A Kennedy Space Center Implementation of IEEE 1451 Networked Smart Sensors and Lessons Learned

Rebecca L. Oostdyk
ASRC Aerospace
M/S: ASRC-25
Kennedy Space Center, FL 32899
321-867-5637
Rebecca.Oostdyk-1@ksc.nasa.gov

Carlos T. Mata
ASRC Aerospace
M/S: ASRC-25
Kennedy Space Center, FL 32899
321-867-6964
Carlos.Mata-1@ksc.nasa.gov

José M. Perotti
NASA, KSC
M/S: YA-D5
Kennedy Space Center, FL 32899
321-867-6746
Jose.M.Perotti@nasa.gov

Abstract—To meet[1,2] the need for more specific and reliable information from ground support instrumentation systems and future spacecraft sensors and to support Intelligent Health Management Systems (IHMS), NASA's Instrumentation Branch and ASRC's Advanced Electronics and Technology Development Laboratory at Kennedy Space Center (KSC) have consulted the IEEE 1451 family of smart-sensor standards to develop smart network elements (SNEs). SNEs provide reliable signal conditioning to raw sensors, complex data processing, and communication capabilities with a light implementation of the IEEE 1451 family of standards. They are capable of assessing the health of the raw sensors and the electronics and the reliability and tolerance of the measurement, and they relay this information to higher-level systems.

The KSC SNEs employ a scaled-down version of the IEEE 1451.1 object model for smart sensors. The sensors are capable of publish-subscribe and client-server communication over an Ethernet network using a custom on-the-wire transmission format that is bandwidth-conservative. KSC, together with NASA Stennis Space Center, has also implemented a user-defined transducer electronic data sheet (TEDS) to store health information about the sensor, defined as the health electronic data sheet (HEDS). The KSC SNEs expand upon the IEEE 1451 family of standards to include a well-defined communication protocol for high-level sensor-to-sensor interaction and a HEDS structure for passing relevant health data over the network.

TABLE OF CONTENTS

[1] 0-7803-9546-8/06/$20.00© 2006 IEEE
[2] IEEEAC paper #1520, Version 3, Updated Jan, 06 2006

1. INTRODUCTION

KSC expends significant effort maintaining sensors that support both ground and flight systems. Reducing the calibration cycles of the sensors and increasing their reliability will result in cost savings for the space program. Allowing operators to view meaningful information about the data, including engineering units and confidence indications, can reduce human error and save time in troubleshooting problematic sensors. The need to save time and money, increase reliability and safety, and reduce errors is the driving force behind KSC's development of SNEs. The sensors are required to operate over a wide temperature range, be intrinsically safe, and be small and light enough for easy installation and removal. In addition, the SNEs have enough computing power to perform real-time complex calculations of over sampled data.

In researching the state of the art for smart sensors, KSC reviewed the IEEE 1451 family of smart sensors. IEEE 1451 is a comprehensive volume describing the architecture, message formats, software objects, and communication protocols within a smart sensor. Because the standard is excessively thorough in defining smart sensors, its complexity often turns away manufacturers simply because of the expected development costs of implementing all the required elements. KSC decided to simplify the architecture to include a raw sensing element, an integrated transducer interface module (TIM), and a network-capable application processor (NCAP). KSC has also chosen to focus on IEEE 1451.1 [1], which describes NCAP-to-NCAP communication. By looking at IEEE 1451.1, KSC could use its resources for developing high-level communications rather than concentrating on communication between the NCAP and TIM or between the TIM and the raw sensors. This simplification, along with narrowing down the software objects to be included in the

KSC smart sensor, has made the implementation of the standard much more manageable.

The SNE is a smart sensor that contains information about itself in a TEDS format and is able to verify and validate its own data, communicate with other sensors, and notify higher levels of its health using a HEDS. Some traditional networked sensors provide redundancy and data verification, whereas the SNEs can perform conversions and run algorithms to detect sensor data and instrumentation health.

Although many smart sensors have already made their way into production, there is no standard communication protocol in place that allows the sensors to share data with one another or with higher levels of the network. As a result, KSC has modified the Agilent/Boeing "1451.1 On-the-Wire Format for IP" [2] to suit its need for sending data over an Ethernet network. The KSC implementation significantly reduces the bandwidth requirements for the network by compressing data into the least number of bytes. This protocol could be expanded over any type of network to accommodate sensors that are not Ethernet-ready.

This paper will present the custom architecture, framework, and protocols that have been defined by KSC to develop a networked smart sensor. The lessons learned from implementing a light version of IEEE 1451.1 should reduce development time and give manufacturers an idea of smart-sensor capabilities. The network-dependent communication protocol that KSC has developed should also lay the groundwork for manufacturers to create NCAPs that can interact with one another over an Ethernet network.

2. Advantages and Disadvantages of IEEE 1451 Implementation

KSC's decision to use the IEEE 1451.1 object model for the smart-sensor firmware required an in depth look at the advantages and disadvantages of the IEEE 1451 standard. The major advantage of the standard was that the smart-sensor architecture had already been defined, along with the software functions to keep the sensor in a known state. In addition, the standard allowed enough flexibility that custom functions could be added to the firmware and the object model could be trimmed down to include only the necessary methods. Implementing a light version of IEEE 1451.1 would give the SNE the functionality KSC required, while not restricting it from further firmware upgrades.

Despite the pros surrounding the IEEE 1451.1 standard, there are several key tradeoffs that manufacturers should consider before choosing to implement an IEEE 1451 smart sensor. For example, the standard gives examples of the smart-sensor firmware in object-oriented programming style. If the embedded controller for a particular application does not support object-oriented programming, the task of implementing the firmware can become more complex. Also, because of the nature of object-oriented languages, an IEEE 1451.1 smart sensor may become burdened by the amount of memory required to instantiate objects in the firmware.

Another key concern about IEEE 1451 is the lack of a defined protocol for allowing communication between NCAPs. If the NCAPs do not encode data onto the network and decode data as it is received from the network in the same manner, there is the possibility of data being misinterpreted. Therefore, it becomes necessary to define the on-the-wire format for all messages exchanged between NCAPs. Agilent and Boeing have developed a written standard for this purpose, but the bandwidth requirements may be too cumbersome for a network supporting a large number of sensors.

The lack of availability of smart products, including transducers, TIMs, and NCAPs, is another obstacle that manufacturers face. Unless a manufacturer is interested in creating its own TIM or NCAP product, there are few smart sensors available to connect to the smart transducers that are being produced. Without continued advances in NCAPs and TIMs, transducers that come equipped with TEDS are of no value. The NCAPs must also be developed with interoperability in mind so that one manufacturer's NCAP can interact with another.

With all of the questions that were raised about IEEE 1451.1, KSC determined that the benefits of a more robust, reliable, and user-friendly smart sensor would outweigh the concerns. However, to make the smart sensors practical for KSC applications, the IEEE 1451.1 standard had to be expanded and modified.

3. Custom Smart-Sensor Architecture

The IEEE 1451 family of smart-sensor standards defines the architecture of a smart sensor. A simplified model of the architecture from IEEE 1451.2 can be found in Figure 1.

From the figure, there are two distinguishable entities within the smart sensor, the NCAP and the TIM (seen here as the smart TIM, or STIM). The separation of the transducer interface from the network interface allows transducers to be removed from the TIM and replaced without interrupting network communications and gives the flexibility required for a plug-and-play device. However, the separation also requires the NCAP to handle two different interfaces, one with the network and one with a transducer-independent TIM.

To relieve the NCAP of the latter task and preserve memory within the NCAP, KSC has integrated the TIM and NCAP into one device. The SNE's architecture is illustrated in Figure 2.

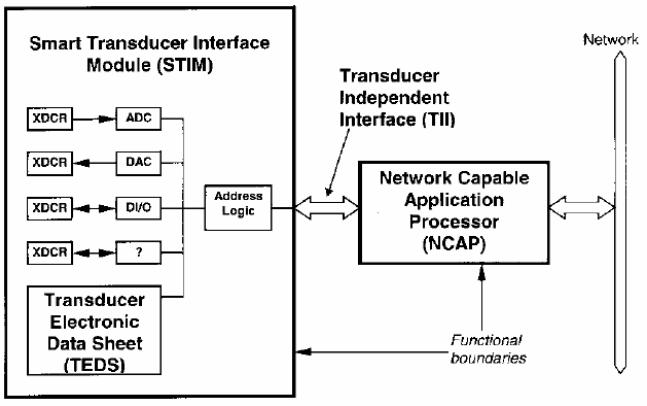

Figure 1 – IEEE 1451 Smart-Sensor Architecture

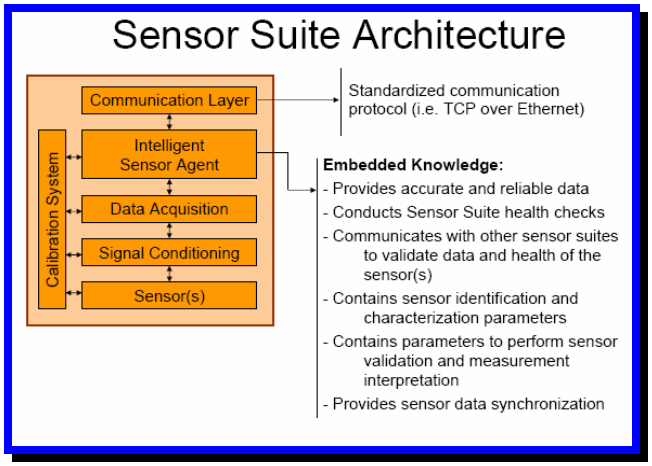

Figure 2 – Kennedy Space Center's SNE Architecture

The Communication Layer of the SNE provides NCAP-like functionality and an interface to an Ethernet network, while the Intelligent Sensor Agent, Data Acquisition, and Signal Conditioning blocks make up the TIM functions, including the TEDS. The KSC smart sensor is desirable over the IEEE 1451-defined architecture because it simplifies the firmware for the NCAP and allows the sensor to be easily removed in the field. Although the SNE allows plug-and-play activity over the network, the entire sensor suite must be reconfigured in order to swap a transducer in the module.

Once the SNE had been defined, the development of the hardware began. The conversion from analog to digital signals and signal conditioning are carried out on an analog circuit board. The analog board has two analog-to-digital converters and two multiplexers and is able to sample up to eight different transducers simultaneously. The signals are then routed into a digital board, which contains the main engine of the smart sensor, a floating-point digital signal processor (DSP). The DSP can be programmed in C++; therefore, it was a natural fit for the object-oriented programming model given in IEEE 1451.1. The DSP performs calculations and health checks on the digital signals and then marshals the information onto the Ethernet network via a commercial off-the-shelf Digi Connect ME.

The Digi Connect ME transforms the serial data from the DSP into User Datagram Protocol (UDP) packets for transmission over a 10/100-BaseT Ethernet network. The TEDS for the SNE reside in flash memory on the digital board and are accessible by the DSP. This particular implementation of the SNE is powered using Power over Ethernet (IEEE 802.3af). The power is injected over the spare pairs of the Ethernet cable, passed through the Digi Connect ME, and transformed using a DC-DC converter on the power circuit board. A final feature of the hardware is a real-time clock for use in time-stamping data as it is sent over the network. Although IEEE 1451.1 makes no concession for synchronizing clocks on the network, a future SNE will employ Precision Time Protocol (IEEE 1588) to maintain synchronization of real-time clocks on the network within a few microseconds.

4. Smart-Sensor Object Model

After creating a robust hardware platform for the smart sensor, KSC began to consider the firmware. The IEEE 1451.1 object model provides a flexible, yet well-defined platform from which to start. Using examples in the appendices of IEEE 1451.1, a light version of the classes in the standard was developed to support the KSC application. The KSC smart sensor is deemed "light" because it implements a small subset of the classes defined in IEEE 1451.1. The KSC smart-sensor object model is presented in Figure 3.

Each block in Figure 3 represents an IEEE 1451.1-defined class. The blocks with bold outlines represent network-visible software objects, or objects that can be accessed via client-server communication over the network. Every IEEE 1451.1 smart sensor must include an NCAP block, which contains the information required for network communications. The transducer block implemented in the SNE is an IEEE 1451.4 [3] transducer block and contains the TEDS required by the standard. The function block is a custom version of the IEEE 1451 function block. The KSC function block contains the same methods as other function blocks, but it also includes functions to access the thresholds and publication contents of the function block's publisher ports. With this scheme, an operator of a higher-level system on the network can change the threshold values that the smart sensor routinely checks as it takes measurements, and the operator can choose whether the data, metadata, or both data and metadata will be published.

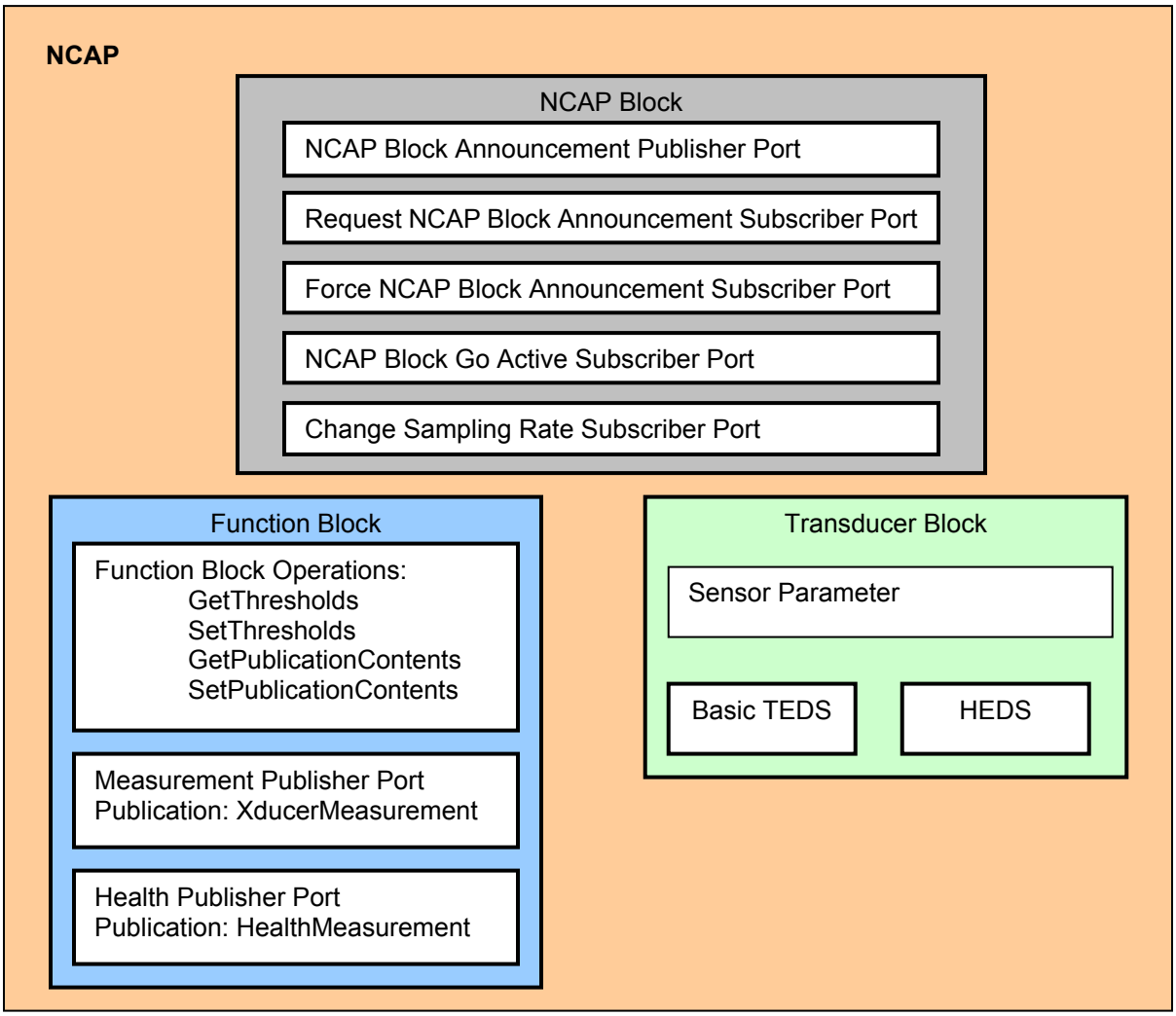

Figure 3 – KSC Smart-Sensor Object Model

5. On-the-Wire Data Format

After creating the classes and implementing the firmware for the object model on the DSP, the next important task was defining how one NCAP would communicate over the network with another NCAP. The SNE will integrate with an expert system that can perform intelligent algorithms, data mining, and feature extraction from the SNE data. For the expert system to configure the SNE and understand its published data, a communication protocol had to be developed.

IEEE 1451.1 does a thorough job in defining the methods to publish data and configure the smart sensor, but it remains silent on how the messages will be encoded and decoded over the network. Agilent and Boeing jointly developed a protocol for communication over an Ethernet network to fill the gap left by the IEEE 1451.1 standard, and KSC adapted Agilent and Boeing's "1451.1 On-The-Wire Format for IP" to conserve bandwidth.

Primitive Datatypes

IEEE 1451.1 uses several primitive datatypes that are commonly defined across platforms. When issuing a message containing a value that has a primitive datatype, the rules in Appendix A, Table 1, will apply.

Arrays of Primitive Datatypes

When a 1451.1 message contains an array of primitive datatypes, the values in the array shall be preceded by the number of elements in the array, represented by a 16-bit unsigned integer (UInteger16). The 0^{th} (or leftmost) element in the array shall be the first value of the given datatype, and successive elements in the array shall then follow. The on-the-wire format for the arrays is given in Appendix A, Table 2.

Fixed-Length OctetArrays

When a 1451.1 message contains a Units, UnitsArray, or PubSubDomain datatype, the format is the same as an OctetArray, but with a fixed length for the number of octets.

The on-the-wire formats for fixed-length OctetArrays are given in Appendix A, Table 3.

String and StringArrays

A String is an IEEE 1451-defined datatype that is a structure with members characterSet from the StringCharacterSet enumeration, characterCode from the StringCharacterCode enumeration, language from the StringLanguage enumeration, and stringData, which is an OctetArray containing the String's content. The octets of stringData will be interpreted differently based on the value of the characterSet, characterCode, and language members. The on-the-wire formats Strings and StringArrays are given in Appendix A, Table 4.

Typedefed Datatypes

Datatypes that are defined in IEEE 1451.1 as a typedef of one of the datatypes whose on-the-wire format is given by the primitive datatypes or arrays of primitive datatypes use the same on-the-wire format. These typedefed datatypes are defined along with their corresponding primitive datatypes in Appendix A, Table 5.

BitSequenceArray and ObjectTagArray

The BitSequenceArray and ObjectTagArray datatypes are defined in IEEE 1451.1 as an array of OctetArrays. The on-the-wire format for these datatypes is given in Appendix A, Table 6.

Heterogeneous Datatypes

The on-the-wire formats of heterogeneous datatypes defined in IEEE 1451.1, including structures, union, arrays of structures, and arrays of unions, are derived from the basic on-the-wire formats that have already been defined. However, they are presented explicitly here to clarify the order in which the members of the heterogeneous datatypes will be marshaled onto the network.

The ObjectDispatchAddress Structure—The ObjectDispatchAddress of an IEEE 1451.1 software object is network- and implementation-dependent. For the purposes of the SNE, the network shall be an Ethernet network using UDP/Internet Protocol (IP) as a transport protocol, and the ObjectDispatchAddress shall be defined by the following structure:

```
struct ObjectDispatchAddress

{

        UInteger32 ipAddress;

        UInteger16 portNumber;

        UInteger32 objectIdentifier;

}
```

The ipAddress and portNumber members refer to IP address and port number required to communicate with the object. The objectIdentifier is equal to the minor universal unique identifier (UUID) of the object, where the four octets of the minor UUID value have been logically OR'ed together to create a UInteger32.

General On-the-Wire Format of Structures—The general on-the-wire format for a heterogeneous datatype is the sequence of field names used in IEEE 1451.1 to define the datatype. If a given field name is also a heterogeneous datatype, then that field name is marshaled as a sequence of field names. The process of marshaling the sequence of field names is repeated in a recursive manner until the datatype is decomposed into field names whose on-the-wire format has already been defined. The result is an ordered forest of field names as given below.

Field $name_0$

Field $name_1$

    Field $name_{10}$

    Field $name_{12}$

        Field $name_{120}$

        Field $name_{121}$

        …

        Field $name_{12i}$

    …

    Field $name_{1j}$

…

Field $name_k$

Order of Fields for Defined Heterogeneous Datatypes—To ensure compatibility between IEEE 1451.1 devices, the order in which the fields of heterogeneous datatypes are sent over the network must be defined. Appendix A, Table 7, is devoted to defining the order of the fields on the wire for IEEE 1451.1 heterogeneous datatypes.

Arrays of Structures—Arrays of IEEE 1451.1 structures are placed into their on-the-wire format using the same rules as arrays of 1451.1 primitive datatypes. The first two bytes give the number of elements in the array, and the following bytes are a sequence of the structures in the array. Appendix A, Table 8, lists the IEEE 1451.1 arrays of structures datatypes and their on-the-wire formats.

Physical ParameterTypes—Physical parameters in IEEE 1451.1 can contain both data and metadata, and IEEE 1451.1 defines unions for PhysicalParameterData and PhysicalParameterMetadata that can be transmitted over the network. The discriminant of the unions for both data and metadata is a member of the PhysicalParameterType enumeration, declared 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 the PhysicalParameter types consists of a UInteger8 value representing the PhysicalParameterType discriminant and the structure for the union's value. The on-the-wire format of the PhysicalParameter types is given in Appendix A, Table 9.

Arrays of Physical Parameter Types—Arrays of IEEE 1451.1 Physical Parameter types are placed into their on-the-wire format using the same rules as arrays of IEEE 1451.1 primitive datatypes. The first two bytes gives the number of elements in the array as a 16-bit unsigned integer, and the following bytes are a sequence of the PhysicalParameterData or PhysicalParameterMetadata elements. Appendix A, Table 10, lists the IEEE 1451.1 arrays of Physical Parameter datatypes and their on-the-wire formats.

Argument and ArgumentArray Types—The Argument and ArgumentArray datatypes provide the infrastructure for marshaling messages directly onto the wire. An Argument is a discriminated union whose discriminant is the datatype of the union's current value. The datatype discriminant comes from the TypeCode enumeration, which contains 57 different datatypes defined in IEEE 1451.1, and the union has a value that corresponds to the discriminated datatype. An ArgumentArray consists of an array of Argument elements. The ArgumentArray is the only datatype that can be marshaled onto the wire; therefore, the contents of every publication or client-server message must be encoded into an ArgumentArray before it can be transmitted over the network. The on-the-wire format of the Argument and ArgumentArray types is given in Appendix A, Table 11.

6. On-the-Wire Message Formats

The data formats described in the previous section are the building blocks for messages sent between NCAPs. There are two types of communication that are specified by IEEE 1451.1, and each one has its own distinguishing message format.

Publish-Subscribe Communication

The first communication style is publish-subscribe. Publish-subscribe is a passive form a communication in which one NCAP subscribes to a publication and does not send a response once the publication is received. Therefore, a publication is sent over the network as the payload of a UDP/IP packet. The publication is not guaranteed to arrive to the subscriber, may be unreliable depending on the loading of the network, but is useful for sending streaming data. The size of the publication is limited to the maximum size of a 16-bit unsigned integer, or 65,535 bytes. Each publication contains a header, and there may be optional publication contents following the header.

The Publish-Subscribe Message Header—The message header for a publication contains several fields that make it unique so that the subscriber can differentiate the publication. The first field is a single byte that alerts the subscriber that the message is in fact a publication. The unique hexadecimal number for a publication is 0xF7. The next two bytes contain a 16-bit unsigned integer for the total number of bytes in the message, followed by two more bytes of a 16-bit unsigned integer for the length of the header. The publication key, domain, and topic make up the next three fields, where the publication key is the value from the PublicationKey enumeration, the domain is an array of eight octets that describe where the publications should be directed over the network, and the topic is a unique octet array with the name of the publication. Following the message header is an ArgumentArray with the publication contents, and the contents are marshaled onto the network based on the data formats that have already been described. The publish-subscribe on-the-wire message format is also described in Appendix B, Table 12.

Client-Server Communication

The second type of communication defined by IEEE 1451.1 is client-server. Client-server communication involves an NCAP making a request to another NCAP to perform an operation. Once the operation is complete, the server NCAP returns a message to the client NCAP about the status of the operation and sends any requested data. The client-server messages can be sent as the payload of either a Transmission Control Protocol (TCP)/IP or UDP/IP packet. Although TCP/IP is preferred for reliability reasons, the KSC implementation uses UDP to conserve bandwidth on the network. The size of the messages is limited to the maximum size of a 16-bit unsigned integer, or 65,535 bytes. The client-to-server and server-to-client messages have distinct headers, and the headers may be followed by input or output arguments as required by the requested operation.

The Client-to-Server Message Header—All client-to-server messages, including messages from ClientPorts or AsynchronousClientPorts and with different execute modes, have the same header format. The header format is also independent of the class of the target server object. Following the header are the input arguments for the target server object. If no input arguments are required, then the number of arguments in the ArgumentArray shall be given as 0x0000 (a 16-bit unsigned integer).

The first field in the header is a byte that identifies the message as a client-to-server message, the hexadecimal number 0xED. The next two bytes are interpreted as a 16-bit unsigned integer for the total length of the message. Another set of two bytes contains the block cookie of the client port as a 16-bit unsigned integer. Next are a 32-bit unsigned integer that represents the object ID of the server where the operation is being requested and a 16-bit unsigned integer for the operation ID. Finally, there is a byte for the execute mode of the operation, which is taken from the ExecuteMode enumeration. If there are input arguments to the operation, they will follow the execute mode as an ArgumentArray. The client-to-server message format is described in Appendix B, Table 13.

The Server-to-Client Message Header—Once the operation has been completed on the server, the server sends a return message to the client. The first byte of the return message identifies it as a server-to-client message with the hexadecimal number 0xD5. The next two bytes contain the total message length represented by a 16-bit unsigned integer. A 32-bit unsigned integer represents the client-server return code that results from the operation, and a 16-bit unsigned integer returns the block cookie from the server. Following the block cookie is the execute mode of the operation, the object ID of the server that performed the operation, and the operation ID. After the message header, any output arguments from the operation are transmitted in the form of an ArgumentArray. The server-to-client message format is defined in Appendix B, Table 14.

7. Custom Transducer Electronic Data Sheets (TEDS)

One of the major accomplishments of IEEE 1451 is the definition of TEDS for use by smart transducers. Manufacturers can provide TEDS embedded with their transducers to provide intelligent systems with more information about the transducer. To this aim, KSC has developed a user-defined TEDS to supplement the IEEE 1451.4 Basic TEDS [3]. The user-defined TEDS contains state information about the transducer, including thresholds for the transducers parameters that are dependent on state. The thresholds are used in determining sensor health; therefore, the user-defined TEDS has been named the Health Electronic Data Sheet, or HEDS.

The first four fields of the HEDS give the length of the HEDS, the HEDS identifier, and version number. Following this short header is the state information for the first state, ST0. Every HEDS must include at least one state, ST0, for the health startup parameters. Additional states are optional, and they shall follow the ST0 structure. The state information consists of the threshold values for the mean, minimum, maximum, and standard deviation. A smart sensor will use the thresholds in the HEDS to determine if the readings from the transducer are in range for a given state. If the parameters are out of range, the sensor may send a health publication message to interested subscribers that indicates which parameters have gone out of range.

8. Lessons Learned

The steps that KSC has taken to develop a network-enabled smart sensor (SNE) based on IEEE 1451 have not come without valuable lessons for industry. The architecture and framework that KSC has defined, as well as the standard network communication protocol, should reduce development time and give manufacturers an idea of smart-sensor capabilities. In addition, the KSC smart-sensor development process can be used as a model for manufacturers to take requirements to end product.

Before a manufacturer begins the design process, it is important to answer a few key questions.

(1) What are the transducer requirements for the system? Are there IEEE 1451-compliant smart transducers available that meet the requirements?

(2) Is it desirable to relieve the higher-level system of processing tasks that could be performed by a smart sensor? If so, is it cost-effective?

(3) What kind of processing functions should the smart sensor be able to perform? How much processing power and memory are required?

(4) Is it necessary for the smart sensor to perform two-way communication with the system, or is data publishing all that is required?

(5) Does the processor have enough peripherals to service the transducers, system communications, and any other required hardware (such as a real-time clock)?

(6) What is the minimum set of IEEE 1451.1 classes required to implement the processing and communication functions? Does the processor have the power to implement these functions?

(7) Is network communication required to be highly reliable, or can reliability be sacrificed for bandwidth (i.e., should UDP/IP or TCP/IP packets carry the communication messages)?

(8) Will the complexity of the smart sensor reduce the reliability of the system by creating more points of failure, or will its ability to monitor its own health make the system more robust? Is flexibility in the system or reliability more important?

The KSC smart sensor was required to measure ambient temperature, pressure, and cryogenic temperatures. As there were no IEEE 1451 sensors for this purpose, KSC had to undertake the development itself. The higher level system that interacts with the smart sensors is a Gensym G2 expert system with a high volume of processing tasks, therefore it is useful for the smart sensors to be able to process data before sending to the expert system. These processing functions include converting the counts from the analog to digital converters into engineering units, compensating the values for the voltage reference, comparing the measured values to thresholds, and making determinations about the health of the sensor and measurement. Although the main purpose of the sensor is to report this information to the expert system, there are also configuration commands that the smart sensor must handle to give the SNEs more flexibility and robustness. For example, the sampling rate of the SNEs could be changed based on the state of the overall system in order to reduce the number of redundant measurements or to take a closer look at an anomaly or process. The Digi Connect ME Ethernet module provides the communication peripherals for the SNE to interact with the expert system, and the RS-232 and serial peripheral interface (SPI) ports of the DSP allow it to communicate with the real-time clock and analog to digital converters.

For the KSC smart sensor implementation, only a small subset of the IEEE 1451.1 classes were required in order to communicate with the expert system, including the NCAP block, transducer block, and function block. As a result of the minimal IEEE 1451.1 classes, the functions could be implemented on a 200 MHz Texas Instruments DSP with 256 kilobytes of RAM. Although the DSP has plenty of power for floating point calculations, the classes consumed a majority of the memory, leaving little room for saving historical data or adding more classes. A review of the SNE firmware for efficiency and further trimming of the IEEE 1451 protocols may result in better memory usage. Offloading some of the encoding and decoding IEEE 1451 functions onto the Digi Connect ME's ARM processor may also relieve the memory burden on the DSP.

As a result of redundant sensors and cross-checking of measurements in the expert system, it was determined that bandwidth is more important to the system than reliability. Because sensors are continuously publishing data to the expert system and due to the handshaking in client-server communication, any lost data will quickly be sent again over the network, and as a result, a UDP/IP packet with a smaller header would be more efficient than a larger TCP/IP packet.

The final question of reliability versus robustness is highly dependent on the final application of the smart sensors. The prototype smart sensor will begin to answer this question as it is tested in live demonstrations and testbeds. The reliability of the end product will be determined by the quality of the SNE's firmware and its ability to handle errors, as well as the integrity of the electrical and mechanical components. Although there are many more parts involved in the smart sensor, it has been designed with redundant analog components to mitigate failures, and by placing a number of different kinds of sensors to monitor a single process, failures by one smart sensor can be ruled out. KSC is dedicated to developing smart sensors that may initially be more expensive to install, but that will pay off in the future in terms of maintenance costs and fault detection.

Taking the time to answer these questions and to carefully review the standards and protocols that have been set forth is vital to the success of the smart sensor. If there are doubts about the amount of development required to implement the sensors or about the cost and availability of the required hardware, an IEEE 1451 smart sensor may be inefficient for the application. However, if a manufacturer does choose IEEE 1451 to guide the development of an intelligent networked sensor, the benefits will be amplified as the technology matures.

Smart sensors can initially be developed using a light version of IEEE 1451 and may be built upon in the future to enhance the flexibility and robustness of the system. Kennedy Space Center has developed prototype SNEs for the purpose of demonstrating the capabilities of IEEE 1451 but plans to continue development by incorporating additional sensor and data health algorithms, a time synchronization protocol for keeping system time (IEEE 1588), and calibration TEDS.

ACKNOWLEDGEMENTS

Some of the work performed by KSC on SNEs has been part of the Integrated Intelligent System Health Management (IISHM) project. The IISHM overall project is led by Dr. Fernando Figueroa from Stennis Space Center. The authors would like to acknowledge the support from Dr. Figueroa and Stennis Space Center during the development of the SNEs.

REFERENCES

[1] Institute of Electrical and Electronics Engineers, Inc., "IEEE Standard for Smart Transducer Interface for Sensors and Actuators – Network Capable Application Processor (NCAP) Information Model," Mixed-Mode Communication Working Group of the Technical Committee on Sensor Technology TC-9 of the IEEE Instrumentation and Measurement Society, June 1999.

[2] Agilent Technologies, Inc. and Boeing Company, "1451.1 On-the-Wire Format for IP," Rev. 1, April 2002.

[3] Institute of Electrical and Electronics Engineers, Inc., "IEEE 1451.4/3.0 Standard for Smart Transducer Interface for Sensors and Actuators – Mixed-Mode Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats," Mixed-Mode Communication Working Group of the Technical Committee on Sensor Technology TC-9 of the IEEE Instrumentation and Measurement Society, New York: April 2004.

BIOGRAPHIES

Rebecca L. Oostdyk *specializes in software development and digital signal processing for intelligent sensors. She has developed hardware and software for NASA's Kennedy Space Center as a contractor for ASRC Aerospace. She has played a key role in developing the architecture for networked sensors, firmware for Kennedy Space Center's smart sensors, and software user interfaces to test and visualize the smart sensors. She has a BSEE from the University of Central Florida, where she graduated with honors and at the top of her class.*

Carlos T. Mata *received his Bachelor's degree from the Universidad Simón Bolívar, Venezuela, in 1993, and his Master's and Ph.D. from the University of Florida in 1997 and 2000, respectively. Dr. Mata is involved in the areas of computer modeling of different lightning processes and responses of power distribution systems to direct and nearby lightning strikes. Dr. Mata is currently a principal investigator and technical lead of ASRC's Advanced Technology Development Laboratory at Kennedy Space Center, where he specializes in embedded systems, advanced data acquisition systems, and smart-sensors design in support of the space program. He is author or coauthor of more than 12 journal publications and 15 technical reports.*

José M. Perotti *is the acting chief of the NASA Instrumentation Branch at Kennedy Space Center. The branch provides engineering and technical expertise to the Space Shuttle, International Space Station, Launch Services, and Exploration programs. Presently, Mr. Perotti is the principal investigator for several projects, including "Wireless Sensors and Networks," "Self-Validating Thermocouple System," and "NIST Traceable Portable Pressure Transfer Standard." He is the coprincipal investigator for "Integrated Intelligent Health Management System (IIHMS)," funded by the Office of Space Flight and led by Stennis Space Center. Mr. Perotti holds a B.S. in Electrical Engineering from the University of Miami and an M.S. in Engineering Management from the University of Central Florida. Mr. Perotti holds five patents for his work in instrumentation and control systems.*

APPENDIX A: ON-THE-WIRE FORMATS OF IEEE 1451.1 DATATYPES

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

IEEE 1451.1 Type	On-the-Wire Format
Boolean	0x00 for FALSE, 0x01 for TRUE
Enumeration member	h_0h_1, which represents the value of an enumeration member defined in 1451.
Float32	$h_0h_1h_2h_3h_4h_5h_6h_7$, where the bytes represent an 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. NaN is not allowed.
Float64	$h_0h_1h_2h_3h_4h_5h_6h_7h_8h_9h_{10}h_{11}h_{12}h_{13}h_{14}h_{15}$, where the bytes represent an ANSI/IEEE 754-1985 standard normalized 8-byte float representation. Exponent is 11 bits biased by 1023, and fractional is the fractional part of the mantissa and is 52 bits. NaN is not allowed.
Integer8	h_0h_1, where the integer's value is in two's complement
Integer16	$h_0h_1h_2h_3$, where the integer's value is in two's complement
Integer32	$h_0h_1h_2h_3h_4h_5h_6h_7$, where the integer's value is in two's complement
Integer64	$h_0h_1h_2h_3h_4h_5h_6h_7h_8h_9h_{10}h_{11}h_{12}h_{13}h_{14}h_{15}$, where the integer's value is in two's complement
Octet	h_0h_1, where the value is in hexadecimal
UInteger8	h_0h_1, where the integer's value is unsigned
UInteger16	$h_0h_1h_2h_3$, where the integer's value is unsigned
UInteger32	$h_0h_1h_2h_3h_4h_5h_6h_7$, where the integer's value is unsigned
UInteger64	$h_0h_1h_2h_3h_4h_5h_6h_7h_8h_9h_{10}h_{11}h_{12}h_{13}h_{14}h_{15}$, where the integer's value is unsigned

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

IEEE 1451.1 Type	On-the-Wire Format
BooleanArray	$n_0n_1n_2n_3h_0h_1\ldots h_{k-1}h_k$, where $n_0n_1n_2n_3$ is the number of elements in the array represented as a 16-bit unsigned integer and each h_ih_{i+1} is either 0x00 for FALSE or 0x01 for TRUE
Float32Array	$n_0n_1n_2n_3h_0h_1h_2h_3h_4h_5h_6h_7\ldots h_{k-7}h_{k-6}h_{k-5}h_{k-4}h_{k-3}h_{k-2}h_{k-1}h_k$, where $n_0n_1n_2n_3$ is the number of elements in the array represented as a 16-bit unsigned integer and each $h_ih_{i+1}h_{i+2}h_{i+3}h_{i+4}h_{i+5}h_{i+6}h_{i+7}$ represents an ANSI/IEEE 754-1985 standard normalized 4-byte float representation. NaN is not allowed.
Float64Array	$n_0n_1n_2n_3h_0h_1h_2h_3h_4h_5h_6h_7h_8h_9h_{10}h_{11}h_{12}h_{13}h_{14}h_{15}\ldots h_{k-15}h_{k-14}h_{k-13}h_{k-12}h_{k-11}h_{k-10}h_{k-9}h_{k-8}h_{k-7}h_{k-6}h_{k-5}h_{k-4}h_{k-3}h_{k-2}h_{k-1}h_k$, where $n_0n_1n_2n_3$ is the number of elements in the array represented as a 16-bit unsigned integer and each $h_ih_{i+1}h_{i+2}h_{i+3}h_{i+4}h_{i+5}h_{i+6}h_{i+7}$ represents an ANSI/IEEE 754-1985 standard normalized 8-byte float representation. NaN is not allowed.
Integer8Array	$n_0n_1n_2n_3h_0h_1\ldots\ h_{k-1}h_k$, where $n_0n_1n_2n_3$ is the number of elements in the array represented as a 16-bit unsigned integer and each h_ih_{i+1} represents an Integer8 array element.
Integer16Array	$n_0n_1n_2n_3h_0h_1h_2h_3\ldots h_{k-3}h_{k-2}h_{k-1}h_k$, where $n_0n_1n_2n_3$ is the number of elements in the array represented as a 16-bit unsigned integer and each $h_ih_{i+1}h_{i+2}h_{i+3}$ represents an Integer16 array element.
Integer32Array	$n_0n_1n_2n_3h_0h_1h_2h_3h_4h_5h_6h_7\ldots h_{k-7}h_{k-6}h_{k-5}h_{k-4}h_{k-3}h_{k-2}h_{k-1}h_k$, where $n_0n_1n_2n_3$ is the number of elements in the array represented as a 16-bit unsigned integer and each $h_ih_{i+1}h_{i+2}h_{i+3}h_{i+4}h_{i+5}h_{i+6}h_{i+7}$ represents an Integer32 array element.
Integer64Array	$n_0n_1n_2n_3h_0h_1h_2h_3h_4h_5h_6h_7h_8h_9h_{10}h_{11}h_{12}h_{13}h_{14}h_{15}\ldots\ h_{k-15}h_{k-14}h_{k-13}h_{k-12}h_{k-11}h_{k-10}h_{k-9}h_{k-8}h_{k-7}h_{k-6}h_{k-5}h_{k-4}h_{k-3}h_{k-2}h_{k-1}h_k$, where $n_0n_1n_2n_3$ is the number of elements in the array represented as a 16-bit unsigned integer and each $h_ih_{i+1}h_{i+2}h_{i+3}h_{i+4}h_{i+5}h_{i+6}h_{i+7}h_{i+8}h_{i+9}h_{i+10}h_{i+11}h_{i+12}h_{i+13}h_{i+14}h_{i+15}$ represents an Integer64 array element.

IEEE 1451.1 Type	On-the-Wire Format
OctetArray	$n_0n_1n_2n_3h_0h_1\ldots\ h_{k-1}h_k$, where $n_0n_1n_2n_3$ is the number of elements in the array represented as a 16-bit unsigned integer and each h_ih_{i+1} represents an Octet array element.
UInteger8Array	$n_0n_1n_2n_3h_0h_1\ldots\ h_{k-1}h_k$, where $n_0n_1n_2n_3$ is the number of elements in the array represented as a 16-bit unsigned integer and each h_ih_{i+1} represents a UInteger8 array element.
UInteger16Array	$n_0n_1n_2n_3h_0h_1h_2h_3\ldots\ h_{k-3}h_{k-2}h_{k-1}h_k$, where $n_0n_1n_2n_3$ is the number of elements in the array represented as a 16-bit unsigned integer and each $h_ih_{i+1}h_{i+2}h_{i+3}$ represents a UInteger16 array element.
UInteger32Array	$n_0n_1n_2n_3h_0h_1h_2h_3h_4h_5h_6h_7\ldots\ h_{k-7}h_{k-6}h_{k-5}h_{k-4}h_{k-3}h_{k-2}h_{k-1}h_k$, where $n_0n_1n_2n_3$ is the number of elements in the array represented as a 16-bit unsigned integer and each $h_ih_{i+1}h_{i+2}h_{i+3}h_{i+4}h_{i+5}h_{i+6}h_{i+7}$ represents a UInteger32 array element.
UInteger64Array	$n_0n_1n_2n_3h_0h_1h_2h_3h_4h_5h_6h_7h_8h_9h_{10}h_{11}h_{12}h_{13}h_{14}h_{15}\ldots\ h_{k-15}h_{k-14}h_{k-13}h_{k-12}h_{k-11}h_{k-10}h_{k-9}h_{k-8}$ $h_{k-7}h_{k-6}h_{k-5}h_{k-4}h_{k-3}h_{k-2}h_{k-1}h_k$, where $n_0n_1n_2n_3$ is the number of elements in the array represented as a 16-bit unsigned integer and each $h_ih_{i+1}h_{i+2}h_{i+3}h_{i+4}h_{i+5}h_{i+6}h_{i+7}$ $h_{i+8}h_{i+9}h_{i+10}h_{i+11}h_{i+12}h_{i+13}h_{i+14}h_{i+15}$ represents a UInteger64 array element.

Table 3. On-the-Wire Format for Fixed Length OctetArrays

IEEE 1451.1 Type	On-the-Wire Format
Units	$n_0n_1n_2n_3h_0h_1h_2h_3h_4h_5h_6h_7h_8h_9h_{10}h_{11}h_{12}h_{13}h_{14}h_{15}h_{16}h_{17}h_{18}h_{19}$, where $n_0n_1n_2n_3$ is the number 10 represented as a 16-bit unsigned integer and $h_0\ldots h_{19}$ is an array of 10 octets representing physical units per IEEE 1451.2.
UnitsArray	$n_0n_1n_2n_3u_0\ldots u_k$, where $n_0n_1n_2n_3$ is the number of Units array elements represented as a 16-bit unsigned integer and u_i is the i^{th} Units element in the array represented as an on-the-wire Units.
PubSubDomain	$n_0n_1n_2n_3h_0h_1h_2h_3h_4h_5h_6h_7h_8h_9h_{10}h_{11}h_{12}h_{13}h_{14}h_{15}$, where $n_0n_1n_2n_3$ is the number 8 represented as a 16-bit unsigned integer and $h_i\ldots h_{15}$ is an array of 8 octets representing the PubSubDomain datatype.

Table 4. On-the-Wire Format for IEEE 1451 String and StringArrays

IEEE 1451.1 Type	On-the-Wire Format
String	$s_0s_1c_0c_1l_0l_1n_0n_1n_2n_3h_0h_1\ldots h_{k-1}h_k$, where s_0s_1 is a value from the enumeration StringCharacterSet, c_0c_1 is a value from the enumeration StringCharacterCode, l_0l_1 is a value from the enumeration StringLanguage, $n_0n_1n_2n_3$ is the number of octets of String data represented as a 16-bit unsigned integer, and each h_ih_{i+1} is an octet of String data.
StringArray	$n_0n_1n_2n_3s_0\ldots s_k$, where $n_0n_1n_2n_3$ is the number of String array elements represented as a 16-bit unsigned integer, and s_i is the i^{th} String element in the array represented as an on-the-wire String.

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

IEEE 1451.1 Type	typedef
BitSequence	OctetArray
ClassID	OctetArray
ClientServerReturnCode	UInteger32
ObjectID	OctetArray

IEEE 1451.1 Type	typedef
ObjectTag	OctetArray
OpReturnCode	UInteger16
PublicationTopic	OctetArray
SubscriptionQualifier	OctetArray

Table 6. On-the-Wire Format for IEEE 1451.1 BitSequenceArray and ObjectTagArray Datatypes

IEEE 1451.1 Type	On-the-Wire Format
BitSequenceArray	$n_0n_1n_2n_3b_0\ldots b_k$, where $n_0n_1n_2n_3$ is the number of BitSequence elements in the array represented as a 16-bit unsigned integer and b_i is the i^{th} BitSequence in the array given in the BitSequence on-the-wire format.
ObjectTagArray	$n_0n_1n_2n_3t_0\ldots t_k$, where $n_0n_1n_2n_3$ is the number of ObjectTag elements in the array represented as a 16-bit unsigned integer and t_i is the i^{th} ObjectTag in the array given in the ObjectTag on-the-wire format.

Table 7. On-the-Wire Format for IEEE 1451.1 Structures

IEEE 1451.1 Type	Fields	On-the-Wire Format
ObjectDispatchAddress	ipAddress	UInteger32
	portNumber	UInteger16
	objectIdentifier	UInteger32
TimeRepresentation	seconds	UInteger32
	nanoseconds	Integer32
TimeInterval	startTime	
	seconds	UInteger32
	nanoseconds	Integer32
	deltaTime	
	seconds	UInteger32
	nanoseconds	Integer32
ObjectProperties	objectDispatchAddress	ObjectDispatchAddress
	objectTag	OctetArray
	owningBlockObjectTag	OctetArray
	objectName	String
	blockCookie	UInteger32
ClientPortProperties	clientPortDispatchAddress	ObjectDispatchAddress
	clientPortObjectTag	OctetArray
	clientPortName	String
	serverDispatchAdress	ObjectDispatchAddress
	serverObjectTag	OctetArray

IEEE 1451.1 Type	Fields	On-the-Wire Format
PublisherInformation	publicationID	UInteger16
	portObjectTag	OctetArray
	publicationChangeID	UInteger16
Uncertainty	interpretation	Enum value (UInteger8)
	uncertaintyValue	Float32
	coverageFactor	Float32
	customFactor	Float32

Table 8. On-the-Wire Format for IEEE 1451.1 Arrays of Structures

IEEE 1451.1 Type	On-the-Wire Format
ClientPortPropertiesArray	$n_0n_1n_2n_3c_0\ldots c_k$, where $n_0n_1n_2n_3$ is the number of ClientPortProperties elements in the array represented as a 16-bit unsigned integer and c_i is the i^{th} ClientPortProperties in the array given in the ClientPortProperties on-the-wire format.
ObjectPropertiesArray	$n_0n_1n_2n_3o_0\ldots o_k$, where $n_0n_1n_2n_3$ is the number of ObjectProperties elements in the array represented as a 16-bit unsigned integer and o_i is the i^{th} ObjectProperties in the array given in the ObjectProperties on-the-wire format.
TimeIntervalArray	$n_0n_1n_2n_3t_0\ldots t_k$, where $n_0n_1n_2n_3$ is the number of TimeInterval elements in the array represented as a 16-bit unsigned integer and t_i is the i^{th} TimeInterval in the array given in the TimeInterval on-the-wire format.
TimeRepresentationArray	$n_0n_1n_2n_3t_0\ldots t_k$, where $n_0n_1n_2n_3$ is the number of TimeRepresentation elements in the array represented as a 16-bit unsigned integer and t_i is the i^{th} TimeRepresentation in the array given in the TimeRepresentation on-the-wire format.
UncertaintyArray	$n_0n_1n_2n_3u_0\ldots u_k$, where $n_0n_1n_2n_3$ is the number of Uncertainty elements in the array represented as a 16-bit unsigned integer and u_i is the i^{th} Uncertainty element in the array given in the Uncertainty on-the-wire format.

Table 9. On-the-Wire Format for IEEE 1451.1 Physical Parameter Types

IEEE 1451.1 Type	Fields	On-the-Wire Format
PhysicalParameterSingletonData	discriminant	Enum value (UInteger8)
	timestamp	TimeRepresentation
	value	ArgumentArray
PhysicalParameterSeriesData	discriminant	Enum value (UInteger8)
	timestamp	TimeRepresentation
	abscissaIncrement	Argument
	abscissaOrigin	Argument
	value	ArgumentArray
ScalarAnalogMetadata	discriminant	Enum value (UInteger8 = 0)
	parameterInterpretation	Enum value (UInteger8)
	buffering	Enum value (UInteger8)
	datatype	Enum value (UInteger8)

IEEE 1451.1 Type	Fields	On-the-Wire Format
	units	Units
	parameterName	String
	upperLimit	Argument
	lowerLimit	Argument
	uncertainty	Uncertainty
ScalarDiscreteMetadata	discriminant	Enum value (UInteger8 = 1)
	parameterInterpretation	Enum value (UInteger8)
	buffering	Enum value (UInteger8)
	datatype	Enum value (UInteger8)
	units	Units
	parameterName	String
	upperLimit	Argument
	lowerLimit	Argument
ScalarDigitalMetadata	discriminant	Enum value (UInteger8 = 2)
	parameterInterpretation	Enum value (UInteger8)
	buffering	Enum value (UInteger8)
	datatype	Enum value (UInteger8)
	rightJustifiedFlag	Boolean
	units	Units
	parameterName	String
	numberOfOctets	UInteger16
	numberOfSiginificantBits	UInteger16
ScalarAnalogSeriesMetadata	discriminant	Enum value (UInteger8 = 3)
	parameterInterpretation	Enum value (UInteger8)
	buffering	Enum value (UInteger8)
	datatype	Enum value (UInteger8)
	units	Units
	abscissaUnits	Units
	parameterName	String
	upperLimit	Argument
	lowerLimit	Argument
	abscissaIncrement	Argument
	abscissaOrigin	Argument
	abscissaIncrementUncertainty	Uncertainty
	abscissaOriginUncertainty	Uncertainty

IEEE 1451.1 Type	Fields	On-the-Wire Format
	uncertainty	Uncertainty
ScalarDiscreteSeriesMetadata	discriminant	Enum value (UInteger8 = 4)
	parameterInterpretation	Enum value (UInteger8)
	buffering	Enum value (UInteger8)
	datatype	Enum value (UInteger8)
	units	Units
	abscissaUnits	Units
	parameterName	String
	upperLimit	Argument
	lowerLimit	Argument
	abscissaIncrement	Argument
	abscissaOrigin	Argument
	abscissaIncrementUncertainty	Uncertainty
	abscissaOriginUncertainty	Uncertainty
ScalarDigitalSeriesMetadata	discriminant	Enum value (UInteger8 = 5)
	rightJustifiedFlag	Boolean
	parameterInterpretation	Enum value (UInteger8)
	buffering	Enum value (UInteger8)
	datatype	Enum value (UInteger8)
	units	Units
	abscissaUnits	Units
	parameterName	String
	abscissaIncrement	Argument
	abscissaOrigin	Argument
	abscissaIncrementUncertainty	Uncertainty
	abscissaOriginUncertainty	Uncertainty
	numberOfOctets	UInteger16
	numberOfSignificantBits	UInteger16
VectorAnalogMetadata	discriminant	Enum value (UInteger8 = 6)
	parameterInterpretation	Enum value (UInteger8)
	buffering	Enum value (UInteger8)
	coordinateSystem	Enum value (UInteger8)
	dimension	UInteger16
	datatype	Array of enums (UInteger8Array)
	units	UnitsArray

IEEE 1451.1 Type	Fields	On-the-Wire Format
	parameterName	String
	upperLimit	ArgumentArray
	lowerLimit	ArgumentArray
	uncertainty	UncertaintyArray
VectorDiscreteMetadata	discriminant	Enum value (UInteger8 = 7)
	parameterInterpretation	Enum value (UInteger8)
	buffering	Enum value (UInteger8)
	coordinateSystem	Enum value (UInteger8)
	dimension	UInteger16
	datatype	Array of enums (UInteger8Array)
	units	UnitsArray
	parameterName	String
	upperLimit	ArgumentArray
	lowerLimit	ArgumentArray
VectorDigitalMetadata	discriminant	Enum value (UInteger8 = 8)
	parameterInterpretation	Enum value (UInteger8)
	buffering	Enum value (UInteger8)
	coordinateSystem	Enum value (UInteger8)
	dimension	UInteger16
	datatype	Array of enums (UInteger8Array)
	units	UnitsArray
	parameterName	String
	numberOfOctets	UInteger16
	numberOfSiginificantBits	UInteger16
	rightJustifiedFlag	BooleanArray
VectorAnalogSeriesMetadata	discriminant	Enum value (UInteger8 = 9)
	parameterInterpretation	Enum value (UInteger8)
	buffering	Enum value (UInteger8)
	coordinateSystem	Enum value (UInteger8)
	dimension	UInteger16
	datatype	Array of enums (UInteger8Array)
	units	UnitsArray
	parameterName	String
	abscissaUnits	Units
	abscissaIncrement	Argument

IEEE 1451.1 Type	Fields	On-the-Wire Format
	abscissaOrigin	Argument
	abscissaIncrementUncertainty	Uncertainty
	abscissaOriginUncertainty	Uncertainty
	upperLimit	ArgumentArray
	lowerLimit	ArgumentArray
	uncertainty	UncertaintyArray
VectorDiscreteSeriesMetadata	discriminant	Enum value (UInteger8 = 10)
	parameterInterpretation	Enum value (UInteger8)
	buffering	Enum value (UInteger8)
	coordinateSystem	Enum value (UInteger8)
	dimension	UInteger16
	datatype	Array of enums (UInteger8Array)
	units	UnitsArray
	parameterName	String
	abscissaUnits	Units
	abscissaIncrement	Argument
	abscissaOrigin	Argument
	abscissaIncrementUncertainty	Uncertainty
	abscissaOriginUncertainty	Uncertainty
	upperLimit	ArgumentArray
	lowerLimit	ArgumentArray
VectorDigitalSeriesMetadata	discriminant	Enum value (UInteger8 = 11)
	parameterInterpretation	Enum value (UInteger8)
	buffering	Enum value (UInteger8)
	coordinateSystem	Enum value (UInteger8)
	dimension	UInteger16
	datatype	Array of enums (UInteger8Array)
	units	UnitsArray
	parameterName	String
	abscissaUnits	Units
	abscissaIncrement	Argument
	abscissaOrigin	Argument
	abscissaIncrementUncertainty	Uncertainty
	abscissaOriginUncertainty	Uncertainty
	numberOfOctets	UInteger16Array

IEEE 1451.1 Type	Fields	On-the-Wire Format
	numberOfSignificantBits rightJustifiedFlag	UInteger16Array BooleanArray

Table 10. On-the-Wire Format for IEEE 1451.1 Arrays of Physical Parameter Types

IEEE 1451.1 Type	On-the-Wire Format
PhysicalParameterDataArray	$n_0n_1n_2n_3p_0\ldots p_k$, where $n_0n_1n_2n_3$ is the number of PhysicalParameterData elements in the array represented as a 16-bit unsigned integer and p_i is the i_{th} PhysicalParameterData element in the array given in the PhysicalParameterData on-the-wire format.
PhysicalParameterMetadataArray	$n_0n_1n_2n_3p_0\ldots p_k$, where $n_0n_1n_2n_3$ is the number of PhysicalParameterMetadata elements in the array represented as a 16-bit unsigned integer and p_i is the i^{th} PhysicalParameterMetadata element in the array given in the PhysicalParameterMetadata on-the-wire format.

Table 11. On-the-Wire Format for IEEE 1451.1 Argument and ArgumentArray Datatypes

IEEE 1451.1 Type	On-the-Wire Format
Argument	$d_0d_1u_0$, where d_0d_1 is the discriminant of the Argument given as a TypeCode enumeration value and u_0 is the value of the union given in the on-the-wire format of the datatype represented by the TypeCode enumeration value.
ArgumentArray	$n_0n_1n_2n_3a_0\ldots a_k$, where $n_0n_1n_2n_3$ is the number of Argument elements in the array represented as a 16-bit unsigned integer and a_i is the i^{th} Argument element in the array given in the Argument on-the-wire format.

APPENDIX B: ON-THE-WIRE FORMATS OF IEEE 1451.1 MESSAGES

Table 12. On-the-Wire Format for IEEE 1451.1 Publication Messages

Byte Number	Datatype	On-the-Wire Format	Interpretation
0	UInteger8	F7	Magic number for 1451.1 publication (0xF7)
1-2	UInteger16	UInteger16	Total message length
3-4	UInteger16	UInteger16	Total header length
5	Enumeration Member	UInteger8	Publication Key
6-13	Octet[8]	$h_0h_1h_2h_3h_4h_5h_6h_7$ $h_8h_9h_{10}h_{11}h_{12}h_{13}h_{14}h_{15}$	Publication Domain
14…k	Octet[k-14]	$h_{14}h_{14}…h_kh_k$	Publication Topic, where the only the Octets in the OctetArray are given, and a null Publication Topic is given by 0x00
K+1…n	ArgumentArray	ArgumentArray	(optional) contents of the publication

Table 13. On-the-Wire Format for IEEE 1451.1 Client-to-Server Messages

Byte Number	Datatype	On-the-Wire Format	Interpretation
0	UInteger8	ED	Magic number for 1451.1 client-to-server message (0xed)
1-2	UInteger16	UInteger16	Total message length
3-4	UInteger16	UInteger16	Client Port's cached BlockCookie
5-8	UInteger32	UInteger32	The id of the target server object (serverObjectIdentifier)
9-10	UInteger16	UInteger16	Operation id of the desired operation on the target server object
11	Enumeration Member	UInteger8	Execute mode (EM_RETURN_VALUE, EM_NO_RETURN_VALUE, EM_ASYNCHRONOUS)
12…k	ArgumentArray	ArgumentArray	(optional) input arguments for target server object

Table 14. On-the-Wire Format for IEEE 1451.1 Server-to-Client Messages

Byte Number	Datatype	On-the-Wire Format	Interpretation
0	UInteger8	D5	Magic number for 1451.1 server-to-client message (0xd5)
1-2	UInteger16	UInteger16	Total message length
3-6	UInteger32	UInteger32	ClientServerReturnCode
7-8	UInteger16	UInteger16	Server object's current Block-Cookie
9	Enumeration Member	UInteger8	Execute mode (EM_RETURN_VALUE, EM_NO_RETURN_VALUE, EM_ASYNCHRONOUS)
10-13	UInteger32	UInteger32	The ID of the target server object (serverObjectIdentifier)
14-15	UInteger16	UInteger16	Operation ID of the desired operation on the target server object
16…k	ArgumentArray	ArgumentArray	(optional) output arguments from the target server object