

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

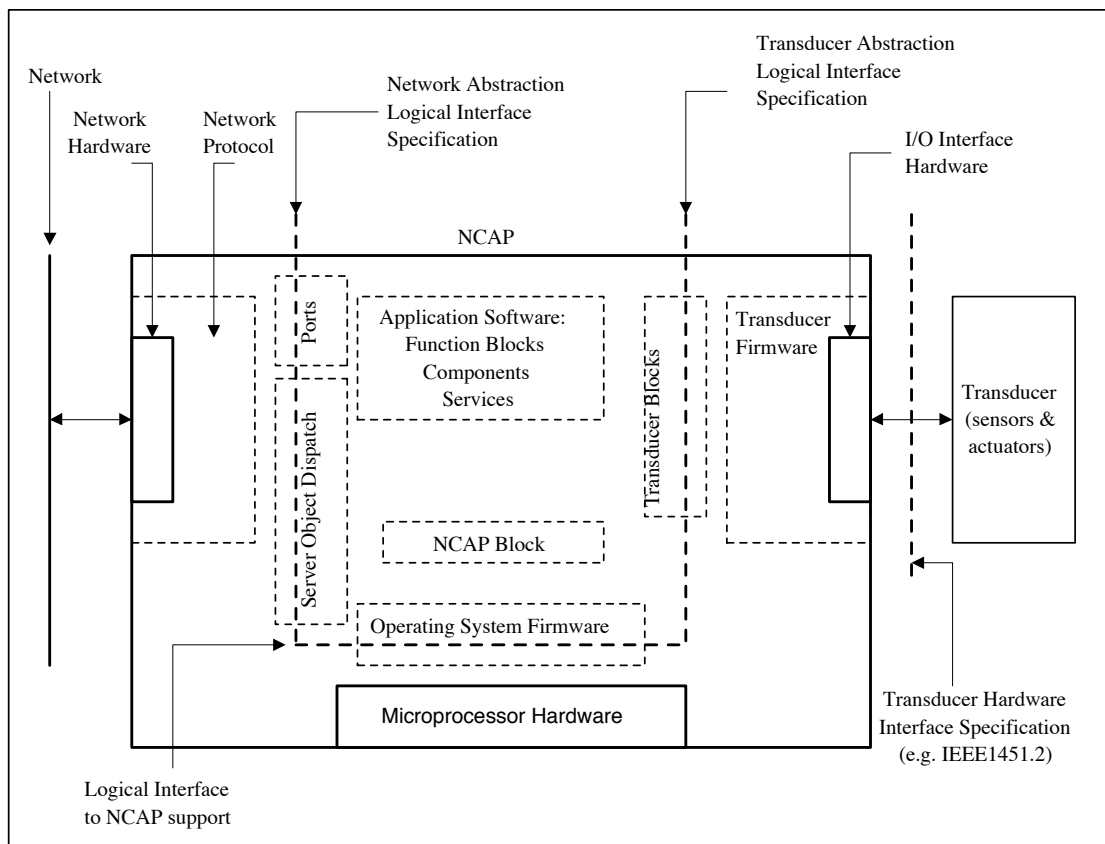


Figure 1—Networked Smart Transducer model

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.

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
Harold Brown

Alec Dara-Abrams
Rick Schneeman

David Wilson
George Woodmansee

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 Nutzerorganisation e.V., Haid-und-Neu Strasse 7, D-76131 Karlsruhe, Germany.

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

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

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

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

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

- 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

¹IEEE 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/>).

²ANSI 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/>).

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

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

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

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

⁶ISO publications are available from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembe, 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/>).

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

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.

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

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

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

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

- 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.

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.

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.

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`, `StringCharCode`, 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.

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.

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.

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.

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.

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).

- 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.

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.”

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,
```

```
/*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.

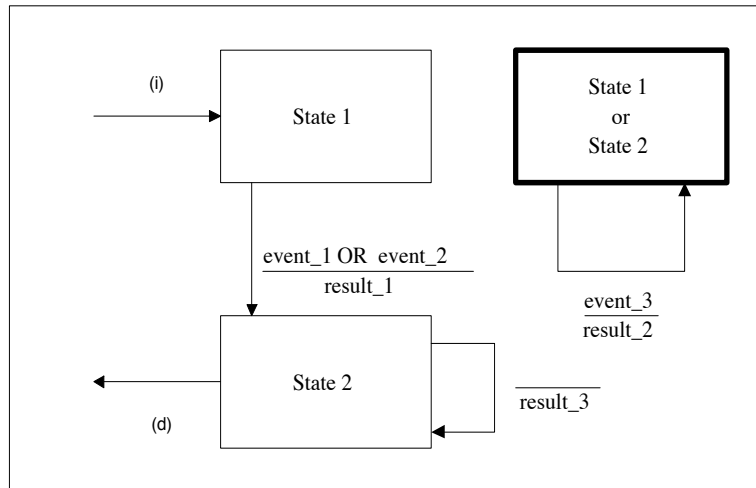


Figure 2—Mealy state transition diagram

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)

- 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

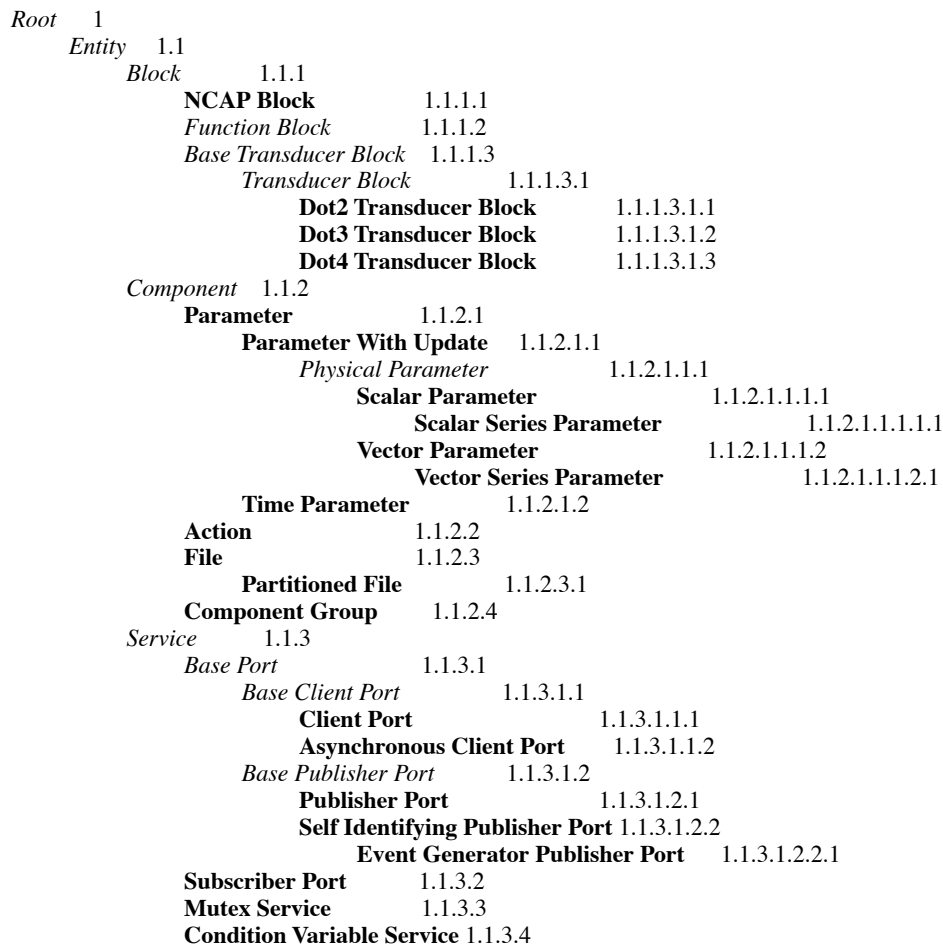


Figure 3—IEEE 1451.1 class hierarchy

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

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.

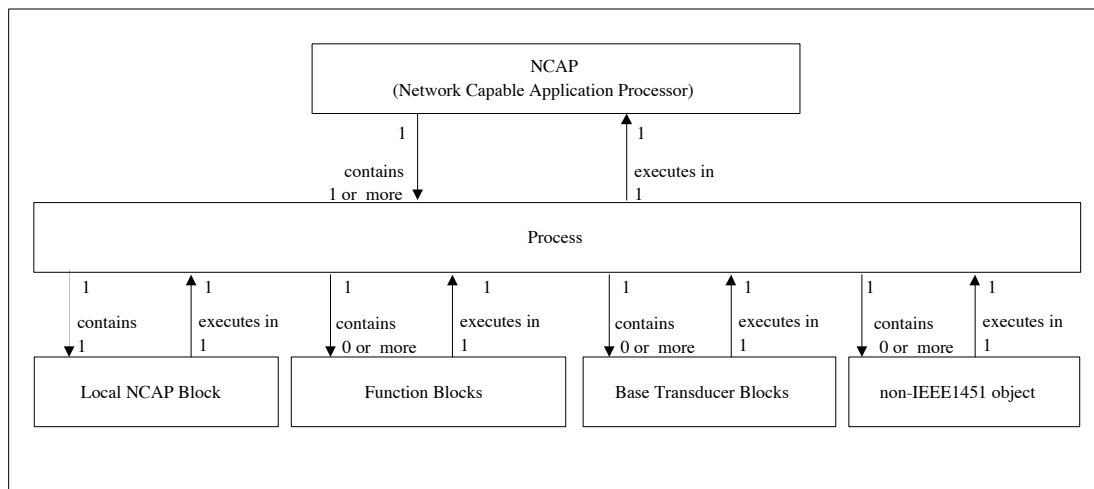


Figure 4—Top-level object relationships

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_NCABlock` 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.

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

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 $\mathcal{B}$ 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 $\mathcal{B}$ 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.

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.

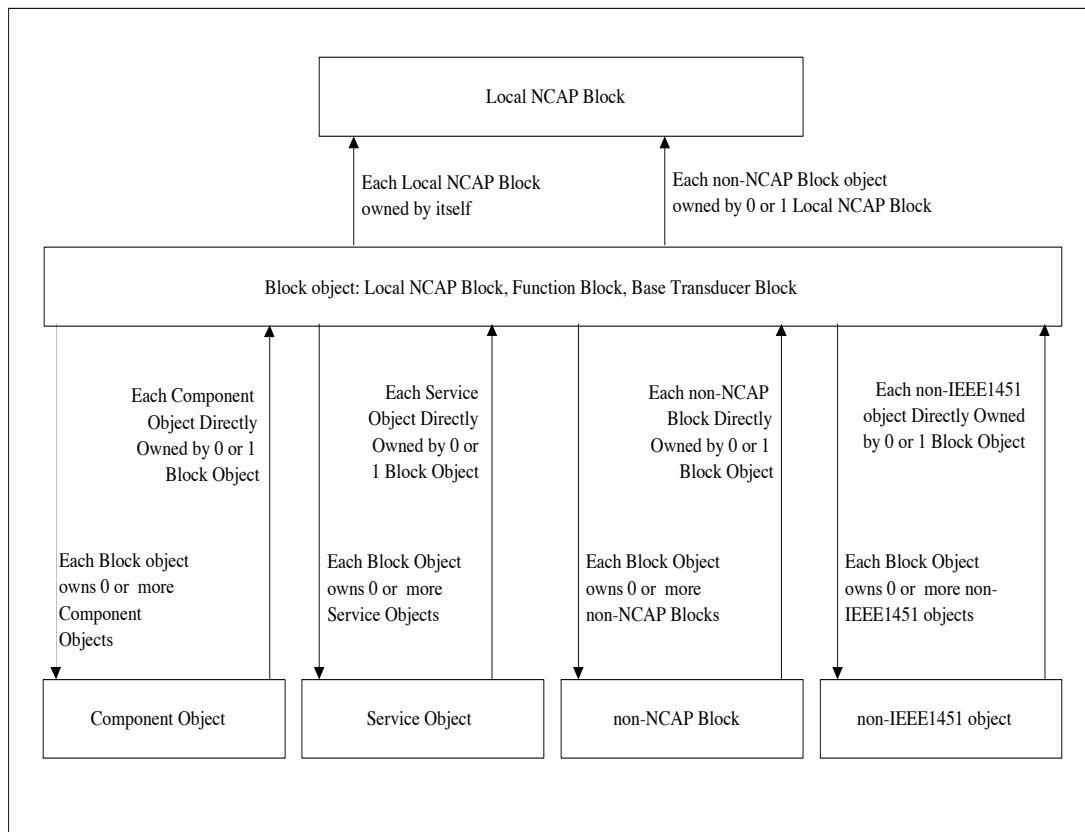


Figure 5—Relationships between Block, Component, Service, and non-IEEE objects

5.1.4 Object identifying properties

An Object’s identifying properties are the Object’s

- Class ID
- Class Name
- Object ID
- Object Tag

- 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

- 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`

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`

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.

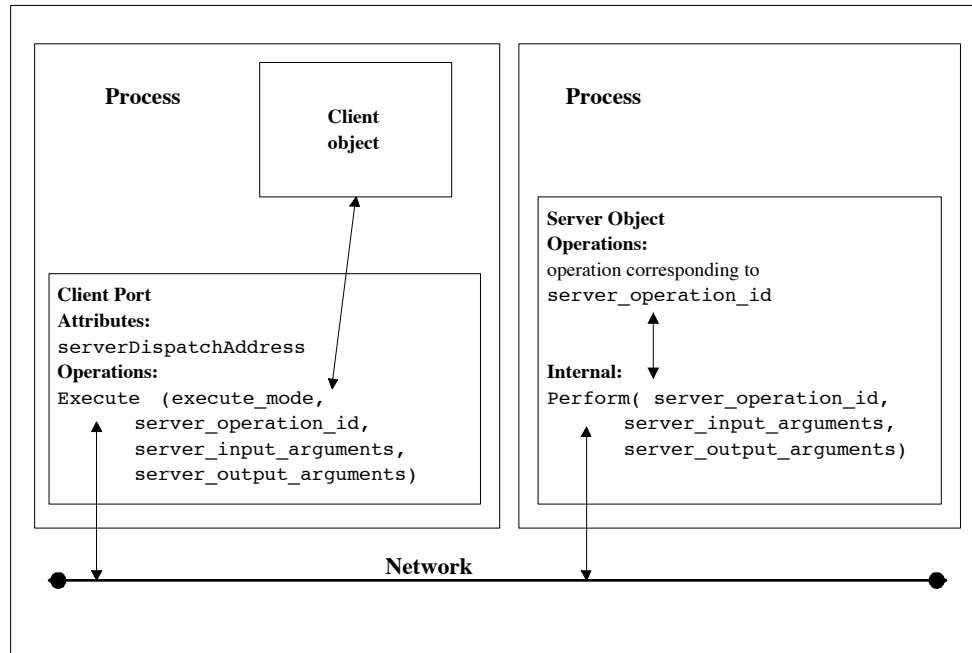


Figure 6—Client-server communication components

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.

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_AynchronousClientPort**. 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.

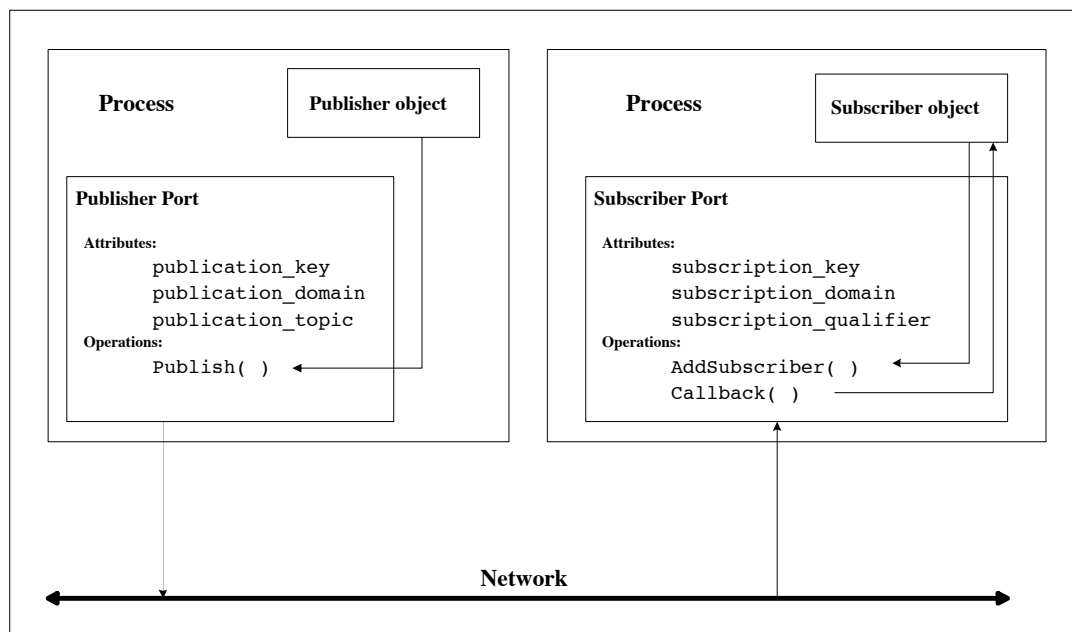


Figure 7—Publish-subscribe communication components

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

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.

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.

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

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

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.

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

- 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[ ];`

```
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

$$0 \leq |\text{nanoseconds}| < 10^9$$

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.

```

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];

```

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.

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;
};

```

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,
```