Advanced Middleware for Space Plug-and-play Avionics (SPA) Architecture Standardization and Rapid Systems Integration

Grant K. Holcomb
ATK Launch Systems
MS-130 P.O. Box 707 Brigham City, UT 84302-0707; 435-640-0072
Grant.Holcomb@ATK.com

ABSTRACT

The Space Plug-and-play Avionics (SPA) architecture is the foundation of the U.S. Department of Defense (DoD) Operationally Responsive Space (ORS) initiative. The ORS mission reflects the need to provide our warfighters, battlefield commanders, and first responders with reliable communications and real-time situational awareness. Driven by operational requirements, the ability to rapidly and economically manufacture, assemble, and launch custom satellite configurations is essential. The most viable solution for delivering the software required to solve this problem is for all approved DoD vendors to have equal access to software middleware that possesses the characteristics to establish a standard for Plug-and-Play (PnP) for the broadest range of transducers (sensors and actuators) and third party algorithms (heritage and future flight systems software and mission control processes). This middleware must support all ORS goals to include fault tolerance, computing topology independence, common data understanding, network topology independence, operating system independence, autonomous processing, and automation of PnP processes for hardware and software sub-components. A middleware candidate will be presented that has the potential to (1) establish industry wide standardization, (2) accelerate satellite system integration, (3) increase reliability and survivability of satellites, and (4) measurably support ORS goals in a cost effective fashion.

INTRODUCTION

The SPA architecture is the foundation of the U.S. DoD ORS initiative. The ORS mission reflects the need to provide our warfighters, battlefield commanders, first responders, and crisis managers with reliable communications and real-time situational awareness. In the ORS environment, a satellite's mission and the mission's scope and duration are not known in advance. Driven by operational requirements, the ability to rapidly and economically manufacture, assemble, and launch custom satellite configurations is essential. Rapidly is defined by ORS as "weeks and months not decades." Satellite based operational support can range from delivering high-resolution imagery and communications to a wide array of advanced capabilities currently under development (surveillance, threat identification, resource tracking, targeting, etc.). The complexity of this problem is compounded by an additional ORS requirement, which is to allow authorized personnel on the ground to be informed of the availability and capabilities of ORS satellites and to provide these users with direct, secure, real-time access to the respective satellite services.

BACKGROUND

Air Force Research Laboratories (AFRL) has led the SPA design effort on behalf of the ORS Office. The current SPA reference design is being developed with the support of the commercial space industry. Still in draft form, a comprehensive array of SPA standard specification documents and reference design input has been produced.[1-5] The design and organization of the modular hardware sub-components (bus architecture, power management, electrical interfaces, sensors, actuators, etc.) reflect decades of proven satellite design experience and flight heritage. However, the primary barrier to success of SPA has not been sufficiently addressed. Satellite software is currently the proprietary intellectual property of the respective vendors, which build and maintain at great expense large complex proprietary code libraries. The state-of-the-art in commercially available software development libraries cannot support ORS requirements for many reasons, including the lack of viable building blocks for delivering PnP. In response to similar issues developing flight system software, ATK has invested in maturing platform independent middleware, which surfaces the power of PnP in a measurable fashion and the potential of establishing the standardization required to support ORS mission accomplishment.

STATEMENT OF THE PROBLEM

All sub-components of a satellite system must possess an interface that facilitates its integration into a coherent solution. A sub-component's interface is defined by the data it provides or consumes, the notifications it generates, the commands it accepts, and

the services it delivers. A service is defined as data delivered in response to a command. From the system software perspective there should be no distinction between the interface of a software sub-component (application, algorithm, or function) and a hardware sub-component (sensor, actuator, or other device). It is also essential to normalize/standardize data and control flow between interfaces within a system to establish efficient and reliable operation. Vendor specific nomenclature and proprietary data formats must not be propagated within a system. This is referred to as establishing a common data understanding.

The pinnacle of system software development is delivering the ability to discover any sub-component's interface and utilize its data, notifications, commands and services in an automated fashion. This is also the definition of PnP. When a sub-component is introduced to the system (e.g., power-up, scheduled activation, etc.) its interface should be automatically discovered and its data, notifications, commands, and services seamlessly integrated into supporting mission accomplishment. Unfortunately, the way we currently define data structures and software objects presents a significant barrier to achieving the desired level of interface discovery automation.

Most of the complexity, labor hours, and problems incurred when developing a satellite system revolves around defining, implementing, and testing sub-component interfaces. More specifically, creating custom interfaces for proprietary sub-components or creating translators or converters for integrating dissimilar interfaces. The primary barrier to success is how software developers implement an interface, particularly without the guidance of a standard. In an object-oriented development environment the "class" is the basic building block, which is a logic grouping of member variables and methods. The member variables contain the atomic data values, which are the smallest data elements managed by the software. The methods are the smallest units of executable computer instructions responsible for accomplishing specific tasks. An efficient class design assumes specific responsibilities that can be reused within an implementation by grouping the appropriate member variables and methods within its definition. Typically, the member variables of a class use data types defined by the respective programming language (C++, C#, Objective-C, Java, etc.). Unfortunately, this common approach directly increases the complexity, time, and cost of automating interface discovery.

To clarify this observation, Figure 1 presents the definition of a C++ programming language class named "Person." The four member variables defined in this class use C++ programming language-defined types.

```
class Person {
        Person();

        std::string     mFirstName;
        std::string     mLastName;
        int             mAge;
        bool            mEmployee;
};
```

Figure 1: Example C++ Class Definition

As defined by published standards for object-oriented programming, the member variables in Figure 1 are intentionally encapsulated (hidden) from both the sub-component and the system. In fact, in this specific case these four variables could only be accessed by methods that must be defined within the class itself. Even if these variables were declared differently to make them more accessible by other software objects, the programming language does not provide a dynamic (runtime) mechanism that enables an external process to automatically discover their respective name, type, or format. Any methods that are defined within this class, which are required to implement notifications, commands, and services are equally hidden or inaccessible. This takes us to the bottom line problem—*there is no standard platform independent mechanism that enables the automated discovery of the member variables and methods used to define the interface of a system's sub-components.*

To compound this issue, current software approaches that do facilitate remote interface discovery are either tied in a proprietary fashion to a specific operating system, processor, or network combination or comprise multiple different processes to support the type of sub-component or how it is interconnected with the system. Software sub-components implemented internal to a single computing device (running in the same memory space) use one type of interface discovery. Hardware sub-components connected to a network use a different process. In other words, system software developers are forced to use different approaches and code libraries to integrate hardware and software sub-components, particularly when there are different computing and network platforms used within the same system. Additionally, these software approaches require so much processing power and memory that they cannot operate in an embedded processor environment (like those currently used on small satellites).

The answer to these technical issues is middleware that includes the right combination of platform independent processes that makes the interface of any system sub-component discoverable in an automated fashion from

anywhere on a network. The answer to the current issue of a lack of standards in this area is vastly more complex. However, making the middleware available to all approved vendors in a well supported, equitable, and cost effective fashion holds great promise for establishing a standard necessary to support ORS goals.

All of this implies the need to define middleware and its benefits and to summarize the characteristics that a middleware solution must possess to support ORS goals. These middleware characteristics include, but are not limited to the following:

- Discoverable Variables, Methods and, Objects
- Hardware Independence
- Operating System Independence
- Signaling, Journaling, Simulations, and Quality Assurance
- Remote Process Monitoring and Control
- Data Modeling

MIDDLEWARE DEFINITION

From an implementation perspective, middleware is a library of software building blocks that a programmer uses to develop system software. However, think of middleware as the layer between the hardware (processors, routers, sensor, actuators, storage devices, etc.) and all system specific software. If the roads in a city where a system and vehicles were the data, the middleware would be a citywide interconnected traffic light system that ensures all traffic flows smoothly and efficiently.

It is important to note that middleware building blocks are intentionally designed to be generally applicable for solving a broad range of system software problems. For example, the base PnP process available in the middleware is only applicable for generic interface discovery. This base capability must receive additions to support the interface specifications established by ORS through the SPA reference design. Because middleware is engineered to be an abstraction layer, adding to the base PnP process is as straight forward as declaring additional variables within the right class definitions under control of a common data dictionary. The middleware and these additions can be established as part of a standard in support of meeting ORS goals.

MIDDLEWARE CHARACTERISTICS

The following sections summarize the characteristics middleware must possess to facilitate PnP and establish a standard for fully supporting ORS small satellite mission requirements.

Discoverable Variables, Methods, and Objects

In Figure 2, a standard C++ class definition is compared to a middleware-based C++ class definition. From a standard C++ compiler perspective both compile the same and produce fully compliant type checked object-oriented constructs. The first difference is that the language defined atomic variable types are replaced with equivalent middleware defined variable types (colored in blue or bold). Herein, middleware defined member variable types are called “attributes” to differentiate them from the programming language defined variable types.

Standard C++ Class		Middleware-Based C++ Class	
class Person {		class Person : *public Node* {	
Person();		Person();	
std::string	mFirstName;	**StringValue**	mFirstName;
std::string	mLastName;	**StringValue**	mLastName;
int	mAge;	**IntegerValue**	mAge;
bool	mEmployee;	**BooleanValue**	mEmployee;
};		};	

Figure 2: Standard C++ Class Compared to Middleware-based C++ Class

The second difference is that the Person class inherits from Node (colored in green or italic), which is a base class provided by the middleware. The Node class implements a platform independent signaling and communication capabilities required for dynamic discovery. In fact, all of the attributes declared in the Figure 2 middleware-based C++ class definition are also derived from the Node class and also inherit the platform independent signaling and communication capabilities. *Think of Node as the foundation for establishing a standard that transforms generic variables and objects into discoverable and manageable system wide network-centric building blocks.*

Attributes and Node based objects inherit a remotely discoverable interface, which includes a name, type and numerous generic notification, command, and service capabilities. Additionally, attributes and Nodes can send a signal to registered listeners when any changes occur. *The critical observation here is that this approach transforms the basic building blocks of all systems and its atomic data types, into dynamically discoverable and controllable sub-components regardless of where they reside on a network.* This is in stark contrast to virtually all current approaches to system software development where every software algorithm or application is a black box that potentially hides performance problems, flaws, incompatibilities, inconsistencies, and most importantly the interfaces required for efficient system wide operation. This

approach has numerous beneficial side effects as presented in Figure 3.

Inherent and Supported Functionality	C++ Object	Middleware C++ Object
Remote Discovery	No	Yes
Remote Instantiation	No	Yes
Remote Modification	No	Yes
Remote Change Notification	No	Yes
Metadata Formatted Persistence and Signaling	No	Yes
Hierarchical Data Modeling	No	Yes
Distributed Data Modeling	No	Yes
Redundant Data Modeling	No	Yes

Figure 3: Comparison of Inherent and Supported Functionalities

Figure 3 presents a partial listing of inherent and supported functionality supported by a current middleware implementation. For example, the ability to remotely establish (instantiate) an object is what makes Java such an attractive offering. However, it can now be accomplished without Java's significant overhead or performance limitations. The middleware extends the capabilities of the C++ programming language without undermining or crippling its strengths, reducing its expressiveness, or forcing programmers to learn entirely new implementation methodologies. By simply using middleware defined member variable types (attributes) and defining classes that inherit from the Node base class the foundation is established for delivering system wide automated interface discovery.

Hardware Independence

The software used in the SPA solution must not be dependent upon any specific processor, memory, storage, bus, or network technology. The power, weight, capacity, reliability, electromagnetic sensitivity, performance, and radiation-tolerance/hardness in these areas can and must be continuously improved. Changing these hardware sub-components should not require any significant portion of the SPA software to be rewritten. A standard for how to logically address remote devices or specific memory or storage locations, in an environment where the addressing mechanisms are different for each technology type, is not currently specified in the SPA documentation. For example, the software interfaces for managing the transmission and reception of data should not be dependent upon the chosen bus or network technology (SpaceWire, Ethernet, USB, etc.). This implies that a middleware solution must provide well-defined hardware abstraction layers including a mechanism that uniquely identifies every software object in a SPA implementation, regardless of the memory-addressing scheme used by different processor or memory architectures. This facilitates continuous improvement of the SPA hardware sub-components independently of the software. Additionally, the middleware's object-to-object communication process must completely hide the specific characteristics of the network interface or topology. In the middleware, these abstraction layers are called "drivers", which represent the logical interface point between the "normalized" processes and any proprietary processes or technologies.

An extremely beneficial side effect has been observed through the approach of cleanly abstracting the memory addressing, network interface, and storage device access through a single common code implementation. Regardless of the number of processors and processor types (size and different types of memory) and different types of storage components resident in a SPA implementation, the system becomes a single managed virtual computing asset. This represents the ideal foundation for creating a redundant and distributed architecture capable of handling a specified level of reliability and performance while also delivering precise control over resource consumption (processing capacity, network bandwidth, power, data storage capacity, etc.).

This middleware characteristic directly supports the ORS requirement to implement network technology independent and network topology independent solutions. It also supports current and future network and bus technologies interconnected to achieve any required level of network path redundancy.

Operating System Independence

The number of lines of source code used to implement key SPA features is published in the current AFRL SPA reference documents.[6] However, they do not reflect the largest and most complex software component—the operating system required by each processor.

Commercially available embedded real-time operating systems must provide a broad array of software support modules in order to win market share. It is very tempting for the typical programmer to use these features, which unfortunately binds the software deliverables to the processor/operating system combination. However, the code that runs on a SPA satellite cannot have any direct dependency on any operating system functionality. When a SPA implementation switches to a different processor a different operating system may be required. At

minimum, a common operating system driver needs to be defined and published to ensure that SPA software has no direct dependence on any operating system. The middleware must include the necessary drivers to support this SPA requirement. However, the middleware implementation should go even further by demonstrating that an operating system is not even required in the SPA design.

Even the most mature flight heritage "certified" embedded real-time operating system (RTOS) represents a very large amount of source code and a significant reliability and quality assurance challenge. Even the optimal RTOS has a negative impact on configuration management, memory usage, complexity, reliability, performance, power consumption, etc. In an embedded environment, all that should be required for the middleware to work on any processor is access to its process thread scheduling mechanism. This is possible if the middleware reduces all process execution to a common signaling mechanism. This is the foundation of the attribute and Node discovery capability previously summarized. In the embedded industry the process thread scheduling mechanism is called a "Super Simple Tasker" (SST),[7] which can be implemented in only a few hundred lines of source code. This sits underneath the aforementioned middleware operating system driver so that different SSTs required for different processor architectures can be independently supported. For example, the SST for a single core processor is very different from an 8-core processor. The middleware can take full advantage of both types of processors through its operating system driver-based abstraction layer. This approach represents a radical simplification of the SPA design while directly contributing to ORS requirements.

Middleware operating system independence directly supports the ORS requirement of achieving computing topology independence. The ability to mix single and multiple-core processors from different vendors, as required to host distributed and redundant processes in a cost effective fashion, holds significant potential for delivering future advanced satellite mission capabilities.

Signaling, Journaling, Simulations, and Quality Assurance

Typically, the four characteristics of how to implement a common signaling (event or messaging) mechanism, how to log signals (journaling), how to simulate a design, and how to implement robust quality assurance processes are all treated as separate issues and actions during a development effort. They are all extremely important. However, the right combination of middleware characteristics has the ability to integrate these elements in a seamless fashion into the foundation of a system software implementation.

As summarized earlier, Nodes can "listen" to the state of other Nodes and attributes. These listening relationships are easily established by the software developers with simple middleware provided calls. Using the common platform independent signaling mechanism, the instant a Node or attribute changes it notifies all established listening Nodes by sending them a signal. Through well defined abstraction layers, these signals can be in many formats.

In low bandwidth environments the signal format can be an efficient binary encoding. The format can even be changed on demand during system operation. One critical signal format, particularly during the design, development, and test phases of a system, is the use of the Extensible Markup Language (XML), which wraps each data element with metadata. In other words, each data item includes additional information regarding what it is. XML is human readable ASCII text allowing a developer or quality assurance specialist to read recoded signals and to some degree make sense of their respective purpose. XML also facilitates highly automated analysis and pattern recognition. For example, a signal that says "<Age>49</Age>" eliminates significant ambiguity regarding the data element "49" both from the perspective of human understanding and from automating analysis. In fact, in a standards-based system environment, which should possess a common data understanding backed by a common data dictionary, the tag "Age" would speak directly to the issue of determining things like format, precision, and initial, minimum and maximum values.

An XML formatted message sent by an attribute or Node to notify a listening Node that it has changed can be persisted. The persistence process (recording signals to a storage device) creates what is called a journal. The metadata format supports highly automated analysis of a journal to surface issues or measure performance metrics. Subsequently, the journal is the foundation of a critical system capability called a feedback loop. For example, regardless of the number or complexity of processes that run during system operation, the journal can be used to measure every aspect of performance, timing, and reliability. In fact, a properly implemented middleware signaling mechanism should be symmetrical, which means a journal can be modified and "played back" into the system. This reflects that a middleware-based solution can be self-simulating. An internal simulation capability has an extraordinarily beneficial impact on the cost, time, and effectiveness of a quality assurance process. This represents the potential for testing an entire SPA system

implementation in a Monte Carlo fashion to surface issues well before operational deployment.

The combination of signaling, journaling, and self-simulation provided by a middleware standard directly contributes to the primary ORS goals of accelerating satellite system integration, delivering solutions in a cost effective fashion, and increasing reliability and survivability of satellites.

Remote Process Monitoring and Control

The seemingly simple process of determining when a specific variable changes and responding to those changes currently represents multiple layers of complexity, particularly when implemented across a network. This functionality is required to facilitate remote process monitoring and control. A critical characteristic of the middleware in the SPA environment must be to provide a robust mechanism for monitoring the variables that define a process. This must also include the ability to change the values stored in these variables, which is the foundation of providing process control.

The middleware-based signaling mechanism facilitates the ability for objects to remotely "listen" to and change the attributes encapsulated within other objects resident anywhere within a SPA implementation. Subsequently, the same objects inherit the ability to remotely monitor and control processes to any required degree. This represents the ability to deliver true system-wide operational visibility and situational awareness.

It is important to note that the remote process and control capability provided by the middleware is the foundation for many other critical capabilities. For example, it is not possible to implement fault tolerance unless the applicable processes are notified (signaled) that a problem has occurred and that all applicable processes can be controlled precisely enough to facilitate failure recovery. This capability presents a significant technological breakthrough essential for ORS mission success.

Data Modeling

The desired middleware must support the delivery of pure data driven solutions. In other words, changes to the data used to model a process must result in the immediate or real-time execution of applicable code. From a software perspective, SPA is a data flow problem that must be properly modeled for successful mission accomplishment. Creating the data structures and subsequent interfaces necessary to accurately model a problem this complex is easily one of the most difficult and time consuming development tasks; regardless of the tools, experience, or level of engineering resources that are available. The issue lies in correctly defining the right combination of attributes and their relationships as necessary to sufficiently model the desired behavior.

Human beings do not know what they do not know and it is guaranteed that an initial deliverable of a complex data model will be wrong. The number of variables and combination of relationships is extraordinary. This problem also implies that the software architecture must be dynamically configurable and extensible in combination with a feedback loop that facilitates the ability to "learn" from mistakes. This is not a capability inherent in commercially available software development frameworks, but is important to ORS mission accomplishment.

The middleware must include the required data modeling processes by combining all of the capabilities summarized in the previous sections with a very critical additional ingredient. Middleware defined base objects must include the ability to be linked on demand at runtime into hierarchies of any complexity (lists, trees, graphs, and cyclic graphs). Figure 4 shows a "cyclic graph" relationship between Node-based software objects (i.e., Nodes 1 through 10). A graph is called cyclic when signals can be propagated in a loop (blue or curved lines shown in Figure 4). Note the direction of the arrowheads on the lines between the Nodes. This reflects that signals can be sent in either direction at the same time. Tree and list sub-relationships are also evident in the figure.

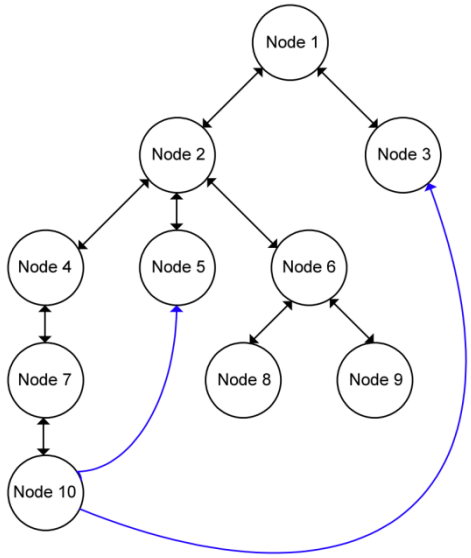

Figure 4: Node-Based Data Model Example

A collection of Node-based objects can support multiple different complex signaling hierarchies (listening and broadcasting relationships) at the same

time. This is due to the thread safe and recursive data modeling mechanism of the Node base class, which when inherited, facilitates the dynamic ability to establish and maintain multiple signal listening and broadcasting links. These capabilities represent the foundation for modeling problems of extraordinary complexity. Each Node-based object can include multiple links to other Node-based objects distributed across a network infrastructure. The issue of processor type, memory type, network type, and the complexity of the network configuration can be so cleanly abstracted by middleware drivers that the object hierarchy will not know or care about the hardware foundation that hosts it. In other words, a complex data model can be defined, implemented, and tested independently of the infrastructure on which it will ultimately run.

One set of object links might model a user interface on a ground based mission control computer so that when an operator selects a button a signal is propagated up a list of user interface objects (views and windows) as necessary to update the user interface and provide instant feedback. Another signal path maintained at the same time may include the object(s) that implement the processes that control any sub-system. Implemented in a robust, reliable, scalable, and well-tested fashion, the middleware includes all of the mechanisms necessary for a programmer to rapidly create data models of significant complexity.

PNP MIDDLEWARE EVALUATION

The system software principles and middleware characteristics summarized in this document will be given a detailed evaluation through a planned software demonstration for the AFRL Responsive Space Group at Kirtland Air Force Base. The first generation Plug-and-play Satellite (PnPSat), which is considered the SPA architecture reference design, has been built and tested by various contractors for this AFRL group. For the evaluation, the current software running on PnPSat will be temporarily replaced by ATK's middleware-based software.

During the PnPSat interface analysis phase, which included reviewing nearly two dozen hardware sub-components (coarse sun sensors, star trackers, GPS units, torque rods, reaction wheels, power hubs, etc.) and dozens of flight software sub-components, a clear design pattern surfaced for how to automate the PnP process using middleware-based mechanisms. All of the hardware sub-components that comprise the PnPSat (mostly sensors and actuators) use an identically formatted hardware register interface to store their respective data elements. The register and its SpaceWire bus connection are implemented through an Appliqué Sensor Interface Module (ASIM).[5] The ASIM for each different sensor and actuator also provides the respective device manufacturer's proprietary electrical interface. Think of the ASIM's register as a hardware abstraction layer between proprietary vendor electronics and a satellite's SpaceWire bus. From a hardware perspective, a register is a very well defined interface that enables an equally efficient software interface. For all sensors, their respective output data values are written to pre-defined register offsets (addresses) within their ASIM. For all actuators, their respective input data values are read from pre-defined register offsets within their ASIM. When combined with the current common data dictionary produced for the PnPSat effort, it will be demonstrated that defining interfaces for sensor and actuators using Nodes and attributes is very straight forward. It will also be demonstrated that building interfaces for dozens of flight software related sub-components is equally straight forward.

For the demonstration, the actual PnP process has been implemented using middleware provided building blocks. For example, the current middleware includes classes that implement a "Node Discovery Protocol." Using a simple signaling process, running from a designated processor within the PnPSat it is possible to rapidly discover any system sub-component (hardware or software endpoint). This process then quickly discovers all of the interfaces each endpoint supports. This is possible regardless of the order in which the endpoints power up or where they are located on the PnPSat's SpaceWire bus. The middleware-based discovery process also includes a very mature presence mechanism that maintains real-time situational awareness over the status of all endpoints. If an endpoint goes down the appropriate response can be immediately executed.

Every aspect of the middleware-based PnP implementation will be thoroughly simulated taking advantage of the proven journal based feedback loop. During the demonstration the middleware-based PnP solution will reflect a performance and functionality level well beyond what would be expected for a system software development effort of less than four months in length.

MIDDLEWARE BENEFITS

Properly implemented and tested middleware is an ideal approach to establishing a standard. It can facilitate the system wide integration of sub-components, make dissimilar technologies interoperable, and normalize data and control flow without the respective software developers having to fully understand the complexity of the underlying building blocks. Based on direct

observation of dozens of solutions built using a current generation of middleware—this approach also radically reduces both the number of lines of source code (by as much as 90 percent in many cases) and the time it takes to design, implement, and test a deliverable.

Middleware also provides insurance; both the underlying hardware and middleware can change continuously over time to support improvements and to add functionality, facilitating the delivery of future proofed systems. None of the sub-component interfaces within a system need to change, but if they do, all software can be fully backwards compatible.

A middleware approach to establishing standardization lowers the barrier of entry and complexity for vendors. All vendors have unlimited degrees of freedom to innovate and produce new solutions they know can be seamlessly integrated into future satellites. Vendors can focus on their technological strengths, not how their deliverables interface with innumerable other sub-components currently in use or built in the future. Any number of vendors can create, own, and sell proprietary software solutions that use the middleware. These vendors retain the intellectual property of what they create and can profitably sell solutions at whatever price the market will bear. As proven by the Internet, establishing both economic incentives and standards grows a competitive technology base that accelerates innovation and reliability while rapidly lowering sub-component costs. From the perspective of ORS mission accomplishment, this represents a clear and measurable "combat multiplier" and will allow ORS to meet its cost to performance goals.

The potential of the right middleware to (1) establish industry wide standardization, (2) accelerate satellite integration, (3) increase reliability and survivability of satellites, and (4) measurably support ORS goals in a cost effective fashion is significant and soon even quantifiable through simulations in a reference design.

CONCLUSION

In the face of current world events the ORS mission of rapidly deploying small satellites has the potential of playing a critical role in supporting national security and assisting in delivering humanitarian aid during a global crisis. Reliable satellite-based communications and information flow are essential to supporting these endeavors. The key to mission success is in the software. The complexities of the ORS requirements are so significant that the need for establishing standards and using a common software foundation are now self evident. A solution in the form of licensable and supported middleware currently demonstrates the potential of meeting or exceeding ORS needs.

References

1. American Institute of Aeronautics and Astronautics (AIAA) Standard, "Space Plug and Play Avionics (SPA) xTEDS," Draft Version 1.1.1_20080715 DLL, July 2008.
2. American Institute of Aeronautics and Astronautics (AIAA) Standard, "Space Plug and Play Avionics (SPA) Satellite Data Model (SDM)," Draft Version 1.1.1_20080715 DLL, July 2008.
3. American Institute of Aeronautics and Astronautics (AIAA) Standard, "Standards Development Guidebook for Space Plug and Play Avionics (SPA)," Draft Version 1.1.1_20080715 DLL, July 2008.
4. American Institute of Aeronautics and Astronautics (AIAA) Standard, "Space Plug and Play Avionics (SPA) – SpaceWire Adaptation," Draft Version 1.1.1_20080715 DLL, July 2008.
5. Air Force Research Laboratory (AFRL), "Appliqué Sensor Interface Module (ASIM) Application Development Kit for Generation One Experimental Modules Updated for ASIM Hardware Revision X3," Version 07.11.28, November 2007.
6. Air Force Research Laboratory (AFRL), "PnPSat Critical Design Review," August 29&30, 2007.
7. Miro Samek and Robert Ward, "Build a Super Simple Tasker," Embedded Systems Design, July 2006.