physical_parameter_data_arg_p;
UInteger8 publication_content_code = PCC_NORMAL_DATA;

// During system operation
// Get a new pH measurement
op_return_code = ph_sensor_p->
          UpdateAndRead(ph_measurement_arg_array_p);
if (op_return_code != 0) { ...

// Extract pH measurement from the output Argument Array
// and check for errors
op_return_code = ph_measurement_arg_array_p[0]->
          GetValue(ph_physical_parameter_data_arg_p);
if (op_return_code != 0) { ...

// Encode the publication and check for errors
op_return_code = ph_publication_arg_array_p ->
          Encode(
             FALSE,
             UINTEGER8_TC, & publication_content_code,
             PHYSICAL_PARAMETER_DATA_TC,
                      ph_physical_parameter_data_arg_p);
if (op_return_code != 0) { ...

// pH PID object requests port to publish the publication
op_return_code = ph_publisher_port_p->
          Publish(ph_publication_arg_array_p);
```

NOTE—That in the above code fragment, the pH PID object could publish the pH measurement's metadata along with the measurement, either always or just occasionally. In this case, the `publication_content_code` would be set to `PCC_BOTH_META_AND_NORMAL_DATA`, the publication Argument Array would have been declared with length 3 rather than 2, and the value of the value of `ph_publication_arg_array_p[2]` would be of type `PhysicalParameterMetadata`.

Upon receipt of the request to publish the pH measurement, the pH Measurement Publisher Port Object invokes the publisher-side network infrastructure operation, `MarshalAndPublishPublication`, on its Local NCAP Block, passing in the appropriate arguments.

```
// Invoke publisher-side network infrastructure
// and check for errors
OpReturnCode =
          ph_control_ncap_block.MarshalAndPublishPublication(
             publication_topic,
             publication_key,
             publication_domain,
                      ph_publication_array_p);
if (op_return_code != 0) { ...
```

The network infrastructure delivers the publication to the Log pH Measurement Subscriber Port and HMI pH Measurement Subscriber Port Objects of the Operator Station subsystem. It does this via calls to the

 Copyright © 2000 IEEE. All rights reserved.

`FilterPublicationAndCallbackSubscribers` operation on each Subscriber Port. The operation first checks if the received publication passes the Port's filter tests, that is, its Domain, key, and topic match those of the Port. If it does, the Port invokes the subscription callback operation on each Subscriber previously registered with it via the Port's `AddSubscriber` operation. In particular, the Log pH Measurement Subscriber Port invokes the Database Accessor Object's `pHMeasurementPublication` callback operation:

```
// Filter publication and callback subscribers operation
// invoked by the subscriber-side network infrastructure
log_ph_subscriber_port_p->
          FilterPublicationAndCallbackSubscribers(...);
// Within the FilterPublicationAndCallbackSubscribers operation
// the callback is made to the database accessor object
database_accessor_p-> pHMeasurementPublication(...
```

The HMI pH Measurement Subscriber Port invokes the Human Machine Interface Object's `pHMeasurementPublication` callback operation:

```
// Filter publication and callback subscribers operation
// invoked by the subscriber-side network infrastructure
hmi_ph_subscriber_port_p->
          FilterPublicationAndCallbackSubscribers(...);
// Within the FilterPublicationAndCallbackSubscribers operation
// the callback is made to the HMI object
human_machine_interface_p-> pHMeasurementPublication(...
```

Within the `pHMeasurementPublication` operation, the passed in `pHMeasurement` publication is decoded, and the publish-subscribe communication has been completed.

Some of the steps in a publish-subscribe communication have a form that is independent of the particular publisher, subscriber, or publication. An implementation of IEEE 1451.1 could have library routines that perform these boilerplate steps. A development tool for IEEE 1451.1 systems could also provide a wizard to help with code for publish-subscribe interactions.

Copyright © 2000 IEEE. All rights reserved.

Annex D

(informative)

System configuration and operation examples

This annex presents examples of the boot-up and configuration of IEEE 1451.1 systems. The examples are for illustrative purposes only. Throughout this annex, the term *node* is used to denote an NCAP and its associated software and transducers.

The IEEE 1451.1 standard is designed for use with a broad spectrum of measurement and control systems ranging from simple systems with two or three NCAPs to complex systems with hundreds, or even thousands, of NCAPs. Accordingly, the standard provides a variety of mechanisms to aid in the commissioning, startup, modification, and maintenance of IEEE 1451.1 smart transducer systems. The mechanisms are listed in Table D.1.

It is expected that vendors will provide a variety of tools for the design and development of software for IEEE 1451.1 systems, as well as tools for system boot-up, commissioning, startup, and maintenance based on these mechanisms.

Most IEEE 1451.1 smart transducer systems will make use of only a small subset of the mechanisms listed in Table D.1.

In this informative annex, two examples of using the mechanisms for IEEE 1451.1 system software installation, boot-up and configuration are presented

— The first example is for the wastewater treatment system described in Annex A. This example is somewhat detailed.

— The second example is a description of boot-up and configuration techniques applicable to large, dynamic IEEE 1451.1 systems.

Table D.1—Commissioning, Startup, Modification, and Maintenance Mechanisms

Activity	System lifecycle phase	Mechanisms

Node discovery	Commission	**Publications with Publication Key values of:**
		`PSK_NCAPBLOCK_ANNOUNCEMENT`
		`PSK_REQUEST_NCAPBLOCK_ANNOUNCEMENT`
		`PSK_FORCE_NCAPBLOCK_ANNOUNCEMENT`
		NCAP Block operations:
		`IgnoreRequestNCAPBlockAnnouncement`
		`RespondToRequestNCAPBlockAnnouncement`

Object initialization		**Block operations:** `Initialize`

 Copyright © 2000 IEEE. All rights reserved.

Table D.1—Commissioning, Startup, Modification, and Maintenance Mechanisms (continued)

Activity	System lifecycle phase	Mechanisms
Object identification	Commission	**Object attributes accessed by the following operations:** GetClassName GetClassID GetObjectTag GetOwningBlockObjectTag GetObjectName GetObjectID GetObjectProperties (accesses the Associated Block Cookie in addition to other attributes accessible via specific Get operations). GetDispatchAddress **Block attributes accessed by the following operations:** GetGroupIDs GetBlockManufacturerID GetBlockModelNumber GetBlockVersion **NCAP Block attributes accessed by the following operations:** GetNCAPManufacturerID GetNCAPModelNumber GetNCAPSerialNumber GetNCAPOSVersion Component Group class objects
Dynamic code download	Commission	File Objects
Configure communication paths	Commission	**Publications with Publication Key values of:** PSK_NETWORK_VISIBLE_SERVER_OBJECT_PROPERTIES PSK_REQUEST_NETWORK_VISIBLE_SERVER_OBJECT_PROPERTIES **Block operations:** GetNetworkVisibleServerObjectProperties **NCAP Block operations:** SetConfigurationToolAddress GetNewObjectID (Local) RegisterObject (Local) DeregisterObject (Local) GetClientPortProperties SetClientPortServerObjectBindings **Publisher Port attributes accessed by the following operations:** GetPublicationKey GetPublicationTopic SetPublicationTopic GetPublicationDomain SetPublicationDomain Publish (Local)

Copyright © 2000 IEEE. All rights reserved.

Table D.1—Commissioning, Startup, Modification, and Maintenance Mechanisms (continued)

Activity	System lifecycle phase	Mechanisms
Configure communication paths (Continued)	Commission (Continued)	**Subscriber Port attributes accessed by the following operations:** GetSubscriptionKey GetSubscriptionQualifier SetSubscriptionQualifier GetSubscriptionDomain SetSubscriptionDomain AddSubscriber (Local) DeleteSubscriber (Local)
Transducer identification and configuration	Commission and Operations	**Transducer Block operations:** EnableCorrections DisableCorrections GetNumberOfTransducerChannels GetMinimumSamplingPeriod GetChannelParameterObjectTags GetParameterObjectChannelNumbers GetUnrepresentedChannelNumbers **Dot2 Transducer Block operations:** EnableDataSequenceSensors DisableDataSequenceSensors EnableEventSequenceSensors DisableEventSequenceSensors GetStatusInterruptMask SetStatusInterruptMask GetSTIMUserString SetSTIMUserString RegisterInterruptNotification (Local) DeregisterInterruptNotification (Local) **Classes used with Transducer Blocks:** Parameter classes Component Group class Files (for Dot2 meta-TEDS) Partitioned Files (for Dot2 channel specific TEDS)
System startup	Commission	**Block operations:** GoActive **Publications with Publication Key values of:** PSK_NCAPBLOCK_GO_ACTIVE **NCAP Component Group Objects with Object Names:** Permanent_Startup_Set Configurable_Startup_Set

Copyright © 2000 IEEE. All rights reserved.

Table D.1—Commissioning, Startup, Modification, and Maintenance Mechanisms (continued)

Activity	System lifecycle phase	Mechanisms
Dynamic system modification	Modification and Maintenance	**Block attributes (see Commissioning section above)** **Block operations:** `RegisterNotifyOnUpdate` (Local) `DeregisterNotifyOnUpdate` (Local) **NCAP Block operations and mechanisms:** `GetBlockCookie` (Local) `ChangeBlockCookie` (Local) Block Cookie checking mechanism for client-server communications Self Identifying Publisher Port class Publication Change ID checking mechanism for publish-subscribe communications File Objects

Transducer hot swap	Modification and Maintenance	**Block mechanisms:**
		Block Cookie mechanism
		Dot2 Transducer Block operations:
		`RegisterInterruptNotification`
		`GetSTIMUserString`
		Dot2 TEDS Files

System shutdown	Operations	**Block operations:**
		`GoInactive`
		`Reset`

D.1 Example 1: The wastewater treatment system

The wastewater treatment system is a simple system that is static with respect to modifications. It is assumed that the system is taken offline when repairs or modifications are required. In particular

— The system's software is not changed during system operations.

— The software need not support dynamic reloading.

— Dynamic configuration of the system's inter-NCAP client-server and publish-subscribe message links is not required.

The application software is developed offline in a vendor-supplied IEEE 1451.1 software development environment. The configuration of the system's inter-NCAP communication links is done as a part of the software development process. The links are hard wired. Modification of these links requires changes in the application source code.

D.1.1 Object Dispatch Addresses

An Object's `DispatchAddress` is dependent on both the network used and the IEEE 1451.1 implementation for that network. For example, one simple way of implementing IEEE 1451.1 network client-server communications is to have the receiving endpoints for all client-server messages be a fixed,

Copyright © 2000 IEEE. All rights reserved.

distinguished operation on each node's Local NCAP Block Object. In the case, for example, that the network is Ethernet, then an Object's Object Dispatch Address could be a triple consisting of

— The IP address of the NCAP on which the Object resides

— A fixed, standard port number on the NCAP

— The Object's Object Tag

The Object Dispatch Address scheme used in the wastewater treatment system is the one just described.

A very simple, but inelegant, example way of implementing IEEE 1451.1 publish-subscribe communications is to have the receiving endpoints for all publication messages be another fixed, distinguished operation on each NCAP Block Object. The NCAP Block Object keeps a list of references to all its Owned Subscriber Port Objects, and when it receives a publication message, it forwards the demarshaled, but still encoded, publication to all of its Subscriber Ports.

The publish-subscribe communication mechanism used in the wastewater treatment system is the one just described.

D.1.2 Development time configuration of system bindings

In the wastewater system, much of the system configuration is set at development time. This is done, in part, by binding many of each Network Visible object's object properties directly in the source code. Specifically, the following Network Visible object properties are bound to source code-supplied constant values:

— Object Tag

— Object Name

— Owning Block's Object Tag

— Object Dispatch Address

For example, in C++, these values could be passed as arguments to the constructor method on the object's class.

In addition to the above bindings, the following Objects will have the member values shown bound to constant values during the initialization of each Object:

— Client Port Object member values bound
  — Server Object Tag
  — Server Object Dispatch Address
— Publisher Port Object member values bound
  — Publication Topic
  — Publication Key
  — Publication Domain
— Subscriber Port Object member values bound
  — Subscription Qualifier
  — Subscription Key
  — Subscription Domain

 Copyright © 2000 IEEE. All rights reserved.

For Publishers and Subscribers, during software initialization

— Each Publisher object is provided with a local reference to the Publisher Port it uses

— Each Subscriber object is provided with a reference to the Subscriber Port it uses

— Each Subscriber registers a callback operation with the local Subscriber Ports of interest

D.1.3 Application executable programs

The run-time application software for each NCAP in the wastewater treatment system is a single, NCAP-specific executable program. The executable contains one main program implementing two system startup steps.

Step 1: If necessary, download each NCAP's executable program. (For example, load it onto the NCAP's hard disk or load it into the NCAP's program ROM).

Step 2: Invoke each node's executable program.

NOTE—In the simple wastewater system, these two steps are manual operator operations.

In the implementation of the wastewater system, each Block Object has an operation that instantiates all of the objects it Directly Owns. This means

— The objects it Owns that are not Owned by any other Block Object.

— Each Block Object is responsible for registering with the Local NCAP Block Object all objects it instantiates that are to be Network Visible.

In each of the system nodes, objects in one Block Object directly access objects in another Block Object. For example, the pH PID object in the pH Function Block invokes the `UpdateAndRead` operation on the pH Sensor Parameter Object in the pH Dot2 Transducer Block. Accordingly, as each object is instantiated, the reference to the object is provided to the main program that binds it to a variable global to the Block Objects.

The main program's actions include

— Instantiate and initialize the Local NCAP Block Object, placing it into the `BL_UNINITIALIZED` state. This action also results in the recursive instantiation of all objects in the main program's space. Each Block Object is put into the `BL_UNINITIALIZED` state.

— The binding of the object reference global variables described above. This results in the resolution of all local object references. In particular, each Client object has a reference to its associated Client Port, each Publisher object has a reference to its associated Publisher Port and each Subscriber object has a reference to its associated Subscriber Port.

— Set the state of the NCAP Block Object and each of its Owned Block Objects to `BL_INACTIVE.`

When the instantiation and the initialization of a node's application objects are completed, the main program turns control over to the NCAP Block Object. At this point, the node is operational, but inactive.

D.1.4 System startup

Included with the vendor supplied software development environment is a simple, manual, system startup tool. This tool consists of an executable program with an HMI. In the wastewater treatment system, the tool

Copyright © 2000 IEEE. All rights reserved.

runs in its own process space on the Operator Workstation PC NCAP. The tool itself is built with the vendor's IEEE 1451.1 object classes.

The startup tool allows the system operator to do simple NCAP Block Object discovery. It does so by providing a control on the tool's HMI, by which the operator can initiate the publishing of the distinguished, IEEE 1451.1-specified publication with Publication Key `PSK_REQUEST_NCAPBLOCK_ANNOUNCEMENT`. The tool uses a Subscriber Port, to listen for the NCAP Blocks' responding publications with Publication Key `PSK_NCAPBLOCK_ANNOUNCEMENT`. As it receives each such publication, it displays the decoded publication's contents on its HMI.

To perform NCAP Block discovery, the operator has the startup tool publish the `Request_NCAPBlock_Announcement` and listen for the responses. The operator repeatedly has the tool publish the `Request_NCAPBlock_Announcement` until it is highly probable that all active NCAPs have responded. At this point, the operator decides if there are sufficient resources for the overall system to perform its tasks. In the case of the wastewater treatment system, all NCAP Blocks must be present for the system to be operational.

When the operator has verified via NCAP Block discovery that the three NCAP Blocks in the system are operational, the operator then has the startup tool publish the distinguished, IEEE 1451.1-specified publication with Publication Key `PSK_NCAPBLOCK_GO_ACTIVE.` This places the three NCAP Block Objects into their `PH_AUTOMATIC` substate.

There is a start sequence dependency between the pH Control subsystem and the Level Control subsystem. The Level Control subsystem should not release any wastewater into the public sewer before the pH Control subsystem has verified that the pH level is acceptable. An `IEEE1451_ConditionVariableService` Object is used by the application to assure that the pH Control subsystem has successfully performed any initial pH modifications before the Level Control subsystem opens the release valve.

D.1.5 System shutdown

The operator can shut the system down via the publication of an application-specific `PH_SYSTEM_SHUTDOWN` publication that is subscribed to by the system's NCAP Block Objects. Upon receipt of this publication, an NCAP Block Object, as well as all of its Owned Block Objects, transitions into its `BL_INACTIVE` state. As with system startup, there is also a sequential dependency in the wastewater treatment system shutdown. This dependency is also handled by the use of a Condition Variable Service Object.

D.1.6 Observation

In the simple waster water treatment system example, only two publications out of the extensive list of commissioning, startup, modification, and maintenance mechanisms provided by IEEE 1451.1 are used. This is because the system is a static system that is never modified in any way during operations.

D.2 Example 2: Dynamic system boot-up and commissioning

The simple static boot-up and configuration mechanism described above is not adequate for a large and/or dynamically changing IEEE 1451.1 systems. Examples of such dynamic systems are an environment control system in a large, multiuse building; the control system for a petroleum refinery; and a factory automation system. Such systems often require the ability to replace nodes and/or transducers during operation (hot swapping), add new nodes, and dynamically modify node software.

 Copyright © 2000 IEEE. All rights reserved.

Example 2 presents an abstract description of example techniques for dynamic system boot-up and commissioning. The example assumes the use of a vendor-supplied central configuration tool that executes on an NCAP on the network. When the terms configuration tool or tool are used in this annex, they refer to the configuration tool and, if needed, a human user of the tool.

D.2.1 Initial system boot-up

When each NCAP in the system is first powered on, the initial boot-up mechanism instantiates an NCAP Block Object and its IEEE 1451.1-required Ports and sets the NCAP Block Object's state to `BL_UNINITIALIZED`. It next partially initializes the NCAP Block Object. In particular, it binds the following attributes of the Object:

— Class ID

— Class Name

— Object ID

— Object Name

— Block Cookie

— Object Dispatch Address

After initial boot-up, the NCAP Block Object has all of its IEEE 1451.1-required Publisher and Subscriber Ports, and supports all of the IEEE 1451.1-required Network Visible and Local operations. The NCAP Block Object is placed in its `BL_INACTIVE` state.

The last action of the initial boot-up sequence for each NCAP is to have the NCAP Block Object publish the publication with key `PSK_NCAPBLOCK_ANNOUNCEMENT`.

The configuration tool subscribes to this publication. In a large IEEE 1451.1 system, initial boot-up will result in a storm of Announcement publications, and there is no assurance that all of the NCAP Block Object Announcement publications will be received by the tool. Accordingly, the configuration tool can remotely invoke the following operations on each NCAP Block Object it has discovered:

— `IgnoreRequestNCAPBlockAnnouncement`

— `RespondToRequestNCAPBlockAnnouncement`

It can also publish on the Publication Keys

— `PSK_REQUEST_NCAPBLOCK_ANNOUNCEMENT`

— `PSK_FORCE_NCAPBLOCK_ANNOUNCEMENT`

Using the first operation to contain Announcement publication message storms, the configuration tool can repeatedly request Announcement publications until there is a high probability that all available NCAPs have been discovered. The `RespondToRequestNCAPBlockAnnouncement` operation and the `Force_NCAPBlock_Announcement` publication can be used to validate the availability of previously discovered and quieted NCAP Block Objects.

At the termination of the node discover activity, the configuration tool assesses if the identified nodes can form a viable, operational system. To make this assessment, the tool uses information in a system configuration file. This configuration file is constructed by using a GUI in the vendor-supplied IEEE 1451.1 system development environment as a part of the system development process.

Copyright © 2000 IEEE. All rights reserved.

D.2.2 Configuration tool capabilities

In addition to the node discovery operations, publications, and subscriptions, the configuration tool can remotely invoke the NCAP Block operations

— `SetConfigurationToolAddress`

— `GetConfigurationToolAddress`

— `GoActive`

— `GoInactive`

— `Initialize`

— `Reset`

— `LoadAndLinkCodeFile` (not specified by IEEE 1451.1)

— `InstantiateObjects` (not specified by IEEE 1451.1)

This last two operations are special, vendor-provided NCAP Block operations that result in the downloading and dynamic linking of application code modules and the instantiation of all of the objects Owned by the NCAP Block Object, respectively.

Moreover, the configuration tool can remotely invoke the operation `GetObjectProperties` on any Network Visible Object, in the system for which it has the Object Dispatch Address.

D.2.3 Dynamic application code installation

The next step is to download the application code to the identified nodes. Using each NCAP Block Object's attribute values, the tool determines the application software needed by each node. The tool obtains this information from the system configuration file.

What code goes on which node can be dependent on the set of available nodes. On a node-by-node basis, the tool downloads the required application code from an application code library using the vendor supplied NCAP Block operation `LoadAndLinkCodeFile`. Each downloaded code entity is a dynamically linkable library module carried in a File.

The recipient nodes cache the downloaded code, often in nonvolatile store. This is so that, except for code changes, the application code will not have to be downloaded again during normal operations or even system reboot.

Next, the configuration tool remotely invokes the `InstantiateObjects` operation on each NCAP Block Object. This results in all of the objects Owned by the NCAP Block Object being instantiated, recursively as in Example 1.

The execution of the `InstantiateObjects` operation also results in the initialization of each object Owned by the NCAP Block. This initialization includes the binding of each object's references to other local objects as in Example 1. The code to perform the object instantiation and initialization is a part of the application-specific code developed in the IEEE 1451.1 system development environment and downloaded in the previous step.

The system's Block Objects remain in the `BL_UNINITIALIZED` state.

 Copyright © 2000 IEEE. All rights reserved.

D.2.4 Dynamic communication pattern configuration

The configuration tool is next used to gather information about the Network Visible Server Objects and Client Ports in the system. It uses the following NCAP Block operations to obtain this information from each of the discovered nodes:

— `GetNetworkVisibleServerObjectProperties`

— `GetClientPortProperties`

The acquired information includes the `ObjectProperties` structure for each Network Visible Object and the `ClientPortProperties` structure for each Client Port. This information is placed in the tool's central database.

With this information and information in the system configuration file created during the system development process, the configuration tool uses the following operations to establish the system's client-server communication paths:

— `SetObjectTag` on Server Objects

— `SetServerObjectTag` on Client Ports

It does this by using the Object Names of the Client Ports and the Object Names of the Server Objects along with information contained in the configuration file to associate with each Client Port its target Server Object. Recall that an Object's name is set at Object initialization and cannot be modified. After establishing the Client Port–Server Object associations, the tool sets the Server Object Tag of each Client Port and the tag of its target Server Object to the common value specified in the configuration file.

The only task remaining to set up the client-server communications is to bind the Server Object's Object Dispatch Address in each Client Port to the address of the Port's target Server Object. This binding is done as a consequence of the configuration tool's invocation of the operation `SetClientPortServerObjectBindings` on each NCAP Block Object. The primary argument to this operation is an `ObjectProperties` structure for each Object in the system that is the target Server Object for some Client Port Owned by the NCAP Block Object. The `ObjectProperties` structure for a Network Visible Object includes the values of

— The Object's Object Tag

— The Object's Object Dispatch Address

The tool is next used to set up publications and subscriptions. The Publisher Ports and Subscriber Ports are always Network Visible Objects. From the database of object information the tool previously constructed and the information in the configuration file, it uses the operations

— `SetPublicationTopic`

— `SetPublicationDomain`

— `SetSubscriptionQualifier`

— `SetSubscriptionDomain`

to set Publication Topics, Publication Domains, and Subscription Qualifiers so that each Subscriber receives precisely those publications for which it has registered a subscription with a local Subscriber Port. Recall that the Publication Keys and Subscription Keys are set as a part of the instantiation and initialization of the Publisher Ports and the Subscriber Ports.

Copyright © 2000 IEEE. All rights reserved.

D.2.5 System startup and shutdown

Assume that the systems NCAP Blocks must be activated in a set order for the system to properly startup. Using the configuration tool and information from the configuration file, the `GoActive` operation is remotely invoked on each NCAP Block Object in the required order. This results in the system being operational. If there were no order dependencies the activation of the NCAP Block Objects, the tool could have instead used the `NCAPBlock_Go_Active` publication to make the system operational.

D.2.6 Dynamic system modification

Several issues must be addressed in any scheme for dynamically modifying an operating distributed measurement and control system

— What types of modification are allowed

— How are the modifications effected

— How are consumers of information provided by a modified producer informed of the modification

The IEEE 1451.1 standard provides software support for the following types of system modification:

— Download new application code to an NCAP, replacing the current application code

— Download new application code to an NCAP and dynamically link it with the existing code

— Replace one or more objects with new objects

— Delete one or more objects

— Add one or more objects

— Hot-swap an NCAP

— Add an NCAP node

— Delete an NCAP node

— Enable or disable the application of calibration coefficients in IEEE 1451.X STIMs

— Hot-swap an IEEE 1451.X transducer

The standard does not directly provide mechanisms to effect software modifications other than the `IEEE1451_File` class for encapsulating and downloading code. It is expected that vendors of IEEE 1451.1 implementations will provide functionality similar to the NCAP Block operation, `LoadAndLinkCodeFile`, described above. The direct support of hardware modification is outside of the scope of the standard.

What the IEEE 1451.1 standard does provide are

— Mechanisms for informing a consumer that a producer has been modified in a way that might affect the syntax and/or the semantics of the producer's outputs

— Mechanisms for making a newly added node visible to select NCAP Blocks in the system

The standard has two primary mechanisms for informing consumers about producer changes

— The Block Cookie mechanism for informing a client that its server has changed

— The publication change ID mechanism for informing a Subscriber that a subscribed to publication has changed

 Copyright © 2000 IEEE. All rights reserved.

What action is taken by a consumer on notification that a producer has changed is application specific, and it is outside of the scope of the standard.

For an added node, if its only producers are publishers on existing topics and its only consumers are subscribers to existing topics, then no special action need be taken to integrate the new node into the system.

Similarly, how an NCAP Block responds to the notification of a new node is also outside of the scope of the standard.

D.2.7 An observation

The wastewater treatment system described in Annex A uses only two publications out of the mechanisms listed in Table D.1 for system boot-up, commissioning, startup, and maintenance. This is because this system is statically configured at development time, and it must be taken offline for any modification. The abstract system described above, however, might make use of all of the listed mechanisms. This is because the system is dynamically configured at startup time and it admits dynamic modifications during system operations.

Copyright © 2000 IEEE. All rights reserved.

Annex E

(informative)

NCAP interoperation and network-independent code

E.1 NCAP network interoperation

The IEEE 1451.1 standard provides a network communication abstraction layer. It does this via

— The various IEEE 1451 Port classes

— The `Perform` operation on the `IEEE1451_Entity` class

— The two principle communication protocols

This abstraction layer allows the same object-level design of an Application System to be realized on a multiplicity of networks and NCAPs.

However, the network communication abstraction layer in and of itself does not assure that two IEEE 1451.1 NCAPs connected to the same network can interoperate. We give below a set of necessary and sufficient conditions for the interoperation of NCAPs hosting IEEE 1451.1 software.

Let N_1 and N_2 be two NCAPs connected to the same network and hosting IEEE 1451.1 software implementations. The NCAPs may

— Be from different manufacturers

— Have different operating systems

— Use different network protocol stacks

Moreover, the IEEE 1451.1 software implementations may

— Be from different vendors

— Be implemented in different programming languages

If all of the following conditions hold, then, N_1 and N_2 can interoperate in any IEEE 1451.1 system.

— For any IEEE 1451.1-specified network communication, N_1 and N_2 can syntactically interpret each others' on-the-wire format. This is the case, in particular, if they use the same on-the-wire format.

— N_1 and N_2 use identical syntax and semantics for the representation and the interpretation, respectively, of network Object Dispatch Addresses.

— For any IEEE 1451.1-specified network communication, at the IEEE 1451.1 Object level, the syntactic and semantic results of demarshaling the encoded communication is independent of which NCAP is the sender and which is the receiver. If N_1 and N_2 use a common on-the-wire format, then this requirement is satisfied if, and only if they use equivalent marshaling and demarshaling algorithms.

— For any IEEE 1451.1-specified network communication, N_1 and N_2 have an identical semantic interpretation of the communication in both its encoded and decoded form. If both IEEE 1451.1 implementations are fully IEEE 1451.1 conformant, then this will always be the case.

 Copyright © 2000 IEEE. All rights reserved.

— If the network infrastructure of one of the NCAPs supports communication services outside of the scope of the IEEE 1451.1 standard, for example, a network infrastructure that uses redundant message sending for more reliable IEEE 1451.1 communications, then so does the other. Moreover, any additional on-the-wire data required to support such a service is interpreted the same, both syntactically and semantically, by N_1 and N_2. If both N_1 and N_2 support only the IEEE 1451.1-specified communication services, then this condition is satisfied by default.

E.2 Network-independent IEEE 1451.1 code

An implementation of IEEE 1451.1 can, in part, be network-independent. The only network dependencies in an IEEE 1451.1 implementation are the network side of the various IEEE 1451 Port classes and the network side of the `Perform` operation of the `IEEE1451_Entity` class. If the network dependent portions of these Object classes are implemented as a separate "network library," then the IEEE 1451.1 implementation can be ported from one network to another by simply linking against different network libraries.

Application code built by using such an IEEE 1451.1 implementation is also, in the same sense as above, network independent.

Copyright © 2000 IEEE. All rights reserved.

Annex F

(normative)

IEEE 1451.1 String character set

The character set defined in this clause is used for certain Strings, defined by this standard.

The `StringLanguage` enumeration value for this character set shall be `SL_IEEE`

The `StringCharacterCode` enumeration value for this character set shall be

— `SCC_OCTET1` for a String where the maximum value of any octet in `stringData` is less than 256

— `SCC_OCTET2` for a String where the maximum value of any octet in `stringData` is less than 65536 but greater than 255

— `SCC_OCTET4` for a String where the maximum value of any octet in `stringData` is less than 4294967296 but greater than 65535

The `StringCharacterSet` enumeration value for this character set shall be `SCS_IEEE1451DOT1`.

The columns in Table F.1 shall be interpreted as follows:

— The first column of the table defines the value of the representation for each member character. This value is presented in decimal unsigned integer form. The encoding of this decimal unsigned integer representation into the specified character shall be big endian. The number of octets in this representation shall be defined by the `characterCode` of the String. At the time this standard was written, no character value exceeds 255; hence, the `characterCode` will be `SCC_OCTET1` based on the restrictions on the `characterCode` values above. That is, there will be a single octet per character. In the future, extensions to this character set may result in Strings requiring two or more octets in the representation.

— The second column of the table gives the formal name for the member character. These formal names are the names that appear in the standard.

— The third column, informative, gives a sample print form of the member character that may be used in a configuration or other display tool. The specification of these print form for human-readable languages is outside the scope of this standard.

The members of this character set shall be as defined in Table F.1.

 Copyright © 2000 IEEE. All rights reserved.

Table F.1—IEEE 1451_1 String character set

Character	Formal character name	Example English print form (informative)
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	9	9
10	IEEE1451_Root	IEEE 1451 Root
11	IEEE1451_Entity	IEEE 1451 Entity
12	IEEE1451_Component	IEEE 1451 Component
13	IEEE1451_Parameter	IEEE 1451 Parameter
14	IEEE1451_ParameterWithUpdate	IEEE 1451 Parameter With Update
15	IEEE1451_PhysicalParameter	IEEE 1451 Physical Parameter
16	IEEE1451_ScalarParameter	IEEE 1451 Scalar Parameter
17	IEEE1451_ScalarSeriesParameter	IEEE 1451 Scalar Series Parameter
18	IEEE1451_VectorParameter	IEEE 1451 Vector Parameter
19	IEEE1451_VectorSeriesParameter	IEEE 1451 Vector Series Parameter
20	IEEE1451_TimeParameter	IEEE 1451 Time Parameter
21	IEEE1451_Action	IEEE 1451 Action
22	IEEE1451_File	IEEE 1451 File
23	IEEE1451_PartitionedFile	IEEE 1451 Partitioned File
24	IEEE1451_ComponentGroup	IEEE 1451 Component Group
25	IEEE1451_Block	IEEE 1451 Block
26	IEEE1451_NCAPBlock	IEEE 1451 NCAP Block
27	IEEE1451_FunctionBlock	IEEE 1451 Function Block
28	IEEE1451_BaseTransducerBlock	IEEE 1451 Base Transducer Block
29	IEEE1451_TransducerBlock	IEEE 1451 Transducer Block
30	IEEE1451_Dot2TransducerBlock	IEEE 1451 Transducer Block 1451.2
31	IEEE1451_Dot3TransducerBlock	IEEE 1451 Transducer Block 1451.3
32	IEEE1451_Dot4TransducerBlock	IEEE 1451 Transducer Block 1451.4
33	IEEE1451_Service	IEEE 1451 Service
34	IEEE1451_BasePort	IEEE 1451 Base Port
35	IEEE1451_BaseClientPort	IEEE 1451 Base Client Port

Copyright © 2000 IEEE. All rights reserved.

Table F.1—IEEE 1451_1 String character set (continued)

Character	Formal character name	Example English print form (informative)
36	IEEE1451_ClientPort	IEEE 1451 Client Port
37	IEEE1451_AsynchronousClientPort	IEEE 1451 Asynchronous Client Port
38	IEEE1451_BasePublisherPort	IEEE 1451 Base Publisher Port
39	IEEE1451_PublisherPort	IEEE 1451 Publisher Port
40	IEEE1451_SelfIdentifyingPublisherPort	IEEE 1451 Self-Identifying Publisher Port
41	IEEE1451_EventGeneratorPublisherPort	IEEE 1451 Event Generator Publisher Port
42	IEEE1451_SubscriberPort	IEEE 1451 Subscriber Port
43	IEEE1451_MutexService	IEEE 1451 Mutex Service
44	IEEE1451_ConditionVariableService	IEEE 1451 Condition Variable Service
45	NCAPBlock_Announcement	NCAP Block Announcement
46	Network_Visible_Server_Object_Properties	Network Visible Server Object Properties
47	Request_NCAPBlock_Announcement	Request NCAP Block Announcement
48	Force_NCAPBlock_Announcement	Force NCAP Block Announcement
49	Request_Network_Visible_Server_Object_Properties	Request Network Visible Server Object Properties
50	NCAPBlock_Go_Active	NCAP Block Go Active
51	Timer_Tick	Timer Tick
52	Timer_Properties	Timer Properties
53	Request_Timer_Properties	Request Timer Properties
54	Physical_Parametric_Data	Physical Parametric Data
55	Physical_Parametric_Data_Publisher_Identifying	Physical Parametric Data Publisher Identifying
56	Meta_TEDS	Meta TEDS
57	Channel_TEDS	Channel TEDS
58	Calibration_TEDS	Calibration TEDS
59	Meta_Identification_TEDS	Meta Identification TEDS
60	Channel_Identification_TEDS	Channel Identification TEDS
61	Calibration_Identification_TEDS	Calibration Identification TEDS
62	End_User_TEDS	End User TEDS
63	Permanent_Startup_Set	Permanent Startup Set
64	Configurable_Startup_Set	Configurable Startup Set
65	Transducer_Block_Grouping	Transducer Block Grouping
66–65535		Reserved

Copyright © 2000 IEEE. All rights reserved.

Annex G

(normative)

Assignment of enumeration values to ISO 639 language codes

The enumerated values of the `StringLanguage` enumeration shall be as given in Table G.1. The columns in Table G.1 shall be interpreted as follows:

— The first column of the table defines the enumeration value for each member language. This value is presented in decimal unsigned integer form.

— The second column of the table gives the formal name for the member language. These formal names are the names that appear in the standard.

— The third column, informative, is the two-character code assigned to each language in ISO 639: 1988.

— The fourth column, informative, is the English print form of the name of the language as given in ISO 639: 1988.

Table G.1—`StringLanguage` enumeration

Value	Enumeration	ISO 639 Language code (informative)	Meaning (informative)
0		Reserved	
1	SL_AA	aa	Afar
2	SL_AB	ab	Abkhazian
3	SL_AF	af	Afrikaans
4	SL_AM	am	Amharic
5	SL_AR	ar	Arabic
6	SL_AS	as	Assamese
7	SL_AY	ay	Aymara
8	SL_AZ	az	Azerbaijani
9	SL_BA	ba	Bashkir
10	SL_BE	be	Byelorussian
11	SL_BG	bg	Bulgarian
12	SL_BH	bh	Bihari
13	SL_BI	bi	Bislama
14	SL_BN	bn	Bengali; Bangla
15	SL_BO	bo	Tibetan
16	SL_BR	br	Breton
17	SL_CA	ca	Catalan
18	SL_CO	co	Corsican
19	SL_CS	cs	Czech

Copyright © 2000 IEEE. All rights reserved.

Table G.1—`StringLanguage` enumeration `(continued)`

Value	Enumeration	ISO 639 Language code (informative)	Meaning (informative)
20	SL_CY	cy	Welsh
21	SL_DA	da	Danish
22	SL_DE	de	German
23	SL_DZ	dz	Bhutani
24	SL_EL	el	Greek
25	SL_EN	en	English
26	SL_EO	eo	Esperanto
27	SL_ES	es	Spanish
28	SL_ET	et	Estonian
29	SL_EU	eu	Basque
30	SL_FA	fa	Persian
31	SL_FI	fi	Finnish
32	SL_FJ	fj	Fiji
33	SL_FO	fo	Faroese
34	SL_FR	fr	French
35	SL_FY	fy	Frisian
36	SL_GA	ga	Irish
37	SL_GD	gd	Scots Gaelic
38	SL_GL	gl	Galician
39	SL_GN	gn	Guarani
40	SL_GU	gu	Gujarati
41	SL_HA	ha	Hausa
42	SL_HI	hi	Hindi
43	SL_HR	hr	Croatian
44	SL_HU	hu	Hungarian
45	SL_HY	hy	Armenian
46	SL_IA	ia	Interlingua
47	SL_IE	ie	Interlingue
48	SL_IK	ik	Inupiak
49	SL_IN	in	Indonesian
50	SL_IS	is	Icelandic
51	SL_IT	it	Italian
52	SL_IW	iw	Hebrew
53	SL_JA	ja	Japanese
54	SL_JI	ji	Yiddish
55	SL_JW	jw	Javanese

Copyright © 2000 IEEE. All rights reserved.

Table G.1—`StringLanguage` enumeration `(continued)`

Value	Enumeration	ISO 639 Language code (informative)	Meaning (informative)
56	SL_KA	ka	Georgian
57	SL_KK	kk	Kazakh
58	SL_KL	kl	Greenlandic
59	SL_KM	km	Cambodian
60	SL_KN	kn	Kannada
61	SL_KO	ko	Korean
62	SL_KS	ks	Kashmiri
63	SL_KU	ku	Kurdish
64	SL_KY	ky	Kirghiz
65	SL_LA	la	Latin
66	SL_LN	ln	Lingala
67	SL_LO	lo	Laothian
68	SL_LT	lt	Lithuanian
69	SL_LV	lv	Latvian, Lettish
70	SL_MG	mg	Malagasy
71	SL_MI	mi	Maori
72	SL_MK	mk	Macedonian
73	SL_ML	ml	Malayalam
74	SL_MN	mn	Mongolian
75	SL_MO	mo	Moldavian
76	SL_MR	mr	Marathi
77	SL_MS	ms	Malay
78	SL_MT	mt	Maltese
79	SL_MY	my	Burmese
80	SL_NA	na	Nauru
81	SL_NE	ne	Nepali
82	SL_NL	nl	Dutch
83	SL_NO	no	Norwegian
84	SL_OC	oc	Occitan
85	SL_OM	om	(Afan) Oromo
86	SL_OR	or	Oriya
87	SL_PA	pa	Punjabi
88	SL_PL	pl	Polish
89	SL_PS	ps	Pashto, Pushto
90	SL_PT	pt	Portuguese
91	SL_QU	qu	Quechua

Copyright © 2000 IEEE. All rights reserved.

Table G.1—`StringLanguage` enumeration `(continued)`

Value	Enumeration	ISO 639 Language code (informative)	Meaning (informative)
92	SL_RM	rm	Rhaeto-Romance
93	SL_RN	rn	Kirundi
94	SL_RO	ro	Romanian
95	SL_RU	ru	Russian
96	SL_RW	rw	Kinyarwanda
97	SL_SA	sa	Sanskrit
98	SL_SD	sd	Sindhi
99	SL_SG	sg	Sangho
100	SL_SH	sh	Serbo-Croatian
101	SL_SI	si	Singhalese
102	SL_SK	sk	Slovak
103	SL_SL	sl	Slovenian
104	SL_SM	sm	Samoan
105	SL_SN	sn	Shona
106	SL_SO	so	Somali
107	SL_SQ	sq	Albanian
108	SL_SR	sr	Serbian
109	SL_SS	ss	Siswati
110	SL_ST	st	Sesotho
111	SL_SU	su	Sundanese
112	SL_SV	sv	Swedish
113	SL_SW	sw	Swahili
114	SL_TA	ta	Tamil
115	SL_TE	te	Telugu
116	SL_TG	tg	Tajik
117	SL_TH	th	Thai
118	SL_TI	ti	Tigrinya
119	SL_TK	tk	Turkmen
120	SL_TL	tl	Tagalog
121	SL_TN	tn	Setswana
122	SL_TO	to	Tonga
123	SL_TR	tr	Turkish
124	SL_TS	ts	Tsonga
125	SL_TT	tt	Tatar
126	SL_TW	tw	Twi
127	SL_UK	uk	Ukrainian

Copyright © 2000 IEEE. All rights reserved.

Table G.1—`StringLanguage` enumeration (continued)

Value	Enumeration	ISO 639 Language code (informative)	Meaning (informative)
128	SL_UR	ur	Urdu
129	SL_UZ	uz	Uzbek
130	SL_VI	vi	Vietnamese
131	SL_VO	vo	Volapük
132	SL_WO	wo	Wolof
133	SL_XH	xh	Xhosa
134	SL_YO	yo	Yoruba
135	SL_ZH	zh	Chinese
136	SL_ZU	zu	Zulu
137–252		Reserved	
253	SL_UD1	N/A	User-defined language 1
254	SL_UD2	N/A	User-defined language 2
255	SL_IEEE	N/A	IEEE 1451.1 character set

Copyright © 2000 IEEE. All rights reserved.

Annex H

(normative)

IEEE 1451.2 Transducer Block definition

This annex defines the Transducer Block that shall be used to represent transducers adhering to [IEEE 1451.2].

NOTE—This class definition is contained in an annex, rather than as a clause in Clause 9 of this standard, to maintain the same style used for the IEEE 1451.3 and IEEE 1451.4 Transducer Block classes. In the latter case, it is expected that the details of the Transducer Block specification for the class will appear as part of the relevant standard, that is [IEEE 1451.3] and [IEEE 1451.4], with only the Class Names and identifications being specified in this standard. Thus, 9.6, 9.7, and 9.8 define the Class Name and identification for the relevant transducer standards. Since [IEEE 1451.2] has already been published at the time this standard went to ballot, the only option was to include the remainder of the Block specification as an annex. At some future ballot revision of this standard and of IEEE 1451.2, this material may be moved to the IEEE 1451.2 document.

Class: `IEEE1451_Dot2TransducerBlock`

Parent Class: `IEEE1451_TransducerBlock`

Class ID: 1.1.1.3.1.1

Description: The Dot2 Transducer Block class provides an interface to transducers supporting IEEE 1451.2 Smart Transducer Interface standard, [IEEE 1451.2].

Individual IEEE Std 1451.2 channels are mapped into Parameter Objects. These Parameters are termed Public Transducers in this clause.

The global properties of the IEEE 1451.2 Transducer are provided by operations on the Transducer Block itself.

This Block provides for the representation of a large proportion of the IEEE 1451.2 semantics and likely implementations. The basic operations inherited from the `IEEE1451_TransducerBlock` class provide for direct manipulation of the IEEE 1451.2 interface. These basic operations may be used to access transducer properties or types not easily managed with Component class Objects.

Class summary:

Network Visible operations (see Table H.1)

Table H.1—`IEEE1451_Dot2TransducerBlock` Network Visible operations

Operation Name	Requirement	Operation ID
`EnableDataSequenceSensors`	(o)	10240
`DisableDataSequenceSensors`	(o)	10241
`EnableEventSequenceSensors`	(o)	10242
`DisableEventSequenceSensors`	(o)	10243
`GetWorstCaseParameterTimestampUncertainty`	(o)	10244
`SetSTIMUserString`	(o)	10245
`GetSTIMUserString`	(o)	10246
`SetStatusInterruptMask`	(m)	10247
`GetStatusInterruptMask`	(m)	10248

 Copyright © 2000 IEEE. All rights reserved.

Inherited operations with additional specifications for this class (see Table H.2).

Table H.2—`IEEE1451_Dot2TransducerBlock` modified inherited operations

Operation Name	Requirement
IORead	(m)
IOWrite	(m)
SetIOControl	(m)
GetIOStatus	(m)
EnableCorrections	(m)
DisableCorrections	(m)

Local Operations (see Table H.3).

Table H.3—`IEEE1451_Dot2TransducerBlock` Local operations

Operation Name	Requirement
RegisterInterruptNotification	(m)
DeregisterInterruptNotification	(m)

Publications: There are no publications defined for this class.

Subscriptions: There are no subscriptions defined for this class.

H.1 Operation specifications: Network Visible

H.1.1 Operation `EnableDataSequenceSensors` specification

IDL: `OpReturnCode EnableDataSequenceSensors( );`

There are no operation-specific additions to the Minor Field of the return code enumeration.

H.1.2 Operation `DisableDataSequenceSensors` specification

IDL: `OpReturnCode DisableDataSequenceSensors( );`

There are no operation-specific additions to the Minor Field of the return code enumeration.

H.1.3 Operation `EnableEventSequenceSensors` specification

IDL: `OpReturnCode EnableEventSequenceSensors( );`

There are no operation-specific additions to the Minor Field of the return code enumeration.

Copyright © 2000 IEEE. All rights reserved.

H.1.4 Operation `DisableEventSequenceSensors` specification

```
IDL: OpReturnCode DisableEventSequenceSensors( );
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

H.1.5 Operation `GetWorstCaseParameterTimestampUncertainty` specification

```
IDL: OpReturnCode GetWorstCaseParameterTimestampUncertainty(
          out Uncertainty parameter_timestamp_uncertainty);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

H.1.6 argument `parameter_timestamp_uncertainty` specification

The `parameter_timestamp_uncertainty` argument value shall define the worst case uncertainty for any channel of the time at which trigger acknowledgment signals are received from the IEEE 1451.2 physical interface after the execution of the `UpdateAll` operation.

The value shall be defined during the initialization phase of the Block State Machine. The value shall include

— Uncertainty in the underlying time base of the NCAP or system

— Errors in generating the timestamp in the NCAP

— The worst case of the channel relative timing errors, defined for each channel as specified by the Worst-Case Channel Update Time field of 5.1.3.13 of [IEEE 1451.2]

The `parameter_timestamp_uncertainty` argument shall be based on time expressed in seconds.

H.1.7 Operation `SetSTIMUserString` specification

```
IDL: OpReturnCode SetSTIMUserString(
          in UInteger16 channel_designation;
          in String user_string);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

H.1.7.1 argument `channel_designation` specification

The value N of the `channel_designation` argument shall indicate with which IEEE 1451.2 channel the `user_string` is associated; see 5.7.1 of [IEEE 1451.2].

The value of `channel_designation,` N, shall be limited to

$$0 \leq N \leq \texttt{number_of_transducer_channels}$$

NOTE—In an IEEE 1451.2 STIM, channel number 0 refers to the STIM as a whole. Channel numbers larger than 0 refer to individual transducer channels.

 Copyright © 2000 IEEE. All rights reserved.

H.1.7.2 argument `user_string` specification

The `user_string` argument value shall define the values to be written to the STIM field "End-Users'Application-Specific TEDS data block" as defined by 5.7.3 of [IEEE 1451.2].

NOTE—This field is intended to be used to allow users to enter application information that remains with the transducer, even when it is separated, if allowed, from the NCAP.

H.1.8 Operation `GetSTIMUserString` specification

```
IDL: OpReturnCode GetSTIMUserString(
           in UInteger16 channel_designation;
           out String user_string);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

H.1.9 Operation `SetStatusInterruptMask` specification

```
IDL: OpReturnCode SetStatusInterruptMask(
           in OctetArray interrupt_mask);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

H.1.9.1 argument `interrupt_mask` specification

The `interrupt_mask` argument value shall encode to an Octet Array of length 4. The bits of this argument shall be as defined by 4.9 of [IEEE 1451.2].

The `interrupt_mask` argument determines which status information shall cause the IEEE 1451.2 interface to generate an interrupt signal to the NCAP.

— The 0th and 1st octets of the Array shall correspond to the Standard Status defined by 4.8.1 of [IEEE 1451.2].

— The 2nd and 3rd octets of the Array shall correspond to the Auxiliary Status defined by 4.8.2 of [IEEE 1451.2].

H.1.10 Operation `GetStatusInterruptMask` specification

```
IDL: OpReturnCode GetStatusInterruptMask(
           out OctetArray interrupt_mask);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

H.2 Operation specifications: inherited

The following inherited operations shall all have Full Implementations:

— `IORead`

— `IOWrite`

— `SetIOControl`

Copyright © 2000 IEEE. All rights reserved.

— `GetIOStatus`

— `EnableCorrections`

— `DisableCorrections`

H.2.1 Inherited operation `IORead` specification

The IORead operation is inherited from the `IEEE1451_BaseTransducerBlock` class. The following class-specific definitions for the operation arguments shall be used.

H.2.1.1 Inherited argument `io_input_arguments` specification

The `io_input_arguments` Argument Array 0th member Argument shall encode to an Octet Array of length 2.

The 0th and 1st octets shall correspond to the address structure specified by 4.4.1 of [IEEE 1451.2].

H.2.1.2 Inherited argument `io_output_arguments` specification

The `io_output_arguments` Argument Array 0th member Argument shall decode from an Octet Array.

The meaning of the octets shall be specific to the information being read as specified by IEEE 1451.2 and is dependent on the value of the `io_input_arguments.`

H.2.2 Inherited operation `IOWrite` specification

The `IOWrite` operation is inherited from the `IEEE1451_BaseTransducerBlock` class. The following class-specific definitions for the operation arguments shall be used.

H.2.2.1 Inherited argument `io_input_arguments` specification

The `io_input_arguments` Argument Array 0th member Argument shall encode to an Octet Array.

— The 0th and 1st octets shall correspond to the field address structure specified by 4.4.1 of [IEEE 1451.2].

— The meaning of the remaining octets shall be specific to the information being written as specified by IEEE 1451.2 and are dependent on the value of the 0th and 1st octets.

H.2.2.2 Inherited argument `io_output_arguments` specification

The `io_output_arguments` argument is not defined for this operation.

H.2.3 Inherited operation `SetIOControl` specification

The `SetIOControl` operation is inherited from the `IEEE1451_BaseTransducerBlock` class. The following class-specific definitions for the operation arguments shall be used.

 Copyright © 2000 IEEE. All rights reserved.

H.2.3.1 Inherited argument `control_arguments` specification

The `control_arguments` Argument Array 0th member Argument shall encode to a `UInteger8`. The value of the `UInteger8` shall be the channel number on the IEEE 1451.2 interface for which control is being set. Channel number 0 shall correspond to global control governing the entire STIM.

The `control_arguments` Argument Array 1st member Argument shall encode to a `UInteger16`. The value of this `UInteger16` shall specify the desired IEEE 1451.2 command as defined by the field Control in 4.7 of [IEEE 1451.2].

H.2.4 Inherited operation `GetIOStatus` specification

The `GetIOStatus` operation is inherited from the `IEEE1451_BaseTransducerBlock` class. The following class-specific definitions for the operation arguments shall be used.

H.2.4.1 Inherited argument `io_input_arguments` specification

The `io_input_arguments` Argument Array 0th member Argument shall encode to a `UInteger8`. The value of the `UInteger8` shall be the channel number on the IEEE 1451.2 interface for which status is sought. Channel number 0 shall correspond to global status governing the entire STIM.

H.2.4.2 Inherited argument `status` specification

— The `status` Argument Array 0th member Argument shall decode to an Octet Array of length 4. The 0th and 1st octets of the Array shall correspond to the field Standard Status defined by 4.8.1 of [IEEE 1451.2].

— The 2nd and 3rd octets of the Array shall correspond to the field Auxiliary Status defined by 4.8.2 of [IEEE 1451.2].

H.3 Operation specifications: Local

H.3.1 Local operation `RegisterInterruptNotification` specification

```
IDL: OpReturnCode RegisterInterruptNotification(
           in <local operation reference>
              notification_operation_reference,
           in UInteger16 registration_id);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

H.3.1.1 argument `notification_operation_reference` specification

The `notification_operation_reference` argument value shall be a local reference to the local operation of the registering object that shall be invoked to signal notification. The signature of the referenced notification operation shall be

```
OpReturnCode <local operation name>(
           in UInteger16 registration_id,
           in TimeRepresentation timestamp);
```

Copyright © 2000 IEEE. All rights reserved.

H.3.1.2 argument `timestamp` specification

The `timestamp` argument value shall be the value of the occurrence time of the class-specific state change mandating the notification.

H.3.2 Local operation `DeregisterInterruptNotification` specification

```
IDL: OpReturnCode DeregisterInterruptNotification(
           in <local operation reference>
              notification_operation_reference,
           in UInteger16 registration_id);
```

There are no operation-specific additions to the Minor Field of the return code enumeration.

H.4 Dot2 Transducer Block class behavior specification

IEEE 1451.2 Transducers may provide multiple channels, each channel serving an individual transducer. For each channel, considerable information specifying the measurement or actuation properties of the transducer, such as units and calibration, are provided in data structures named TEDS. IEEE 1451.2 provides for seven types of transducers, called

— Actuators

— Sensors

— Buffered sensors

— Data sequence sensors

— Buffered data sequence sensors

— Event sequence sensors

— General transducers

Additional information regarding the collection of channels and relationships among channels is also provided. For example, a group of channels representing the three axes of a vector measurement such as acceleration may be defined.

The Dot2 Transducer Block models each of these transducer types by exposing it as a Network Visible Object on its interface as an instance of a subclass of the Component class. These interface Objects are termed Public Transducers. During the initialization phase of the block life cycle, the Transducer Block or a third party establishes the mapping between the individual channels of the IEEE 1451.2 Transducer and the Public Transducers of the Transducer Block. Each of these Public Transducers shall represent IEEE 1451.2 specified information associated with the channel or channels represented by the Object. These Public Transducers, as contrasted to IEEE 1451.2 Transducers, represent the abstraction of one or more of the underlying channels present on the IEEE 1451.2 interface.

During the initialization of a Dot2 Transducer Block, zero or more instances of Parameters, Files, Component Groups, and Event Generator Publisher Ports shall be instantiated and may be registered to create the Network Visible interface appropriate to the specific transducer. This activity shall be successfully completed before the `BL_INACTIVE` state defined for all Blocks is entered.

 Copyright © 2000 IEEE. All rights reserved.

The selection of the interface Objects and the assignment of their names during registration or instantiation of those Objects shall be based on information supplied from the IEEE 1451.2 TEDS and the following clauses.

H.4.1 Initialization: Determination of required Files and their naming conventions

The following IEEE 1451.2-defined TEDS data structures shall be represented by this Transducer Block class as Objects of class `IEEE1451_PartitionedFile` or `IEEE1451_File`.

The contents of these Files shall deliver the exact representations and information as defined in Clause 5 of [IEEE 1451.2].

The properties of these Files shall be as defined in Table H.4.

— The File Object Name shall be the name returned by the `GetObjectName` operation on the File Object. The File Object Name shall be represented in the IEEE 1451.1 character set defined in Annex F.

— The File Object Name shall identify the Files associated with the IEEE 1451.2 TEDS component given in Table H.4.

— `IEEE1451_File` class Objects shall be used for TEDS pertaining to the entire STIM.

— `IEEE1451_PartitionedFile` class Objects shall be used for channel TEDS. The `partition_id` shall correspond to the IEEE 1451.2 channel number for which the TEDS are sought. `partition_id` shall be represented as a number N, where

$$0 < N \leq \texttt{number_of_transducer_channels}$$

Table H.4—Required files

IEEE 1451.2 TEDS component	File Object Name	File	Partitioned File	IEEE 1451.2 defining clause (informative)	IEEE 1451.2 requirement (informative)
Meta-TEDS	`Meta_TEDS`	X		5.1	(m)
Channel TEDS	`Channel_TEDS`		X	5.2	(m)
Calibration TEDS	`Calibration_TEDS`		X	5.3	(o)
Meta-Identification TEDS	`Meta_Identification_TEDS`	X		5.4	(o)
Channel Identification TEDS	`Channel_Identification_TEDS`		X	5.5	(o)
Calibration Identification TEDS	`Calibration_Identification_TEDS`		X	5.6	(o)
End User TEDS	`End_User_TEDS`		X	5.7	(o)

H.4.2 Initialization: TEDS STIM fields used to determine the interface

The IEEE 1451.2 TEDS fields of interest are defined in the H.4.2.1 through H.4.2.13. The title of each subclause includes the formal name for the field as it is used throughout this annex. The values of these fields for a specific STIM are obtained by means specified in IEEE 1451.2.

Copyright © 2000 IEEE. All rights reserved.

The values assigned in the following enumerations correspond to the values defined in IEEE 1451.2. The precise meaning of each member shall be in accordance with IEEE 1451.2. For all enumerations taken from IEEE 1451.2, the members have been augmented by a suitable prefix, for example, CTK_, for use in this standard.

H.4.2.1 Channel Type Key: definition

The value of the Channel Type Key field is taken from the enumeration shown in Table H.5, see 5.2.3.8 of [IEEE 1451.2].

IDL: `enumeration ChannelTypeKey;`

Table H.5—`ChannelTypeKey` enumeration

Enumeration	Value
CTK_SENSOR	0
CTK_ACTUATOR	1
CTK_EVENT_SEQUENCE_SENSOR	2
CTK_DATA_SEQUENCE_SENSOR	3
CTK_GENERAL_TRANSDUCER	4
CTK_BUFFERED_SENSOR	5
CTK_BUFFERED_DATA_SEQUENCE_SENSOR	6
Reserved	7–127
User-defined	128–255

H.4.2.2 Channel Grouping: definition

The value of the Channel Grouping field shall be taken from the enumeration shown in Table H.6; see 5.1.3.26 of [IEEE 1451.2].

IDL: enumeration `ChannelGrouping;`

Table H.6—`ChannelGrouping` enumeration

Enumeration	Value
CG_ARBITRARY_RELATION	0
CG_RECTANGULAR_COORDINATES	1
CG_CYLINDRICAL_COORDINATES	2
CG_SPHERICAL_COORDINATES	3
CG_PLANETARY_COORDINATES	4
CG_TEMPORAL_COORDINATES	5
CG_RGB_COLOR_COORDINATES	6
CG_ANALOG_EVENT_SEQUENCE_PLUS	7
CG_SENSOR_PLUS	8
CG_SAMPLE_INTERVAL_VIRTUAL_ACTUATOR	9
CG_DIGITAL_EVENT_SEQUENCE_PLUS	10
Reserved	11–127
User-defined	128–255

 Copyright © 2000 IEEE. All rights reserved.

H.4.2.3 Channel Data Repetitions: definition

The value of the Channel Data Repetitions field is a `UInteger16` whose value shall specify whether the data represents

— A single measurement, where the value = 0

OR

— A series of measurements where the value > 0

Refer to 5.2.3.17 of [IEEE 1451.2].

H.4.2.4 Physical Units: definition

The value of the Physical Units field, see 5.2.3.9 of [IEEE 1451.2], shall be taken from the enumeration shown in Table H.7, see 3.3.8 of [IEEE 1451.2].

IDL: enumeration `PhysicalUnitsInterpretation;`

Table H.7—`PhysicalUnitsInterpretation` enumeration

Enumeration	Value
`PUI_BASE_SI_UNITS`	0
`PUI_RATIO_BASE_SI_UNITS`	1
`PUI_LOG_BASE_SI_UNITS`	2
`PUI_LOG_RATIO_BASE_SI_UNITS`	3
`PUI_DIGITAL_DATA`	4
`PUI_ARBITRARY`	5
Reserved	6–255

H.4.2.5 Channel Data Model: definition

The value of the Channel Data Model field is taken from the enumeration in Table H.8, see 5.2.3.14 of [IEEE 1451.2].

IDL: enumeration `ChannelDataModel;`

Table H.8—`ChannelDataModel` enumeration

Enumeration	Value
`CDM_N_BYTE_INTEGER`	0
`CDM_FLOAT32`	1
`CDM_FLOAT64`	2
`CDM_N_BYTE_FRACTION`	3
Reserved	4–255

Copyright © 2000 IEEE. All rights reserved.

H.4.2.6 Calibration Key: definition

The value of the Calibration Key field shall be taken from the enumeration shown in Table H.9, see 5.2.3.2 of [IEEE 1451.2].

IDL: enumeration `CalibrationKey;`

Table H.9—`CalibrationKey` enumeration

Enumeration	Value
CK_CAL_NONE	0
CK_CAL_FIXED	1
CK_CAL_MODIFIABLE	2
CK_CAL_SELF	3
CK_CAL_CUSTOM	4
CK_STIM_CAL_FIXED	5
CK_STIM_CAL_MODIFIABLE	6
CK_STIM_CAL_SELF	7
Reserved	8–255

H.4.2.7 Group Name: definition

The value of the Group Name field shall be as defined in 5.4.3.21 of [IEEE 1451.2]. This field is a member of the Meta-Identification TEDS of an IEEE 1451.2 STIM applicable to the entire STIM.

H.4.2.8 Channel Description: definition

The value of the Channel Description field shall be as defined in 5.5.3.14 and 5.5.3.15 of [IEEE 1451.2]. This field is a member of the Channel Identification TEDS of an IEEE 1451.2 STIM applicable to an individual channel.

H.4.2.9 Member Channel Numbers List: definition

The value of the Member Channel Numbers List field shall be as defined in 5.1.3.28 of [IEEE 1451.2]. This field is a member of the Meta TEDS of an IEEE 1451.2 STIM applicable to the entire STIM.

H.4.2.10 Standard Status: definition

The value of the Standard Status field shall be as defined in 4.8.1 of [IEEE 1451.2].

H.4.2.11 Auxiliary Status: definition

The value of the Auxiliary Status field shall be as defined in 4.8.2 of [IEEE 1451.2].

H.4.2.12 NACK: definition

The NACK signal shall be as defined in 6.2 of [IEEE 1451.2].

 Copyright © 2000 IEEE. All rights reserved.

H.4.2.13 Timing Correction: definition

The value of the Timing Correction field shall be as defined in 5.2.3.27 of [IEEE 1451.2].

H.4.3 Initialization: Selection of interface components to represent channels

The channels of an IEEE 1451.2 STIM shall be represented by a combination of the following:

— `IEEE1451_Parameter` Objects
— `IEEE1451_ComponentGroup` Objects
— The operations of the Block itself

The specific combinations depend on the contents of the STIM TEDS fields listed in H.4.2.

The determination of the representation of a channel shall proceed by resolving the following issues:

Issue 1: Determine whether the channel represents a special case.

Issue 2: Determine whether the channel is to be represented as one dimension of a Vector Parameter, as a Scalar Parameter, or as a Scalar Parameter plus an entry as a member of a Component Group Object. Each group defined for a Dot2 Transducer Block shall be represented by a separate Component Group Object.

NOTE—[IEEE 1451.2] provides certain grouping information to alert the user that the channels included in the group are intended to be sampled simultaneously to produce a snapshot in time. The usual case is for components of a spatial vector such as velocity where measurements of the three components at different points in time could lead to misinterpretation. The IEEE 1451.1 Vector Parameter types are intended to model this behavior in that a `UpdateAndRead` operation, for example, is intended to be implemented with the global trigger facility of [IEEE 1451.2]. For IEEE 1451.2-suggested groupings other than spatial vectors it can not be assumed that this is the reason for the grouping. If it were known that a time snapshot was desirable, then a Vector Parameter would be appropriate. For nonspatial groupings, the Component Group representation can be used. In this case, it is entirely up to the application to manage the relative updating of the channels forming the group. Component Group Objects defined in this standard hold information, but do not themselves enforce any behavior on the members.

Issue 3: Determine whether the Parameter is to be represented by a Series Parameter.

Issue 4: Determine the Physical Parameter Type of the selected Parameter.

The values of the member elements of the appropriate metadata structures for the selected Parameters shall be determined from the IEEE 1451.2 TEDS of the transducer according to the steps outlined in this clause. Member values determined from the IEEE 1451.2 Meta TEDS or Channel TEDS shall be read only, since by 4.4.3 of [IEEE 1451.2], the corresponding TEDS values are read only.

The above issues are resolved by the following steps.

Step 1: Restrictions based on the type of transducer served by the channel

Based on the value of the Channel Type Key, a channel may be treated as a special case as follows:

Step 1—Alternative A:

The channel is not a special case if the Channel Type Key is one of the following:

— `CTK_SENSOR`
— `CTK_ACTUATOR`

Copyright © 2000 IEEE. All rights reserved.

— CTK_DATA_SEQUENCE_SENSOR

— CTK_BUFFERED_SENSOR

— CTK_BUFFERED_DATA_SEQUENCE_SENSOR

When the channel is not a special case, it may be represented by the appropriate Physical Parameter as determined by the remaining steps.

Step 1—Alternative B:

If the Channel Type Key is CTK_GENERAL_TRANSDUCER, the channel may be represented by the appropriate Physical Parameter. It is recommended that general transducer channels not be represented by a Parameter but instead be included as one of the unrepresented_channel_numbers for this class.

NOTE—For general transducers, [IEEE 1451.2] does not specify the relationship of the triggers and data collection. This type of transducer is more readily managed by a subclass defining a Block or Parameter designed specifically for the transducer.

Step 1—Alternative C:

If the Channel Type Key is CTK_EVENT_SEQUENCE_SENSOR, the channel shall be represented by an instance of the IEEE1451_TimeParameter class. A Channel Data Repetitions value > 0 for this type of transducer shall be considered an error. This channel type may also be included in certain groups as discussed in Step 2.

NOTE—[IEEE 1451.2] does not specify the behavior of a channel of this type with Channel Data Repetitions > 0.

Step 1—Alternative D:

An ill-defined case exists if any of the following statements are true:

— The Channel Type Key is user-defined or a reserved value.

— Inconsistent TEDS data or known errors are encountered in parsing the TEDS.

— The channel falls outside of the cases specified in this clause.

For this case, it is recommended that the channel not be represented by a Parameter, but instead be included as one of the unrepresented_channel_numbers for this class.

NOTE—This type of transducer is more readily managed by a subclass defining a Block or Parameter designed specifically for the transducer.

Unrepresented channels may be accessed by means of the operations inherited from the IEEE1451_BaseTransducerBlock class.

Step 2: Restrictions based on grouping information involving the channel

Channels may be included in grouping information provided by the TEDS of an IEEE 1451.2 STIM.

Step 2—Alternative A:

A Parameter instance of the classes IEEE1451_VectorParameter or IEEE1451_VectorSeriesParameter shall be used for a channel that is a member of a group with a Channel Grouping enumeration value of

— CG_RECTANGULAR_COORDINATES

— CG_CYLINDRICAL_COORDINATES

 Copyright © 2000 IEEE. All rights reserved.

— CG_SPHERICAL_COORDINATES

— CG_PLANETARY_COORDINATES

In this case, there shall be no Component Group representing the group specified by the TEDS.

Step 2—Alternative B:

A Parameter instance of the classes IEEE1451_VectorParameter or IEEE1451_VectorSeriesParameter may be used for channel that is a member of a group with Channel Grouping enumeration value of

— CG_ARBITRARY_RELATION

— CG_TEMPORAL_COORDINATES

— CG_RGB_COLOR_COORDINATES

Alternatively, a channel in this category may be represented by a Parameter instance of the classes IEEE1451_ScalarParameter or IEEE1451_ScalarSeriesParameter with the grouping information represented by including the channel as a member of an instance of the Component Group class representing the group specified in the TEDS. It is recommended that this alternative representation be used.

NOTE—In the first of the previous two cases, a Vector Parameter was recommended for groupings identified from the TEDS, as a spatial grouping. In the second case, a Vector Parameter is allowed but a collection of Scalar Parameters and a grouping are recommended. One of the primary reasons is that for spatial vectors, the mathematics of vector manipulations are well defined. While this may be true in certain instances of the second case, in general the mathematics of manipulating these vectors is not known.

Step 2—Alternative C:

A Parameter instance of the classes IEEE1451_ScalarParameter or IEEE1451_ScalarSeriesParameter shall be used for a channel that is a member of a group with any Channel Grouping enumeration value of

— CG_SENSOR_PLUS

— CG_SAMPLE_INTERVAL_VIRTUAL_ACTUATOR

The Parameter representing the channel shall also be included as a member of an instance of the Component Group representing the group specified in the TEDS.

Step 2—Alternative D:

A Parameter instance of the classes IEEE1451_TimeParameter or IEEE1451_ScalarParameter shall be used for a channel that is a member of a group with any Channel Grouping enumeration value of

— CG_ANALOG_EVENT_SEQUENCE_PLUS

— CG_DIGITAL_EVENT_SEQUENCE_PLUS

If the channel's Channel Type Key is CTK_EVENT_SEQUENCE_SENSOR, the channel shall be represented by a Time Parameter. Otherwise, a Scalar Parameter shall be used.

The Parameter representing the channel shall also be included as a member of an instance of the Component Group representing the group specified in the TEDS.

Copyright © 2000 IEEE. All rights reserved.

Step 2—Alternative E:

Unless otherwise indicated in Step 1, if no grouping information concerning a channel is provided by the IEEE 1451.2 TEDS, the channel shall be represented by an instance of the classes

— `IEEE1451_ScalarParameter`
— `IEEE1451_ScalarSeriesParameter`
— `IEEE1451_TimeParameter`

Step 3: Restriction on Parameters based on Channel Data Repetitions TEDS data

The Channel Data Repetitions value must be considered if the preceding steps have specified that the channel is to be represented as one of the following classes:

— `IEEE1451_ScalarParameter`
— `IEEE1451_ScalarSeriesParameter`
— `IEEE1451_VectorParameter`
— `IEEE1451_VectorSeriesParameter`

For this case the non-Series Parameter classes shall be selected if the Channel Data Repetitions value = 0 and the Series Parameter classes shall be selected if the value > 0.

Step 4: Selection of Parameter metadata properties

In this step the term “analog” shall be interpreted as one of the Parameter types

— `PP_SCALAR_ANALOG`
— `PP_SCALAR_ANALOG_SERIES`
— `PP_VECTOR_ANALOG`
— `PP_VECTOR_ANALOG_SERIES`

In this step the term “discrete” shall be interpreted as one of the Parameter types

— `PP_SCALAR_DISCRETE`
— `PP_SCALAR_DISCRETE_SERIES`
— `PP_VECTOR_DISCRETE`
— `PP_VECTOR_DISCRETE_SERIES`

In this step the term “digital” shall be interpreted as one of the Parameter types

— `PP_SCALAR_DIGITAL`
— `PP_SCALAR_DIGITAL_SERIES`
— `PP_VECTOR_DIGITAL`
— `PP_VECTOR_DIGITAL_SERIES`

The rules of this step shall be applied if the preceding steps have specified that the channel is to be represented as one of the following classes:

 Copyright © 2000 IEEE. All rights reserved.

— IEEE1451_ScalarParameter

— IEEE1451_ScalarSeriesParameter

— IEEE1451_VectorParameter

— IEEE1451_VectorSeriesParameter

The Physical Parameter Type, that is the value returned in the output argument parameter_type by the GetPhysicalParameterType operation on the Parameter representing the channel, shall be determined by the following alternatives:

Step 4—Case A: TB_CORRECTED state

For all Calibration Key values except CK_CAL_CUSTOM the following rules shall apply:

— The Physical Parameter Type shall be “analog” for the following combinations of Physical Units and Channel Data Model TEDS fields:

[[PUI_BASE_SI_UNITS, or

PUI_RATIO_BASE_SI_UNITS, or

PUI_LOG_BASE_SI_UNITS, or

PUI_LOG_RATIO_BASE_SI_UNITS]

AND

[CDM_N_BYTE_INTEGER, or

CDM_FLOAT32, or

CDM_FLOAT64, or

CDM_N_BYTE_FRACTION]]

OR

[PUI_ARBITRARY

AND

[CDM_FLOAT32, or

CDM_FLOAT64]]

— The Physical Parameter Type shall be “discrete” for the combination of Physical Units and Channel Data Model TEDS fields: PUI_ARBITRARY AND CDM_N_BYTE_INTEGER

— The Physical Parameter Type shall be “digital” for the combination of Physical Units and Channel Data Model TEDS fields: PUI_DIGITAL_DATA AND CDM_N_BYTE_INTEGER

— The units returned by the GetUnits operation on the selected subclass of IEEE1451_PhysicalParameter shall be the entire Physical Units structure of the STIM TEDS

— The nature of the calibration corrections performed shall be based on the value of the Calibration Key field.

NOTE—This may require transformation between the datatype received from the STIM and that specified by the Parameter class. For some Calibration Key fields, the correction may be the identity transformation.

Step 4—Case B: TB_UNCORRECTED state and Calibration Key values of

Copyright © 2000 IEEE. All rights reserved.

— CK_CAL_FIXED

— CK_CAL_MODIFIABLE

— CK_CAL_SELF

For the case where the Physical Parameter Type value in the TB_CORRECTED case is "analog," the parameter type in the TB_UNCORRECTED case shall be "analog" for Channel Data Model field values of CDM_FLOAT32 or CDM_FLOAT64, otherwise the Physical Parameter Type shall be "discrete."

The "digital" and "discrete" Physical Parameter Type values of the TB_CORRECTED case are retained in the TB_UNCORRECTED case.

The units returned by GetUnits shall be the default value for the datatype.

Step 4—Case C: TB_UNCORRECTED state and Calibration Key values of

— CK_CAL_NONE

— CK_STIM_CAL_FIXED

— CK_STIM_CAL_MODIFIABLE

— CK_STIM_CAL_SELF

This case follows the rules of Case A for the TB_CORRECTED state.

Step 4—Case D: Default

If none of the cases of this clause apply, the channel shall not be represented by a Public Transducer but shall instead be included as one of the unrepresented_channel_numbers for this class. This type of transducer is more readily managed by a subclass defining a Block or Parameter designed specifically for the transducer. These channels may be accessed by means of the operations inherited from the IEEE1451_BaseTransducerBlock class.

H.4.4 Parameter naming conventions

The Object Name returned by the operation GetObjectName for a Parameter serving as a Public Transducer shall be determined according to the first of the following alternatives to apply:

— If the Parameter is a Vector Parameter, and the Group Name field of the Meta-Identification TEDS of the STIM is available and is not the null String, the Object Name shall be the Group Name.

— If the Parameter is a Vector Parameter, and the Group Name field of the Meta-Identification TEDS of the STIM is not available for the grouped channels, but on the basis of a search starting with the 0th channel of the channel grouping, the Channel Description field of the Channel Identification TEDS is available for at least one of the dimensions of the vector, then Object Name shall be the first such Channel Description field obtained.

— If the Parameter is a Scalar Parameter, and the Channel Description field of the Channel Identification TEDS is available for the channel, then Object Name shall be the Channel Description field.

— Otherwise the name of the Parameter shall consist of a sequence of characters designating the Parameter class followed by an ordinal number assigned starting with the value 0 for the first Parameter named on the basis of this alternative. For this alternative, the String shall be represented in the IEEE 1451.1 character set; see Annex F.

 Copyright © 2000 IEEE. All rights reserved.

— String lengths under this clause may be limited to an implementation-specified length.

NOTE—The referenced Strings defined by IEEE 1451.2 are description fields and may be longer than desired. It is recommended that implementers of IEEE 1451.2 STIMs make the first 16 characters of these Strings meaningful in the context of a Dot2 Transducer Block.

H.4.5 Grouping specification conventions

The `object_name` value returned by the `GetObjectName` operation of a Component Group Object, that is the Object Name of the Object, shall be determined according to the first of the following alternatives to apply:

— If the Group Name field of the Meta-Identification TEDS of the STIM is available and is not the null String, the `object_name` shall be the Group Name.

— If the Group Name field is not present, the `object_name` shall be `Transducer_Block_GroupingX`. Here X is an ordinal assigned starting at 0 for the first such grouping specified. This `object_name` shall be represented in the IEEE 1451.1 character set defined in Annex F.

NOTE—The referenced Strings defined by [IEEE 1451.2] are description fields and may be longer than desired. It is recommended that implementers of IEEE 1451.2 STIMs make the first 16 characters of these Strings meaningful in the context of a Dot2 Transducer Block.

The Parameters represented in the group shall be represented as an `ObjectProperties` data structure that is one of the members of the `ObjectPropertiesArray` returned by the `GetMembers` operation on the Group Component. The member fields of the `ObjectProperties` data structure shall correspond to the object properties of the corresponding Parameter.

The channel number of each Parameter so represented shall appear as a member of the group defined by the Member Channel Numbers List of the STIM Meta TEDS.

If this rule results in the implicit inclusion in the group members of a channel not belonging to the group, then sufficient additional Scalar Parameters representing the common members of the overlapping groups shall be present at the Transducer Block interface to avoid this situation.

Example: Suppose the STIM Meta TEDS indicates that channels 1, 2, and 3 form a spatial vector group. These channels will be represented as a Vector Parameter by the previous clauses, to ensure that the three channels are sampled as a time snapshot. Suppose further that channels 3 and 4 are identified as a member of a second grouping, GroupA, which by the previous clauses is represented as a Group Component. Channel 3 is common to both groups, yet an application updating channels 3 and 4 will, depending on the implementation of Vector Parameters, violate the integrity of the Vector Parameter's value. This clause states that there shall be an additional Scalar Parameter representing channel 3 and that this Scalar Parameter shall be the representative of channel 3 in GroupA.

H.4.6 Status Interrupt Mask behavior specification

During initialization, the value of the Standard Status or Auxiliary Status registers of the STIM may be set.

Subclause 4.9 of [IEEE 1451.2] mandates the following:

— On power-up of the STIM, the default value for the Standard Status register is all ones (i.e., all standard status bits can generate interrupts).

Copyright © 2000 IEEE. All rights reserved.

— The default value for the Auxiliary Status register is all zeroes (i.e., no auxiliary status bits can generate interrupts).

— The contents of these status registers are not changed by any STIM or channel IEEE 1451.2 reset operation.

H.4.7 Interrupt service support

IEEE 1451.2 Transducer interfaces generate an interrupt signal configurable by the `interrupt_mask`, of the `SetStatusInterruptMask` operation H.1.9 and the mechanism defined in 4.10 of [IEEE 1451.2].

Each detection of this interrupt signal by the Transducer Block shall result in the invocation of the local operation registered by the IEEE 1451.1 `RegisterInterruptNotification` operation on the Transducer Block. If no local operation has been registered, IEEE 1451.2 interrupts shall have no effect visible at the interface of the Transducer Block.

H.4.8 Notification behavior specification

The receipt of an IEEE 1451.2 interface trigger_ack signal resulting from the application of an IEEE 1451.1 update operation on a Public Transducer or a IEEE 1451.1 `UpdateAll` operation on the Transducer Block itself shall result in the activation of any registered, by IEEE 1451.1 `RegisterNotifyOnUpdate`, local notification operations on the affected Parameters and of the notification operation, if registered, of the Transducer Block itself.

H.4.9 Triggering

IEEE 1451.2 supports the following two modes of triggering:

— Individual channel triggering

— Group triggering

H.4.9.1 Individual channel triggering behavior specification

Individual channel triggering may be activated by an IEEE 1451.1 `UpdateAndRead` or `WriteAndUpdate` of the Public Transducer associated with a channel. The relationship of reading, writing, and triggering shall be as defined in 6.3 [IEEE 1451.2].

H.4.9.2 Group triggering behavior specification

IEEE 1451.2 group triggering shall be activated by invoking the IEEE 1451.1 `UpdateAll` operation and shall result in all IEEE 1451.2 channels being triggered.

The use of IEEE 1451.1 `UpdateAll` is strongly recommended for any application that requires that the values in the Public Transducers represent a collection of readings made at the same time. Vector Parameters should be implemented by group triggering whenever possible.

H.4.10 Timestamp behavior specification

The values returned by the `GetLastTimestamp` operation on a Parameter used as a Public Transducer shall be the timestamp generated by the IEEE 1451.2 NACK signal corrected by the value of the Timing Correction field of the channel.

 Copyright © 2000 IEEE. All rights reserved.

`GetLastUpdateTimestamp` and `GetWorstCaseParameterTimestampUncertainty` may be Interface Only Implementations if the implementation does not provide the ability to timestamp events; otherwise, they shall be Full Implementations.

H.4.11 Operations `EnableDataSequenceSensors` and `DisableDataSequenceSensors` behavior specifications

The IEEE 1451.1 operations `EnableDataSequenceSensors` and `DisableDataSequenceSensors` may both be Interface Only Implementations if the IEEE 1451.2 transducer does not have any channels of type `CTK_Data_Sequence_Sensor`, otherwise these operations shall have a Full Implementation. For Full Implementations, these operations shall enable or disable all data sequence sensors present at the IEEE 1451.2 interface.

H.4.12 Operations `EnableEventSequenceSensors` and `DisableEventSequenceSensors` behavior specifications

The IEEE 1451.1 operations `EnableEventSequenceSensors` and `DisableEventSequenceSensors` may both be an Interface Only Implementation if the IEEE 1451.2 transducer does not have any channels of type `CTK_EVENT_SEQUENCE_SENSOR`, otherwise these operations shall have a Full Implementation. For Full Implementations, these operations shall enable or disable all event sequence sensors present at the IEEE 1451.2 interface.

H.4.13 Operations `SetSTIMUserString` and `GetSTIMUserString` behavior specifications

The IEEE 1451.1 operations `SetSTIMUserString` and `GetSTIMUserString` may both be an Interface Only Implementation if the IEEE 1451.2 transducer does not have provision for this field in its TEDS, otherwise these operations shall have a Full Implementation. For Full Implementations, these operations shall set and get the value of the designated "End-User's Application-Specific TEDS data block" field; see H.1.7.

H.4.14 Specific Dot2 Transducer Block operation restrictions based on state

In addition to and conformant with the restrictions of 9.1.3.2.1, 9.1.3.2.2, 9.1.3.2.3, and 9.1.3.2.4, the allowed operations defined by this class shall be restricted as defined in Table H.10. Where an operation is indicated as operational, it still shall be restricted to specific uses defined by the restrictions of 9.1.3.2.1, 9.1.3.2.2, 9.1.3.2.3, and 9.1.3.2.4.

Table H.10—`IEEE1451_Dot2TransducerBlock` class specific restrictions

Operation	Block object's major state value		
	BL_UNINITIALIZED	BL_INACTIVE	BL_ACTIVE
All operations except those in the following row	Local invocation only	Operational	Operational
`SetSTIMUserString`, `SetStatusInterruptMask`	Local invocation only	Operational	Not operational

Copyright © 2000 IEEE. All rights reserved.

H.4.15 State behavior specification

H.4.15.1 Correction engine behavior specification

For the purposes of H.4.15, the term Appropriate Data Corrections shall be interpreted as follows:

— For STIM channels with channel Calibration Key values of `CK_CAL_NONE`, `CK_STIM_CAL_FIXED`, `CK_STIM_CAL_MODIFIABLE`, or `CK_STIM_CAL_SELF`, there shall be no Dot2 Transducer Block applied corrections. In this case the Transducer Block correction engine shall pass the associated channel value without modification in both the `TB_CORRECTED` and `TB_UNCORRECTED` states.

— For STIM channels with channel Calibration Key, values of `CK_CAL_FIXED`, `CK_CAL_MODIFIABLE`, or `CK_CAL_SELF`, there shall be corrections applied by the Dot2 Transducer Block correction engine in the `TB_CORRECTED` state. In the `TB_UNCORRECTED` state, the correction engine shall pass the associated channel value without modification. In `the TB_CORRECTED` state, the data corrections to be applied by the correction engine shall be defined by the methods and coefficients specified in 5.3 of [IEEE 1451.2].

— For STIM channels with channel Calibration Key values of `CK_CAL_CUSTOM`, the correction behavior for this class is outside the scope of this standard. Channels with this value of Calibration Key shall not be represented by Public Transducers. The channel number shall be returned by `GetUnrepresentedChannelNumbers.`

The behavior of the Block's correction engine shall be restricted by the following subclauses.

H.4.15.2 `TB_CORRECTED` state

The values visible at the Public Transducer interfaces shall be subjected to the Appropriate Data Corrections. In this state, all such values available from the Public Transducer's interface shall be in the units and datatypes specified by [IEEE 1451.2] for the corrected data.

H.4.15.3 `TB_UNCORRECTED` state

The values visible at the Public Transducer interfaces shall not be subjected to the Appropriate Data Corrections. In this state, all such values available from the Public Transducer's interface shall be in the units and datatypes specified by [IEEE 1451.2] for the uncorrected data.

 Copyright © 2000 IEEE. All rights reserved.

Annex I

(informative)

Bibliography

This annex provides references useful in understanding the body of this standard. Also included are references to commercial, trade, or governmental, non-balloted, standards controlled by specific organizations. The notation [document-designator] in this standard is a reference to the document in this clause denoted [document-designator].

I.1 General bibliography

[B1] {BIRM}: Birman, Kenneth P., *Building Secure and Reliable Network Applications*. Manning Publications Co., 1996.

[B2] Bryan, L. A. and Bryan, E. A., *Programmable Controllers Selected Applications*, Vol. 1. Chicago, IL: Industrial Text & Video, 1987, pp. 58–61.

[B3] {IDL}: IDL is specified by CORBA/IIOP 2.2 Specification, The Object Management Group (OMG®), February 1998, Framingham, MA. See also the web page at http://www.omg.org/corba/cichpter.htm.

A third party reference discussing the same issues is Inside CORBA, Mowbray, Thomas J., Ruh, William A., Addison-Wesley, 1997. ISBN 0201895404.

[B4] {LIPP}: Lippman, Stanley B., *C++ Primer*, 3rd ed. Reading, MA: Addison-Wesley 1991. ISBN 0-201-54848-8.

[B5] {UML}: Booch, Grady, Rumbaugh, James, and Jacobson, Ivar, *The Unified Modeling Language User Guide*,. Reading, MA: Addison-Wesley, 1998. ISBN 0201571684.

I.2 Reference pertaining to UUID algorithms

[B6] {DCE}: The Open Group, *DCE TODAY.* Englewood Cliffs, NJ: Prentice Hall, 1998. ISBN 1-85912-157-8.

This document discusses the DCE UUID algorithm.

I.3 Reference pertaining to time specification

[B7] {MILLS}: Mills, David L., “On the Chronometry and Metrology of Computer Network Timescales and their Application to the Network Time Protocol,” *ACM Computer Communications Review 21*, vol. 5, pp. 8–17, Oct. 1991.

Copyright © 2000 IEEE. All rights reserved.