1451.1 On-The-Wire Format for IP

Agilent Technologies, Inc.

Boeing Company

Revision History					
Rev.	**Reason for Change**	**Pages Affected**	**Approved by**	**Initiated by**	**Rev. Date**
1	Publication	All	Marc Mayer	Jay Warrior	4/11/2002

1451.1 On-The-Wire Format for IP

Table of Contents

1 Scope

This document presents an on-the-wire mapping of the IEEE 1451.1 specification's network communication models for any 1451.1 implementation using the TCP/IP and UDP/IP transport protocols.

This document assumes a familiarity with the IEEE 1451.1 standard.[1]

2 Responsibilities

Responsibility for the update and maintenance of this document lies with its authors.

3 IEEE 1451.1 Using IP

The IEEE 1451.1 standard is a field bus neutral software architecture for distributed measurement and control (DMC) systems. The use of TCP/IP and UDP/IP as field bus transport protocols is gaining popularity among DMC developers.

The IEEE 1451.1 standard requires an inter-processor communication infrastructure in which all data are communicated as the generic 1451.1 datatype ArgumentArray.[2] To achieve this, a sender of a 1451.1 message must *encode* the message's data from their local datatypes into an ArgumentArray and the receiver of the message must *decode* the message's data from an ArgumentArray into local datatypes.

Any implementation of the 1451.1 standard must provide a communication infrastructure[3] that on the sender's side takes a 1451.1 message consisting of header information and an ArgumentArray and *marshal* the message into a linear sequence of bytes, the message's *on-the-wire* format. This byte sequence is dependent on the communication transport used but independent of the communication media. Analogously, the communication infrastructure on a receiver's side must *demarshal* a message's data from its on-the-wire format into an ArgumentArray.

To implement the IEEE 1451.1 communication infrastructure using TCP/IP and/or UPD/IP requires the design of on-the-wire formats for 1451.1 messages on these transports. The IEEE 1451.1 communication infrastructure on IP (more simply, *the Infrastructure*) presented here uses TCP/IP for client-server communications and UDP/IP multicast for publisher-subscriber communications.

In the balance of this document, the IEEE 1451.1 standard will be denoted more simply by *the Standard* (upper case 'S').

[1] IEEE 1451.1-1999 IEEE Standard for a Smart Transducer Interface for Sensors and Actuators--Network Capable Application Process (NCAP) Information Model

[2] See Section 6 in the IEEE 1451.1 standard document.

[3] See Section 5.3 in the IEEE 1451.1 standard document

3.1 On-The-Wire Format Tradeoff

The IEEE 1451.1 on-the-wire formats for IP have been designed to favor ease of marshaling and demarshaling over minimizing the number of on-the-wire bytes. This is a reasonable tradeoff given that almost all 1451.1 messages will be small, for example less than the size of an Ethernet frame. For those cases where large amounts of data must be transmitted in 1451.1 messages, the on-the-wire format admits the use of a message payload compression algorithm as an option.

4 On-the-Wire Format for IEEE 1451.1 Datatypes

The IEEE 1451.1 Standard is presented as a set of datatypes and object classes. The Standard defines fifty-six datatypes. These datatypes, and only these datatypes, are used in the signatures of the operations of the 1451.1-specified object classes.

Associated with each 1451.1 datatype is a unique member of the TypeCode enumeration. A definition of this enumeration is given in Appendix A.

In this section, the on-the-wire format for each of the fifty-six 1451.1 datatypes is defined.

4.1 XDR

The IEEE 1451.1 on-the-wire format for data is based on the XDR External Data Representation Standard[4]. XDR provides a language for describing data formats. XDR is the data format standard used by ONC RPC[5].

A basic assumption underlying XDR is that bytes (octets) are portable. That is, devices must marshal bytes onto any communications transport and media in such a way that other devices can demarshal the bytes without loss of meaning.

For any datum, its XDR representation is always a multiple of 4 bytes. Padding is used for datums that are not of themselves a multiple of 4 bytes.

The bytes in an XDR format are numbered from 0 through (n–1) with $n \equiv 0 \pmod 4$. The bytes are read from or written to a byte stream such that byte m always precedes byte m+1. For those datums that represent integral values, the XDR representation is big-endian. That is, byte 0 is the most significant byte (MSB).

4.2 Notation for On-the-Wire Formats

In this document, the following notation is used to describe the on-the-wire format of a 1451.1 datatype:

$|h_0h_1|h_2h_3|h_4h_5|h_6h_7|\ldots$,

where:

[4] RFC 1014.
[5] RFC 1831

- Each h_i is a hexadecimal digit
- Each byte h_ih_{i+1}, $i \equiv 0 \pmod 2$ is in its in-core form, high order bit leftmost.
- The bytes are transported left-to-right. That is, h_0h_1 goes "on-the-wire" first.
- The byte order is big endian for all multi-byte, integral types, that is, h_0h_1 is the high order byte.

The above notation is equivalent to the traditional notation:

```
+--------+--------+--------+--------+ ...
|  byte 0  |  byte 1  |  byte 2  |  byte 3  | ...
+--------+--------+--------+--------+ ...
```

4.2.1 IEEE 1451.1 Uniform Datatypes

The on-the-wire formats for the 1451.1 primitive datatypes and arrays of the primitive datatypes are derived from the formats for corresponding XDR datatypes.

The total number of bytes used to represent any 1451.1 datum in its on-the-wire format is a multiple of 4.[6]

4.2.2 Primitive Datatypes

The following table gives:

- The correspondence between each primitive 1451.1 datatype and an XDR datatype
- The 1451.1 datatype's on-the-wire format.

IEEE 1451.1 Type	XDR Type	On the Wire
Boolean	opaque[4]	|00|00|00|00| for FALSE, |00|00|00|01| for TRUE
Enumeration Member	opaque[4]	|00|00|00|h_0h_1|, where the member's value is h_0h_1[7]. Here, an enumeration member is one of the values of one of the enumerations defined in the Standard.
Float32	float	f, where f is sign+exponent+fractional in ANSI/IEEE 754-1985 standard normalized 4-byte float representation. sign is 1-bit, exponent is 8-bits biased by 127, fractional is the fractional part of the mantissa and is 23-bits. All numbers are base 2. NaN is not allowed[8]

[6] See the XDR specification, RFC 1014, for a discussion of why the choice of 4.

[7] An enumeration member in 1451.1 has typecode 0x33. That is, the same typecode as the typecode of a UInteger8. This is because for all 1451.1-specified enumerations, each network communicated enumeration member has local representation a UInteger8. Any casting of a communicated enumeration member in a 1451.1 application system is the responsibility of the application.

[8] XDR excludes NaN as a value for a floating point number as it is not portable across platforms.

Float64	double	d, where d is sign+exponent+fractional in ANSI/IEEE 754-1985 standard normalized 8-byte float representation. Here exponent is 11-bits biased by 1023 and fractional is 52-bits. NaN is not allowed
Integer8	opaque[4]	|00|00|h_0h_1|, where the integer's value is $0xh_0h_1$ in twos complement
Integer16	int	|00|00|h_0h_1|h_2h_3|, where the integer's value is $0xh_0h_1h_2h_3$ in twos complement
Integer32	int	|h_0h_1|h_2h_3|h_4h_5|h_6h_7|, where the integer's value is $0xh_0h_1h_2h_3h_4h_5h_6h_7$ in twos complement
Integer64	hyper	|h_0h_1|h_2h_3|h_4h|h_6h_7|**…**$h_{14}h_{15}$|, where the integer's value is $0xh_0h_1h_2h_3h_4h_5h_6h_7$**…** $h_{14}h_{15}$ in twos complement
Octet	opaque[4]	|00|00|00|h_0h_1|, where the octet's value is $0xh_0h_1$
UInteger8	opaque[4]	|00|00|00|h_0h_1|, where the integer's value is $0xh_0h_1$
UInteger16	unsigned int	|00|00|h_0h_1|h_2h_3|, where the integer's value is $0xh_0h_1h_2h_3$
UInteger32	unsigned int	|h_0h_1|h_2h_3|h_4h_5|h_6h_7|, where the integer's value is $0xh_0h_1h_2h_3h_4h_5h_6h_7$
UInteger64	unsigned hyper	|h_0h_1|h_2h_3|h_4h|h_6h_7|**…** |$h_{14}h_{15}$|, where the integer's value is $0xh_0h_1h_2h_3h_4h_5h_6h_7$**…** $h_{14}h_{15}$

Table 1. On-the-Wire Format for IEEE 1451.1 Primitive Datatypes

4.2.3 Arrays of Primitive Datatypes

The following table gives the correspondence between 1451.1 arrays of primitive types and XDR datatypes and the 1451.1 on-the-wire format.

As is usual, an array index indicates a positive offset. Hence, all arrays are put on-the-wire with 0^{th} (leftmost) element first

For an array of elements where the individual elements are not themselves represented as a multiple of 4 bytes the on-the-wire format of the array has a terminal (right-most) sub-sequence of 0, 1, 2 or 3 bytes, each with value 0x00. This sub-sequence of bytes is denoted by *PAD*. The length of PAD is chosen such that the entire on-the-wire format of the array has number of bytes a multiple of 4.

All integral values in the table are in hexadecimal notation.

IEEE 1451.1 Type	**XDR Type**	**On the Wire**
BooleanArray	bool<>[9]	|n_0n_1|n_2n_3|n_4n_5|n_6n_7|h_0h_1|h_2h_3|**...**h_k|PAD. Here $0xn_0n_1n_2n_3n_4n_5$ n_6n_7 gives the number of elements in the array represented as a 32-bit unsigned integer and each h_ih_{I+1}, $i \equiv 0 \pmod 2$, is one of 0x00 for FALSE or 0x01 for TRUE

[9] The notation "datatype<>" denotes an XDR variable length (counted) array.

Float32Array	float<>	$\|n_0n_1\|n_2n_3\|n_4n_5\|n_6n_7\|h_0h_1\|h_2h_3\|...h_k\|$. Here $0xn_0n_1n_2n_3n_4n_5n_6n_7$ gives the number of elements in the array represented as a 32-bit unsigned integer and each sequence of 4 bytes h_i **...** h_{i+7} with $i \equiv 0 \pmod 8$ represents a float in IEEE standard notation. NaN is not allowed as an array element
Float64Array	double<>	$\|n_0n_1\|n_2n_3\|n_4n_5\|n_6n_7\ h_0h_1\|h_2h_3\|...h_k\|$. Here $0xn_0n_1n_2n_3n_4n_5n_6n_7$ gives the number of elements in the array represented as a 32-bit unsigned integer and each sequence of 8 bytes h_i **...** h_{k+i} with $i \equiv 0 \pmod{16}$ represents a double in IEEE standard notation. NaN is not allowed as an array element
Integer8Array	opaque<>	$\|n_0n_1\|n_2n_3\|n_4n_5\|n_6n_7\|h_0h_1\|h_2h_3\|...h_k\|$PAD. Here $0xn_0n_1n_2n_3n_4n_5n_6n_7$ gives the number of elements in the array represented as a 32-bit unsigned integer and each h_ih_{i+1} with $i \equiv 0 \pmod 2$ represents an Integer8
Integer16Array	opaque<>	$\|n_0n_1\|n_2n_3\|n_4n_5\|n_6n_7\ \|h_0h_1\|h_2h_3\|$ **...**$h_k\|$PAD. Here $0xn_0n_1n_2n_3n_4n_5n_6n_7$ gives the number of elements in the array represented as a 32-bit unsigned integer and each $h_ih_{i+1}h_{i+2}h_{i+3}$, with $i \equiv 0 \pmod 4$ represents the value of an Integer16 array element

Integer32Array	int<>	$\|n_0n_1\|n_2n_3\|n_4n_5\|n_6n_7\|h_0h_1\|h_2h_3\|...h_k\|$. Here $0xn_0n_1n_2n_3n_4n_5n_6n_7$ gives the number of elements in the array represented as a 32-bit unsigned integer and each $h_ih_{i+1}h_{i+2}h_{i+3}$ $h_{i+4}h_{i+5}h_{i+6}$ h_{i+7}, with $i \equiv 0$ (mod8) represents an Integer32
Integer64Array	hyper<>	$\|n_0n_1\|n_2n_3\|n_4n_5\|n_6n_7\|h_0h_1\|h_2h_3\|...h_k\|$. Here $0xn_0n_1n_2n_3n_4n_5n_6n_7$ gives the number of elements in the array represented by an unsigned, 32-bit integer and each $h_ih_{i+1}h_{i+2}h_{i+3}$ $h_{i+4}h_{i+5}h_{i+6}$ h_{i+7} **...**h_{i+14} h_{i+15}, with $i \equiv 0$ (mod 16) represents an Integer64
OctetArray	opaque<>	$\|n_0n_1\|n_2n_3\|n_4n_5\|n_6n_7\|h_0h_1\|h_2h_3\|...h_k\|$PAD. Here $0xn_0n_1n_2n_3n_4n_5n_6n_7$ gives the number of elements in the array represented as a 32-bit unsigned integer and each h_ih_{i+1} with $i \equiv 0$ (mod 2) represents an Octet.
UInteger8Array	opaque<>	$\|n_0n_1\|n_2n_3\|n_4n_5\|n_6n_7\|h_0h_1\|h_2h_3\|...h_k\|$PAD. Here $0xn_0n_1n_2n_3n_4n_5n_6n_7$ gives the number of elements in the array represented as a 32-bit unsigned integer and each h_ih_{i+1} with $i \equiv 0$ (mod 2) represents a UInteger8.
UInteger16Array	opaque<>	$\|n_0n_1\|n_2n_3\|n_4n_5\|n_6n_7\|h_0h_1\|h_2h_3\|...h_k\|$PAD. Here $0xn_0n_1n_2n_3n_4n_5n_6n_7$ gives the number of elements in the array represented as a 32-bit unsigned integer and each $h_ih_{i+1}h_{i+2}h_{i+3}$, with $i \equiv 0$ (mod4) represents the value of a UInteger16 array element
UInteger32Array	unsigned int<>	$\|n_0n_1\|n_2n_3\|n_4n_5\|n_6n_7\|h_0h_1\|h_2h_3\|...h_k\|$. Here $0xn_0n_1n_2n_3n_4n_5n_6n_7$ gives the number of elements in the array represented as a 32-bit unsigned integer and each $h_ih_{i+1}h_{i+2}h_{i+3}$ $h_{i+4}h_{i+5}h_{i+6}$ h_{i+7}, with $i \equiv 0$ (mod 8) represents a UInteger32
UInteger64Array	unsigned hyper<>	$\|n_0n_1\|n_2n_3\|n_4n_5\|n_6n_7\|h_0h_1\|h_2h_3\|...h_k\|$. Here $0xn_0n_1n_2n_3n_4n_5n_6n_7$ gives the number of elements in the array represented as a 32-bit unsigned integer and each $h_ih_{i+1}h_{i+2}h_{i+3}$ $h_{i+4}h_{i+5}h_{i+6}$ h_{i+7}**...**h_{i+14} h_{i+15}, with $i \equiv 0$ (mod 16) represents a UInteger64

Table 2. On-the-Wire Format for IEEE 1451.1 Arrays of Primitives

4.2.4 Units, UnitsArray and PubSubDomain

The 1451.1 datatypes Units and PubSubDomain are specified as fixed length OctetArrays. Their on the wire formats are given in Table 3.

IEEE 1451.1 Type	XDR Type	On the Wire
Units	opaque[10][10]	$\|h_0h_1\|h_2h_3\|\ldots\|h_{18}h_{19}\|00\|00\|$.Here each h_ih_{i+1}, $i \equiv 0 \pmod 2$, is is an octet member of the Unit's value
UnitsArray	opaque<>	$\|n_0n_1\|n_2n_3\|n_4n_5\|n_6n_7\|00\|00\|h_0h_1\|h_2h_3\|\ldots$ $\|h_{18}h_{19}\|00\|00\|h_{20}h_{21}\|\ldots h_k\|$. Here $0xn_0n_1n_2n_3n_4n_5n_6n_7$ gives the number of elements in the array represented as a 32-bit unsigned integer and each $h_ih_{i+1}\ldots h_{i+19}0000$, $i \equiv 0 \pmod{20}$, is a Units value
PubSubDomain	opaque[8]	$\|h_0h_1\|h_2h_3\|\ldots\|h_{14}h_{15}\|$. Here each h_ih_{i+1}, $i \equiv 0 \pmod 2$ is an octet member of the PubSubDomain's value

Table 3. On-the-Wire Format for Units and PubSubDomain

4.2.5 String and StringArray

The on-the-wire formats for a 1451.1 String and a StringArray are given in Table 4.

IEEE 1451.1 Type	XDR Type	On the Wire
String	opaque<>	$\|00\|s_1s_2\|c_1c_2\|l_1l_2\|n_0n_1\|n_2n_3\|n_4n_5\|n_6n_7\|h_0h_1\|h_2h_3\|\ldots\|h_{k-1}h_k\|$PAD
StringArray	opaque<>	$\|n_0n_1\|n_2n_3\|n_4n_5\|n_6n_7\|s_0\ldots s_t$

Table 4. On-the-Wire Format for String and StringArray

In the on-the-wire format for a String, $0xs_1s_2$ is an enumeration value from the enumeration StringCharacterSet, $0xc_1c_2$ is an enumeration value from StringCharacterCode, $0xl_1l_2$ is an enumeration value from StringLanguage and $0xn_0n_1n_2n_3n_4n_5n_6n_7$ gives the number of bytes of string data represented by an unsigned, 32-bit integer. The bytes $0xh_ih_{i+1}$ with $i \equiv 0 \bmod (2)$ give the String's data content where the number of bytes used to represent each character in the string is dependent on the value of the String's StringCharacterCode.

In the on-the-wire format for a StringArray, $0xn_0n_1n_2n_3n_4n_5$ gives the number of Strings in the StringArray represented by an unsigned, 32bit integer and s_i is the i^{th} String element in the array represented as an on-the-wire String.

[10] The notation "datatype[n]" denotes an XDR array of fixed length n.

4.2.6 Typedefed Datatypes

Each of the 1451.1 datatypes that is defined in the Standard as a typedef of one of the 1451.1 datatypes whose on-the-wire format is given in Section 4.2.2 or Section 4.3.3, above, uses the same on-the-wire format as the latter 1451.1 datatype.

Table 5 gives the datatype to which each of these typedefed types corresponds.

IEEE 1451.1 Type	typedef
BitSequence	OctetArray
ClassID	OctetArray
ClientServerReturnCode	UInteger32
ObjectID	OctetArray
ObjectTag	OctetArray
OpReturnCode	UInteger16
PublicationTopic	OctetArray
SubscriptionQualifier	OctetArray

Table 5. On-the-Wire Correspondence for IEEE 1451.1 Typedefed Datatypes

4.2.7 BitSequence

The 1451.1 datatype BitSequence is declared via a typedef to an OctetArray and, hence, has an OctetArray's on-the-wire format. However, the BitSequence and OctetArray datatypes have quite different syntactic interpretations. A BitSequence is interpreted on a bit-by-bit basis from left-to-right. That is, an OctetArray that represents a BitSequence is syntactically interpreted as a sequence of 8*k bits. Here, k is the number of Octets in the representing OctetArray. The ordering of the bits in the sequence is as follows:

1. Any bit in the k^{th} Octet precedes any bit in the $(k+m)^{th}$ Octet, $m > 0$.

2. The bits in an Octet are numbered 0 through 7 in a left-to-right order (see Section 4.2). Within an Octet, the i^{th} bit precedes the j^{th} bit for $i < j$.

4.2.8 PublicationTopic and SubscriptionQualifier Wildcards

Publication topics and subscription qualifiers may contain the wildcard octets WC_ANY_OCTET and WC_ANY_OCTET_ARRAY. The octet values associated with these wild cards are **0xfe** and **0xff**, respectively (the Standard, Section 12.3.4). The on-the-wire forms of the wildcards are these values marshaled opaquely.

4.2.9 BitSequenceArray and ObjectTagArray

The datatypes BitSequenceArray and ObjectTagArray are both defined in the Standard as a typedef to an array of OctetArrays. An array of OctetArrays is never directly communicated on-the-wire using its own identity. Rather, it occurs on-the-wire only as an instance of BitSequenceArray or ObjectTagArray.

The on-the-wire format for a BitSequenceArray is:

$$| n_0n_1|n_2n_3|n_4n_5|n_6n_7|b_0...b_k$$

where each b_i is a BitSequence represented on-the-wire as described in Section 4.4.6 and $0xn_0n_1n_2n_3n_4n_5n_6n_7$ gives the number of BitSequences in the array represented as a 32-bit unsigned integer.

Analogously, the on-the-wire format for an ObjectTagArray is

$$| n_0n_1|n_2n_3|n_4n_5|n_6n_7|t_0...t_k$$

with each t_i an ObjectTag represented on-the-wire as described in Section 4.2.6.

4.3 IEEE 1451.1 Heterogeneous Datatypes

Of the fifty-six 1451.1 datatypes, thirty-one are defined in the Standard as structures, unions or arrays of structures or unions. The on-the-wire format for each of these thirty-one heterogeneous datatypes is described in this section. The on-the-wire formats are presented explicitly (as opposed to algorithmically).

4.3.1 The 1451.1 on IP Object Dispatch Address Structure

The ObjectDispatchAddress of a Network Visible object (see the Standard, Sections 3 and 6) is, in general, both network and 1451.1 implementation dependent. For an implementation of the Standard using TCP/IP as a transport protocol, the ObjectDispatchAddress of a Network Visible object (see the Standard, Sections 3 and 6) is defined as an instance of the following structure:

```
struct ObjectDispatchAddress
{
        UInteger32      ipAddress;
        UInteger32      portNumber;
        UInteger32      objectIdentifier;
};
```

An object's dispatch address is used in 1451.1 client-server communications. These communications in this on-the-wire mapping for IP use TCP/IP. The value of the portNumber field in a dispatch address always refers to a TCP port.

How the value of objectIdentifier is assigned to a Network Visible object is implementation dependent. However, it must be assigned in such a manner that for any two distinct Network Visible objects in the same NCAP process space, their objectIdentifiers are also distinct.

4.3.2 Notation

Several special notations are used to describe the on-the-wire formats of the 1451.1 heterogeneous datatypes.

4.3.2.1 Structure Fields

Each 1451.1 datatype that is defined as a structure can be presented as the (ordered) sequence of the field names used in defining the datatype in the Standard. If, for a given field name, its associated datatype is itself a structure, that field name can be presented as a sequence of field names. This process of presenting a 1451.1 structure datatype as a sequence of field names can be applied recursively until the structure datatype is decomposed into a hierarchical sequence of field names each of whose associated datatype is a datatype defined elsewhere in this document in a non-circular manner. This hierarchy of field names forms an ordered forest.

The notation used to describe a 1451.1 structure datatype as an ordered forest of field names is:

field name$_1$
    field name$_{11}$
        field name$_{111}$
        field name$_{112}$
        …
    field name$_{12}$
    …
field name$_2$
…
field name$_n$
    field name$_{n1}$
    …

Here:

- Each leftmost field name is the root of a tree in the forest
- Indentation indicates that a field name is a branch or a leaf of the field name immediately above it
- The order of the field names may differ from the order given by the datatype's definition in the Standard. The field name order for a structure's on-the-wire format is chosen to minimize the need for byte padding.

4.3.2.2 On-The-Wire Formats

The notation:

<datatype>,

where **datatype** is a 1451.1 datatype, is used to denote the syntactic equivalent of the on-the-wire format of a generic datum whose type is **datatype**.

For example,

<Octet> = |00|00|00|$\underline{h}_0\underline{h}_1$|,

where the octet's value is 0x$\underline{h_0}\underline{h_1}$, and

<Integer16Array> = |$\underline{n_0}\underline{n_1}$|$\underline{n_2}\underline{n_3}$|$\underline{n_4}\underline{n_5}$|$\underline{n_6}\underline{n_7}$ |$\underline{h_0}\underline{h_1}$|$\underline{h_2}\underline{h_3}$| ...$\underline{h_k}$|**PAD**,

where, 0x$\underline{n_0}\underline{n_1}\underline{n_2}\underline{n_3}\underline{n_4}\underline{n_5}\underline{n_6}\underline{n_7}$ gives the number of elements in the array represented as a 32-bit unsigned integer and each $\underline{h_i}\underline{h_{i+1}}\underline{h_{i+2}}\underline{h_{i+3}}$, with $i \equiv 0 \pmod 4$ represents the value of an Integer16 array element.

The notation used to denote the on-the-wire format of a specific datum, **datum**, whose type is a 1451.1 datatype is:

<datum>.

For example, if **datum** is a UInteger16 of value 23076, then

<datum> = |00|00|5a|24|.

With the exception of **<Argument>** and **<ArgumentArray**>, the <**datatype**> and <**datum**> notations are used only when the on-the-wire format for **datatype** or for **datum** has been defined earlier in this document. The on-the-wire formats for Argument and ArgumentArray are discussed in Section 4.3.7.

4.3.3 Structures

The forest of field names used to describe the on-the-wire format for each of the 1451.1 structure datatypes is listed in the "Fields" column of the following table. The on-the-wire format for the value of each of the leaf field names is given in the "On the Wire" column of the table on the same line as the field name. The sequence of datums corresponding to the sequence of leaf field name values of a 1451.1 structure datatype is marshaled from top to bottom.

IEEE 1451.1 Type	Fields	On the Wire
ObjectDispatchAddress	ipAddress portNumber objectIdentifier	<UInteger32> <UInteger32> <UInteger32>
TimeRepresentation	seconds nanoseconds	<UInteger32> <Integer32>

TimeInterval	startTime
	  seconds
	  nanoseconds
	deltaTime
	  seconds
	  nanoseconds
		
		<UInteger32>
		<Integer32>
		
		<UInteger32>
		<Integer32>

ObjectProperties	objectDispatchAddress objectTag owningBlockObjectTag objectName blockCookie	<ObjectDispatchAddress> <OctetArray> <OctetArray> <String> <UInteger32>
ClientPortProperties	clientPortDispatchAddress clientPortObjectTag clientPortName serverDispatchAddress serverObjectTag	<ObjectDispatchAddress> <OctetArray> <String> <ObjectDispatchAddress> <OctetArray>
PublisherInformation	publicationID portObjectTag publicationChangeID	<UInteger32> <OctetArray> <UInteger32>
Uncertainty	interpretation uncertaintyValue coverageFactor customFactor	<Enumeration Member> <Float32> <Float32> <Float32>

Table 6. IEEE 1451.1 On-the-Wire Formats for IEEE 1451.1 Structures

4.3.4 Arrays of Structures

The on-the-wire format for an array of one of the 1451.1 structure types listed in Table 6 is analogous to the on-the-wire format for an array of a 1451.1 primitive datatype. The arrays of the structures in Table 6 that are defined by the Standard are:

ClientPortPropertiesArray
ObjectPropertiesArray
TimeIntervalArray
TimeRepresentationArray
UncertaintyArray

The on-the-wire format for each of these array types consists of the sequence of bytes:

<n>
<on-the-wire form of the value of array element$_0$>
<on-the-wire form of the value of array element$_1$>
• • •
<on-the-wire form of the value of array element$_{n-1}$>

Here, n is the number of elements in the array represented as a UInteger32.

4.3.5 Physical Parameter Types

A 1451.1 structure, which represents the data or metadata, associated with a PhysicalParameter is marshaled onto the wire as the value of the union

PhysicalParameterData or PhysicalParameterMetadata, respectively. These unions have as discriminate a member of the enumeration PhysicalParameterType.

The PhysicalParameterType enumeration is declared in the Standard as

```
enum PhysicalParameterType
{
        PP_SCALAR_ANALOG = 0,
        PP_SCALAR_DISCRETE = 1,
        PP_SCALAR_DIGITAL = 2,
        PP_SCALAR_ANALOG_SERIES = 3,
        PP_SCALAR_DISCRETE_SERIES = 4,
        PP_SCALAR_DIGITAL_SERIES = 5,
        PP_VECTOR_ANALOG = 6,
        PP_VECTOR_DISCRETE = 7,
        PP_VECTOR_DIGITAL = 8,
        PP_VECTOR_ANALOG_SERIES = 9,
        PP_VECTOR_DISCRETE_SERIES = 10,
        PP_VECTOR_DIGITAL_SERIES = 11,
};
```

The on-the-wire format of a PhysicalParameterData union or a PhysicalParameterMetaData union consists of the union's discriminate represented as a UInteger32 followed by the on-the-wire form of the structure which is the union's value.

The on-the-wire format for the PhysicalParameter-related data and metadata structures is given in Table 7.

IEEE 1451.1 Type	Fields	On the Wire
PhysicalParameterSingletonData		w, w = 0,1,2,6,7 or 8 as a UInteger32
	timestamp	<TimeRepresentation>
	value	<ArgumentArray>
PhysicalParameterSeriesData		w, w = 3,4,5,9,10 or 11 as a UInteger32
	timestamp	<TimeRepresentation>
	abscissaIncrement	<Argument>
	abscissaOrigin	<Argument>
	value	<ArgumentArray>
ScalarAnalogMetadata		0x00000000
	parameterInterpretation	<Enumeration Member>
	buffering	<Enumeration Member>
	datatype	<TypeCode>
	units	<Units>
	parameterName	<String>
	upperLimit	<Argument>
	lowerLimit	<Argument>
	uncertainty	<Uncertainty>
ScalarDigitalMetadata		0x00000002
	parameterInterpretation	<Enumeration Member>
	buffering	<Enumeration Member>
	datatype	<TypeCode>
	rightJustifiedFlag	<Boolean>
	units	<Units>
	parameterName	<String>
	numberOctets	<UInteger16>
	numberOfSignificantBits	<UInteger16>

ScalarDiscreteMetadata		0x00000001
	parameterInterpretation	<Enumeration Member>
	buffering	<Enumeration Member>
	datatype	<TypeCode>
	units	<Units>
	parameterName	<String>
	upperLimit	<Argument>
	lowerLimit	<Argument>
ScalarAnalogSeriesMetadata		0x00000003
	parameterInterpretation	<Enumeration Member>
	buffering	<Enumeration Member>
	datatype	<TypeCode>
	units	<Units>
	abscissaUnits	<Units>
	parameterName	<String>
	upperLimit	<Argument>
	lowerLimit	<Argument>
	abscissaIncrement	<Argument>
	abscissaOrigin	<Argument>
	abscissaIncrementUncertainty	<Uncertainty>
	abscissaOriginUncertainty	<Uncertainty>
	uncertainty	<Uncertainty>

ScalarDigitalSeriesMetadata		0x00000005
	rightJustifiedFlag	<Boolean>
	parameterInterpretation	<Enumeration Member>
	buffering	<Enumeration Member>
	datatype	<TypeCode>
	units	<Units>
	abscissaUnits	<Units>
	parameterName	<String>
	abscissaIncrement	<Argument>
	abscissaOrigin	<Argument>
	abscissaIncrementUncertainty	<Uncertainty>
	abscissaOriginUncertainty	<Uncertainty>
	numberOctets	<UInteger16>
	numberOfSignificantBits	<UInteger16>
ScalarDiscreteSeriesMetadata		0x00000004
	parameterInterpretation	<Enumeration Member>
	buffering	<Enumeration Member>
	datatype	<TypeCode>
	units	<Units>
	abscissaUnits	<Units>
	parameterName	<String>
	upperLimit	<Argument>
	lowerLimit	<Argument>
	abscissaIncrement	<Argument>
	abscissaOrigin	<Argument>
	abscissaIncrementUncertainty	<Uncertainty>
	abscissaOriginUncertainty	<Uncertainty>

VectorAnalogMetadata		0x00000006
	parameterInterpretation	<Enumeration Member>
	buffering	<Enumeration Member>
	coordinateSystem	<Enumeration Member>
	dimension	<UInteger16>
	datatype	<Array of TypeCodes carried as a UInteger8Array>
	units	<UnitsArray>
	parameterName	<String>
	upperLimit	<ArgumentArray>
	lowerLimit	<ArgumentArray>
	uncertainty	<UncertaintyArray>
VectorDigitalMetadata		0x00000008
	parameterInterpretation	<Enumeration Member>
	buffering	<Enumeration Member>
	coordinateSystem	<Enumeration Member>
	dimension	<UInteger16>
	datatype	<Array of TypeCodes carried as a UInteger8Array>
	units	<UnitsArray>
	parameterName	<String>
	numberOctets	<UInteger16>
	numberOfSignificantBits	<UInteger16>
	rightJustifiedFlag	<BooleanArray>

VectorDiscreteMetadata		0x00000007
	parameterInterpretation	<Enumeration Member>
	buffering	<Enumeration Member>
	coordinateSystem	<Enumeration Member>
	dimension	<UInteger16>
	datatype	<Array of TypeCodes carried as a UInteger8Array>
	units	<UnitsArray>
	parameterName	<String>
	upperLimit	<ArgumentArray>
	lowerLimit	<ArgumentArray>
VectorAnalogSeriesMetadata		0x00000009
	parameterInterpretation	<Enumeration Member>
	buffering	<Enumeration Member>
	coordinateSystem	<Enumeration Member>
	dimension	<UInteger16>
	datatype	<Array of TypeCodes carried as a UInteger8Array>
	units	<UnitsArray>
	parameterName	<String>
	abscissaUnits	<Units>
	abscissaIncrement	<Argument>
	abscissaOrigin	<Argument>
	abscissaIncrementUncertainty	<Uncertainty>
	abscissaOriginUncertainty	<Uncertainty>
	upperLimit	<ArgumentArray>
	lowerLimit	<ArgumentArray>
	uncertainty	<UncertaintyArray>

VectorDigitalSeriesMetadata		0x0000000b
	parameterInterpretation	<Enumeration Member>
	buffering	<Enumeration Member>
	coordinateSystem	<Enumeration Member>
	dimension	<UInteger16>
	datatype	<Array of TypeCodes carried as a UInteger8Array>
	units	<UnitsArray>
	parameterName	<String>
	abscissaUnits	<Units>
	abscissaIncrement	<Argument>
	abscissaOrigin	<Argument>
	abscissaIncrementUncertainty	<Uncertainty>
	abscissaOriginUncertainty	<Uncertainty>
	numberOctets	<Uinteger16Array>
	numberOfSignificantBits	<UInteger16Array>
	rightJustifiedFlag	<BooleanArray>

VectorDiscreteSeriesMetadata		0x0000000a
	parameterInterpretation	<Enumeration Member>
	buffering	<Enumeration Member>
	coordinateSystem	<Enumeration Member>
	dimension	<UInteger16>
	datatype	<Array of TypeCodes carried as a UInteger8Array>
	units	<UnitsArray>
	parameterName	<String>
	abscissaUnits	<Units>
	abscissaIncrement	<Argument>
	abscissaOrigin	<Argument>
	abscissaOriginUncertainty	<Uncertainty>
	abscissaIncrementUncertainty	<Uncertainty>
	UpperLimit	<ArgumentArray>
	lowerLimit	<ArgumentArray>

Table 7. IEEE 1451.1 On-the-Wire Formats for Physical Parameter Structure

4.3.6 Arrays of Physical Parameter Data and Metadata

The on-the-wire format for an array of PhysicalParameter data or PhysicalParameter metadata is analogous to the on-the-wire format for an array of a 1451.1 primitive datatype. The two PhysicalParameter-related arrays that are defined by the Standard are

- PhysicalParameterDataArray

and

- PhysicalParameterMetadataArray.

The on-the-wire formats consist of the sequence of bytes:

<n>
<value of element$_0$>
< value of element$_1$>
• • •
< value of element$_{n-1}$>

Here, n is of type UInteger32 with value the number of elements in the array.

4.3.7 Argument and ArgumentArray

An Argument is defined in the Standard as a discriminated union with fifty-seven possible discriminate values. The discriminate indicates the 1451.1 datatype of the union's current value.

The discriminate values are enumeration members of the 1451.1-specified TypeCode enumeration (see Appendix A). This enumeration includes a typecode for ArgumentArray, ARGUMENT_ARRAY_TC, and the typecode EMPTY_TC. The datatype Argument itself has no typecode. Thus, the value of an Argument can be an ArgumentArray but not an Argument.

The on-the-wire format for an Argument, **arg**, consists of the sequence of bytes:

<discriminate of arg><value of arg>.

Here, the on-the-wire form of the discriminate is just the form for enumeration members, namely

|00|00|00|h_0h_1|,

where the discriminate's value is h_0h_1.

ArgumentArray is the only 1451.1 datatype that is directly marshaled onto the wire outside of the elements in a 1451.1 message header. The Standard requires that:

- The arguments (lower case argument, not Argument) in a client-server message or the content of a publication must be encoded into an ArgumentArray before being presented to a Client Port or publication port, respectively, for transmission over the network.

- Any user defined datatype which is to be communicated using the 1451.1 communication mechanism must first be encoded by user application code into an ArgumentArray before submission to a ClientPort or PublicationPort and decoded by application code into the original type by a server or subscriber.

The on-the-wire format for an ArgumentArray consists of the sequence of bytes:

<n>
<element$_0$>
< element$_1$>
• • •
< element$_{n-1}$>

Here, n is of type UInteger32 with value the number of members in the array and element$_i$ is either of type Argument or type ArgumentArray.

Note:

- In the case that an Argument member of an ArgumentArray has value that is an ArgumentArray, the above serialization of an ArgumentArray must be applied recursively.

- The discriminate EMPTY_TC signifies that the Argument union's current value is not bound. It is an error to try to marshal the value of an Argument with TypeCode EMPTY_TC.

4.3.7.1 Type Checking

The recipient of a marshaled ArgumentArray, the 1451.1 communication infrastructure, must demarshal the received sequence of bytes into a local ArgumentArray. In most implementations, this infrastructure will not be aware of the expected contents of the received array with respect to the number of array Argument members and the TypeCode discriminate of each member. Thus, in general, the recipient cannot perform any number of members and/or member datatype verification. Any such verification, if done at all, must be performed by the decoder of the ArgumentArray. In the case of a client-server message, the decoder is the target object's Perform operation. In the case of a publication, the decoders are the subscribers.

5 IEEE 1451.1 On-The-Wire Messages

Implementations of the Standard on IP must support two communication models:

- Point-to-point client-server

- Multicast publish-subscribe

The on-the-wire formats for client-server and publish-subscribe messages are described below.

Note that the marshaling and demarshaling of 1451.1 messages must obey the memory management rules of Section 15 of the Standard.

5.1 Client-Server Messages

Client-server communications in the Infrastructure use TCP/IP.[11] Hence, a client-server communication is reliable, and it can carry an arbitrary size payload.

5.1.1 Client-To-Server Message

A 1451.1 client-to-server message is carried as the payload of a TCP/IP message. This payload consists of a client-to-server message header as described below in Section 5.1.1.1 followed by the input arguments for the target server operation encoded in an ArgumentArray. The on-the-wire format of an argument array is described in Section 4.3.7, above.

The TCP/IP message payload for a 1451.1 client-to-server message is carried on the wire as a sequence of opaque bytes.

5.1.1.1 Client-To-Server Message Header

The same header format is used for the client-to-server messages that use an IEEE1451_ClientPort with ExecuteMode either EM_RETURN_VALUE or EM_NO_RETURN_VALUE. In addition, the same header is used for client-to-server messages that use an IEEE1451_AsynchronousClientPort. From the server's perspective, there is no difference between the three types of client service requests. The specified client-server header is used for all client-server messages independent of the class of the target server object.

The 1451.1 client-to-server message header is described in Table 8.

Byte Number	1451.1 In-Core Datatype	On-The-Wire Format In Header[12]	Interpretation
0 - 1	UInteger16	|00|ed|	Magic number for 1451.1 client-to-server message. Value: 0xed.
2 - 3	UInteger16	$\|h_0h_1\|h_2h_3\|$	IEEE 1451.1 IP on-the-wire request message protocol version number.
4 - 7	UInteger32	<UInteger32>	Total message length in 32-bit words.
8 - 9	UInteger16	$\|h_0h_1\|h_2h_3\|$	Total header length in 32-bit words. Determines where data starts.
10 -11	UInteger16	$\|h_0h_1\|h_2h_3\|$	Optional parameters block length in Octets not including any right-most padding. Length may be 0.
12 - 15	UInteger16	$\|00\|00\|h_4h_5\|h_6h_7\|$	The Client Port's cached BlockCookie.

[11] The decision to require the use of TCP only for point-to-point communications will need to be re-visited when Ipv6 becomes widely available.

[12] See Sections 4.3 and 4.5.2.2 for a description of the notation used in this column.

16 - 19	UInteger32	<UInteger32>	The id of the target server object: serverObjectIdentifier.
20 - 21	UInteger16	$\|h_0h_1\|h_2h_3\|$	The operation id of the desired operation on the target server object.
22 - 23	Enumeration Member	$\|00\|h_0h_1\|$	Execute mode[13], i.e., EM_RETURN_VALUE = 0, EM_NO_RETURN_VALUE = 1, or EM_ASYNCHRONOUS = 2.
24 ...	Zero or more Octets	$\|h_0h_1\|$ for each Octet followed by a right-most PAD[14] if necessary	TBD options, e.g., authentication information, quality of service, encryption method, data compression method, etc.

Table 8. Client-To-Server Message Header

5.1.1.1.1 Header Option Field

The last field in the header, the option field is semantically analogous to the option field in an IP header. The field is variable in length. There may be zero or more options. There are two cases for the format of an option:

1. A single octet of option-type.

2. An option-type octet, an option-length octet, and the actual option-data octets.

The option-length octet counts the option-type octet and the option-length octet as well as the option-data octets.

The semantics of the options block in a 1451.1 client-server message header are:

- The allowable options will be set by an as yet to be determined P1451.1a committee. Additional option may be added by this committee over time.

- All committee-approved options must be supported in all 1451.1 implementations using TCP/IP as a communications transport.

- The "optional" aspect is that a given option may appear or not in a specific 1451.1 client-server message header. That is, the use of an option in the header of a given client-server message is optional – the implementation of the option in a 1451.1 implementation is not.

Currently, there are no allowed options for 1451.1 client-server messages.

[13] This is an extension of the enumeration ExecuteMode defined in the Standard. However, the EM_ASYNCHRONOUS member is used only within an implementation of Client Ports and the communication infrastructure. It is not a part of the interface of any 1451.1 class.

[14] See Section 4.4.2 for the definition of PAD.

5.1.1.2 Client-To-Server TCP/IP Payload Format

The 1451.1 client-to-server header and the ArgumentArray carrying the input arguments for the target operation are marshaled into an opaque array. Each 1451.1 implementation must provide marshaler and demarshaler routines for transforming the client-to-server header to and from its opaque array form.

5.1.2 Server-To-Client Return Message

A 1451.1 server-to-client return message is carried as the payload of a TCP/IP message. This payload consists of a server-to-client message header as described below in Section 5.1.2.1 followed by the output argument results of the target server operation encoded in an ArgumentArray. The on-the-wire format of an argument array is described in Section 4.3.7, above.

The TCP/IP message payload for a 1451.1 server-to-client message is carried on the wire as a sequence of opaque bytes.

5.1.2.1 Server-To-Client Return Message Header

The 1451.1 server-to-client return message header is described in Table 9.

Byte Number	1451.1 In-Core Datatype	On-The-Wire Format In Header	Interpretation
0 - 1	UInteger16	|00|d5|	Magic number for 1451.1 server-to-client return message. Value: 0xd5.
2 - 3	UInteger16	$\|h_0h_1\|h_2h_3\|$	IEEE 1451.1 IP on-the-wire return message protocol version number.
4 - 7	UInteger32	<UInteger32>	Total message length in 32-bit words.
8 - 9	UInteger16	$\|h_0h_1\|h_2h_3\|$	Total header length in 32-bit words. Determines where data starts.
10 -11	UInteger16	$\|h_0h_1\|h_2h_3\|$	Optional parameters block length in Octets not including any right-most padding. Length may be 0.
12 - 15	UInteger32	<UInteger32>	ClientServerReturnCode as defined in the Standard.
16 - 17	UInteger16	$\|h_0h_1\|h_2h_3\|$	Server object's current BlockCookie.
18 - 19	Enumeration Member	$\|00\|h_0h_1\|$	Execute mode.

21 ...	Zero or more Octets	$\|h_0h_1\|$ for each Octet followed by a right-most PAD[15] if necessary	TBD options, e.g., authentication information, quality of service, encryption method, data compression method, etc.

Table 9. Server-To-Client Return Message Header

5.1.2.1.1 Header Option Field

The syntax and the semantics of the options field in a 1451.1 server-to-client return message header is the same as that of the option field in a client-to-server header described in Section 5.1.1.1.

Currently, there are no allowed options for 1451.1 server-to-client return messages.

5.1.2.1.2 Server-To-Client TCP/IP Payload Format

The 1451.1 server-to-client header and the ArgumentArray carrying the result arguments from the target operation are marshaled into an opaque array. Each 1451.1 implementation must provide marshaler and demarshaler routines for transforming the server-to-client return message header to and from its opaque array form.

5.2 Publish-Subscribe Messages

Publish-subscribe communications use UDP/IP multicast.[16] Hence, at least until Ipv6, publication communication is not reliable and, in the case that the data link layer used is IP, publications are limited in size to one IP frame, including all headers.

A 1451.1 publication message is carried as the payload of a UDP/IP message. This payload consists of a publication message header as described below in Section 5.2.1 followed by the contents of the publication encoded in an ArgumentArray. The on-the-wire format of an argument array is described in Section 4.3.7, above.

The UDP/IP payload is carried on the wire as an opaque array.

Each 1451.1 implementation must provide marshaler and demarshaler routines for transforming a 1451.1 publication message header to and from its opaque array form.

5.2.1 Publication Message Header

The contents of an IEEE 1451.1 publication header are given in Table 10.

Byte Number	1451.1 In-Core Datatype	On-The-Wire Format In Header	Interpretation

[15] See Section 4.4.2 for the definition of PAD.

[16] The decision to require use of UDP multicast only for publish-subscribe communications will need to be re-visited when Ipv6 becomes widely available.

0 - 1	UInteger16	|00|f7|	Magic number for 1451.1 publication multicast message. Value: 0xf7.
2 - 3	UInteger16	$\|h_0h_1\|h_2h_3\|$	IEEE 1451.1 IP on-the-wire publication protocol version number.
4 - 7	UInteger32	<UInteger32>	Total message length in 32-bit words.
8 - 9	UInteger16	$\|h_0h_1\|h_2h_3\|$	Total header length in 32-bit words. Determines where data starts.
10 -11	UInteger16	$\|h_0h_1\|h_2h_3\|$	Optional parameters block length in Octets not including any right-most padding. Length may be 0.
12 - 15	Enumeration Member	$\|00\|00\|00\|h_0h_1\|$	Publication Key as defined in the Standard, Section 12.3.2.
14 - 21	Octet[8]	$\|h_0h_1\|h_2h_3\|$ ... $\|h_{14}h_{15}\|$	Publication Domain as defined in the Standard, Section 12.3.5.
22 - 27	UInteger32	<UInteger32>	Hash of Publication Topic (See Section 5.2.1.2)
28- k	OctetArray	<OctetArray>	Publication Topic.
Zero or more Octets	Zero or more Octets	$\|h_0h_1\|$ for each Octet followed by a right-most PAD if necessary	Optional parameters, e.g., authentication information, quality of service, encryption method, data compression method, etc. Padded with zeros, if needed, to a multiple of four bytes.

Table 10. Publication Header

5.2.1.1 Header Option Field

The syntax and the semantics of the options field in a 1451.1 publication message header is the same as that of the option block in a client-server header described in Section 5.1.1.1.1.

Currently, there is one allowed option for 1451.1 publications, the *UniqueAddress* option-type.

The *UniqueAddress* option is a single octet option, and has value 0x0001. The semantics of the option is described in Section 7.1.2, Requirement 8.

5.2.1.2 Publication Topic Hash Value

The publication header has a 4-octet field, octets 28 through 31, that carries a hash code value of the publication's topic considered as a bit sequence.[17]

The hash algorithm used is the version of the CRC-32 polynomial algorithm used to compute Ethernet checksums.

Note that there are some requirements on the values and the use of these hash code values. These requirements are given in Section 7.

[17] This hash code value can be used in 1451.1 implementations to assist in deciding if a given publication is of interest to subscribers.

5.2.1.3 Publication UDP/IP Payload Format

The 1451.1 publication header and the ArgumentArray carrying the publication's contents are marshaled into an opaque array. Each 1451.1 implementation must provide marshaler and demarshaler routines for transforming a publication header to and from its opaque array form.

6 1451.1 Client-Server Communication Protocols

Common on-the-wire message formats alone are not sufficient to insure network-level interoperability between NCAPs using different implementations of the P1451.1a standard for IP. Common protocols with respect to the use of sockets, ports, point-to-point connections and multicast messages are also required.

The 1451.1 communication infrastructure when using TCP sockets is specified in this section.

The 1451.1 on IP client-server message protocols are specified as a set of requirements on an implementation of the 1451.1 communication infrastructure. These requirements are asymmetric in that the client-side and server-side requirements are quite different.

For notational brevity, in this section the client-side, client-server portion of an implementation of the 1451.1 communication infrastructure on an NCAP is referred to as the NCAP's *Client-Side Net Service.* The server-side, client-server portion of an implementation of the 1451.1 communication infrastructure on an NCAP is referred to as the NCAP's *Server-Side Net Service*.

6.1 Server-Side Net Service Port Numbers

For each NCAP in a 1451.1 system, each 1451.1 process space on that NCAP has an associated Server-Side Net Service. This Server-Side Net Service is the target of all messages from remote clients. Each *Server-Side Net Service* has an associated TCP port number. The value of the port number may be a member of the set of 64 ordered, IEEE 1451.1 reserved TCP port numbers.[18] The ordered set of reserved TCP port numbers is:

{TBD}

The first port number, TBD, is a well-known port number.

A 1451.1 implementation is not required to use only the reserved 1451.1 TCP port numbers, but it is strongly recommended that it does so.

6.2 Client-Server Communication Infrastructure Requirements

1. Client-Side Net Services and Server-Side Net Services and only Client-Side Net Services and Server-Side Net Services marshal and demarshal 1451.1 client-server messages.

[18] The reserved port numbers have been applied for.

2. In the case that an NCAP hosts only one 1451.1 application process, it is recommended that the portNumber associated with the NCAP's Server-Side Net Service be the first, well-known, reserved port number. [19]

3. All client-server communications use TCP (stream) sockets.

4. The closing of a connected TCP socket must be a graceful closing using the *shutdown* socket routine.

5. The only remote clients of a Server-Side Net Service are Client-Side Net Services requesting remote operation execution. The only remote clients of a Client-Side Net Service are Server-Side Net Services returning remote operation execution results.

6. If an object is both a remote server object and a remote client object, it must use distinct TCP ports for its different roles

7. During overall 1451.1 system configuration, each Client Port is supplied with the 1451.1 dispatch address of its associated server object. This dispatch address contains the server object host's IP address and the port number for the host's Server-Side Net Service.

8. For each NCAP process space, during 1451.1 initialization a TCP socket is created which is associated with a Server-Side Net Service's port number. The socket routine *accept* is used by the Server-Side Net Service to wait for and process *connection* requests from Client-Side Net Services.

9. A Client-Side Net Service uses local TCP sockets to *connect* to the Server-Side Net Service on remote hosts.

10. For each connection request from a Client-Side Net Service socket to the Server-Side Net Service, the server-side *accept* routine creates, as usual, a new TCP socket connecting to the requesting Client-Side Net Service's socket. This connection is used for both remote operation requests to server objects resident on the Server-Side Net Service's NCAP host and for the return of the operation's results to the requesting Client-Side Net Service in the case that the request does not have execute mode EM_NO_RETURN_VALUE.

11. The Client-Side Net Service must not close any connected local TCP socket which is being used for a remote operation request with execute mode EM_RETURN_VALUE or EM_ASYNCHRONOUS before the remote operation request returns or the participating Client Port times out.

12. In normal operation a Server-Side Net Service never closes a TCP socket connected to a Client-Side Net Service TCP socket on it own initiative. Rather, it closes a connected socket only as a consequence of a Client-Side Net Service closing its peer socket.

13. The only allowed case where a Server-Side Net Service can initiate the closure of a connection between it and a Client-Side Net Service is when the server-side's NCAP

[19] This requirement allows a limited form of component "plug-and-play".

does a graceful reset or shutdown. In this case, the Server-Side Net Service must observe requirement 4.[20]

14. Except for the above requirements, management of TCP sockets and connections is 1451.1 implementation dependent.

15. If the ExecuteMode of a client operation request message to a remote server is EM_NO_RETURN_VALUE or EM_ASYNCHRONOUS, the local Client-Side Net Service must return to the requesting Client Port as soon as it succeeds or fails in its attempt to initiate the transmission of the request message to the server's Server-Side Net Service. The value of the ClientServerReturnCode returned to the ClientPort by the Client-Side Net Service indicates the outcome of this attempt. This behavior is consistent with the "send and forget" nature of an EM_NO_RETURN_VALUE client-to-server message and the "let me know the result when you are done" nature of an EM_ASYNCHRONOUS client-to-server message.

16. If the ExecuteMode of a client operation request message to a remote server is EM_RETURN_VALUE or EM_ASYNCHRONOUS, then the Server-Side Net Service marshals the return values and sends an appropriate return message back to the requesting Client-Side Net Service. This return message is sent via the server-side TCP socket. That is, the return message must use the same TCP port pair connection used by the service request message. This return message must be sent to the Client-Side Net Service even if the invoked server operation has no return values.

17. If the ExecuteMode of a client operation request message to a remote server is EM_NO_RETURN_VALUE, the Server-Side Net Service discards any values returned by the operation and does not reply to the requesting Client-Side Net Service.

7 1451.1 Publish-Subscribe Message Protocols

1451.1 publish-subscribe messages use connectionless UDP (datagram) sockets and UDP multicast addresses. The 1451.1 communication infrastructure requirements when using UDP sockets are specified in this section.

For notational convenience, in this section the publisher-side, publish-subscribe portion of an implementation of the 1451.1 communication infrastructure on an NCAP is referred to as the NCAP's *Publisher-Side Net Service*. The subscriber-side, publish-subscribe portion of an implementation of the 1451.1 communication infrastructure on an NCAP is referred to as the NCAP's *Subscriber-Side Net Service.*

7.1 1451.1 Multicast Address and Port Number Pairs

There is a block of 64 UDP multicast addresses and a block of 64 UDP port numbers reserved for use by 1451.1 systems.[21] The addresses are TBD and the port numbers are TBD. Each

[20] Requirements 7, 9 and 10 imply that the initiative for creating and closing a Server-Side Net Service TCP socket almost always resides with the Client-Side Net Service.

[21] The reserved addresses and port numbers have been applied for.

reserved 1451.1 UDP multicast address is paired in a one-to-one manner with a reserved 1451.1 UDP port number. The 1451.1 multicast address-port number pairs are listed in Table 11.

UDP Multicast Address	UDP Port Number
TBD	TBD

Table 11. 1451.1 UDP Multicast Address-Port Number Pairs

Each publisher port has an associated multicast address-port number pair. This pair may be selected from the 64 reserved pairs. The pair is used by the Publisher-Side Net Service of the port's host NCAP for sending all publication messages emanating from the port. It is 1451.1 implementation dependent whether or not publication messages must use only multicast address-port number pairs from the reserved set.

Each subscriber port may have one or more associated multicast address-port number pairs. This pair may be selected from the 64 reserved pairs. These pairs are used by the Subscriber-Side Net Service of the port's host NCAP for selectively choosing publication messages to forward to the port. Whether or not subscriber ports have associated multicast address-port number pairs and how these pairs are used is 1451.1 implementation dependent.

7.1.1 Default Port Number and Multicast Address Pair

The multicast address-port number pair, TBD, is the default UDP port number and UDP multicast address used by any NCAP's Publisher-Side Net Service and Subscriber-Side Net Service. This default multicast address-port number pair can be overridden by other multicast address-port number pairs during 1451.1 application system instantiation, during application system configuration or at application system run-time. Whether such overriding is allowed and how it is done is 1451.1 implementation dependent.[22]

7.1.2 Setting Publisher Port and Subscriber Port Multicast Addresses and Port Numbers

The 1451.1 object classes, IEEE1451_BasePublisherPort and IEEE1451_SubscriberPort may each have implementation specific, network visible operations for setting and getting a publisher port's or a subscriber port's associated multicast address-port number pairs. If they do have such operations, the signatures of the operations must be as described below. The notation used for these signatures is that used for signatures in the Standard.

```
OpReturnCode SetPublicationAddressAndPort(
        in UInteger32 multicastAddress,
        in Uinteger32 portNumber);


OpReturnCode GetPublicationAddressAndPort(
        out UInteger32 multicastAddress,
        out Uinteger32 portNumber);
```

[22] The default multicast address-port number pair, TBD, permits a simple form of "plug-and-play' in 1451.1 implementations which support its use.

OpReturnCode SetSubscriptionAddressAndPort(
in UInteger32 multicastAddress,
in Uinteger32 portNumber);

OpReturnCode GetSetSubscriptionAddressAndPort(
out UInteger32 multicastAddress,
out Uinteger32 portNumber);

These four operations, the first two on IEEE1451_BasePublisherPort and the last two on IEEE1451_SubscriberPort, have Operation IDs 8223, 8224, 6216 and 6217, respectively.

7.2 Publish-Subscribe Communication infrastructure Requirements

1. Publisher-Side Net Services and Subscriber-Side Net Services and only Publisher-Side Net Services and Subscriber-Side Net Services marshal and demarshal 1451.1 publication messages, respectively.

2. All publication ports on an NCAP use the NCAP's Publisher-Side Net Service to send multicast publication messages. This is the only allowed mechanism for publishing 1451.1 messages.

3. Each Publisher-Side Net Service uses a local UDP socket to send multicast publication messages. The management of these UDP sockets is 1451.1 implementation dependent.

4. If a 1451.1 implementation admits the use of port numbers and multicast addresses not in the reserved blocks of Section 7.1, it still must pair each non-reserved multicast address used with a non-reserved UDP port number in a one-to-one manner.

5. If a 1451.1 implementation does support supplying a multicast address-port number pair to publisher ports during system configuration or at run-time, it must do so using the operations whose signatures are specified in Section 7.1.2.

6. If a 1451.1 implementation does support supplying multicast address-port number pairs to subscriber ports, it must do so using the operations whose signatures are specified in Section 7.1.2.

7. How multicast address-port number pair information is used by an NCAP's Publisher-Side Net Services and Subscriber-Side Net Services, if at all, is 1451.1 implementation dependent.

8. The topic hash value 0x00000000 in a publication's header is distinguished. A publication has header topic hash value 0x00000000 if the publication topic's contains wild cards.[23]

[23] There is a small probability, $\sim 2^{-32}$, that the actual hash value of a publication topic without wildcards is 0x00000000. Any 1451.1 implementation must handle this case correctly, for example, by always performing a full publication topic to subscription qualifier match in the case of topic hash value 0x00000000.

9. If a publication header's option field contains the UniqueAddress option octet, the Subscriber-Side Net Service must interpret the presence of this option to signify that the received publication's address-port number pair is such that the pair uniquely identifies the publication with respect to its domain, key and topic.

10. A 1451.1 implementation's algorithms for matching a publication's key, domain and topic with a subscriber port's key, topic and subscription qualifier, respectively, must adhere to the matching rules specified in Section 12.3.4.1 of the Standard.

Appendix A: IEEE 1451.1 TypeCode Enumeration Values

The IEEE 1451.1 standard constrains the datatype of each argument in a client-server message or each datum in the contents of a publication to be one of a specified set of 56 non-empty datatypes. The standard assigns a member in the TypeCode enumeration for each of these types. The TypeCode members and their values are listed in Table A-1.

TypeCode Enumeration Member	Integer Value (Hex)
EMPTY_TC (The TypeCode of an Argument with unbound value)	0x0
ARGUMENT_ARRAY_TC	0x1
BIT_SEQUENCE_TC	0x2
BIT_SEQUENCE_ARRAY_TC	0x3
BOOLEAN_TC	0x4
BOOLEAN_ARRAY_TC	0x5
CLASS_ID_TC	0x6
CLIENT_PORT_PROPERTIES_TC	0x7
CLIENT_PORT_PROPERTIES_ARRAY_TC	0x8
CLIENT_SERVER_RETURN_CODE_TC	0x9
FLOAT32_TC	0xa
FLOAT32_ARRAY_TC	0xb
FLOAT64_TC	0xc
FLOAT64_ARRAY_TC	0xd
INTEGER16_TC	0xf
INTEGER16_ARRAY_TC	0xe
INTEGER32_TC	0x10
INTEGER32_ARRAY_TC	0x11
INTEGER64_TC	0x12
INTEGER64_ARRAY_TC	0x13

INTEGER8_TC	0x14
INTEGER8_ARRAY_TC	0x15
OBJECT_DISPATCH_ADDRESS_TC	0x16
OBJECT_ID_TC	0x17
OBJECT_PROPERTIES_TC	0x18
OBJECT_PROPERTIES_ARRAY_TC	0x19
OBJECT_TAG_TC	0x1a
OBJECT_TAG_ARRAY_TC	0x1b
OCTET_TC	0x1c
OCTET_ARRAY_TC	0x1d
OP_RETURN_CODE_TC	0x1e
PHYSICAL_PARAMETER_DATA_TC	0x1f
PHYSICAL_PARAMETER_DATA_ARRAY_TC	0x20
PHYSICAL_PARAMETER_METADATA_TC	0b21
PHYSICAL_PARAMETER_METADATA_ARRAY_TC	0x22
PUBLICATION_TOPIC_TC	0x23
PUBLISHER_INFORMATION_TC	0x24
PUBSUB_DOMAIN_TC	0x25
STRING_TC	0x26
STRING_ARRAY_TC	0x27
SUBSCRIPTION_QUALIFIER_TC	0x28
TIME_INTERVAL_TC	0x29
TIME_INTERVAL_ARRAY_TC	0x2a
TIME_REPRESENTATION_TC	0x2b
TIME_REPRESENTATION_ARRAY_TC	0x2c

UINTEGER16_TC	0x2d
UINTEGER16_ARRAY_TC	0x2e
UINTEGER32_TC	0x2f
UINTEGER32_ARRAY_TC	0x30
UINTEGER64_TC	0x31
UINTEGER64_ARRAY_TC	0x32
UINTEGER8_TC	0x33
UINTEGER8_ARRAY_TC	0x34
UNCERTAINTY_TC	0x35
UNCERTAINTY_ARRAY_TC	0x36
UNITS_TC	0x37
UNITS_ARRAY_TC	0x38

Table A-1. IEEE 1451.1 TypeCode Enumeration

Appendix B: Proposed Additional 1451.1 Communication Return Codes

The IEEE 1451.1 standard in Section 7.2.3.1.2 defines the syntax and the semantics of a ClientServerReturnCode. As currently defined in the standard, there is no way to signify that an error has occurred in the communications infrastructure or the type of that error. It is proposed that a set of return codes be used in the portCode and performCode fields of a ClientServerReturnCode to signify such errors. The following table lists a draft set of such codes.

Communication Return Code Enumeration Member
CC_OK
CC_UNSPECIFIED_ERROR
CC_PROTOCOL_VERSION_MISMATCH
CC_MARSHALING_FAILURE
CC_DEMARSHALING_FAILURE
CC_CLIENT_SIDE_SEND_TIMEOUT
CC_SERVER_SIDE_RETURN_TIMEOUT
CC_AUTHENTICATION_FAILURE
CC_ENCRYPTION_FAILURE
CC_DECRYPTION_FAILURE
CC_COMPRESSION_FAILURE
CC_DECOMPRESSION_FAILURE
TBD …