

Plug-and-Sense: An Enabling Onboard Networking Technology for Modular Spacecraft

Luis G. Jordan, Scott A. McDermott

AeroAstro, Inc.

20145 Ashbrook Place, Ashburn, VA 20145; (703) 723-9800

luis.jordan@aeroastro.com, scott.mcdermott@aeroastro.com

Distribution Statement "A" (Approved for Public Release, Distribution Unlimited)

ABSTRACT: Software, more than any other engineering discipline, allows extensive systems to be created with relatively little effort. This leveraging of engineering time is often aimed at making a software system as useful, flexible, and feature-laden as possible; but the complexity that inherently attends such efforts renders them unreliable, difficult to maintain, and nearly impossible to update. As the space industry moves toward modularity and “Plug-and-Play” concepts, significant discipline is required to control this new complexity. Layering is the tool that accomplishes this end. These issues are dealt with in the AstroLogic™ software suite for modular spacecraft. AstroLogic is a “Plug-and-Sense” architecture, meaning that it encompasses the exchange of functional information (as Plug-and-Play does for personal computers) as well as physical information relevant to a physical vehicle such as a spacecraft. It was developed to deal autonomously with the fact that the modules in a modular spacecraft do not know *a priori* what other modules will be part of the system or where they will be relative to each other. All information exchanged must be done autonomously with minimal or no custom software, or else the advantages of modularity are lost.

INTRODUCTION

The long lead and cycle times currently associated with development and launch of satellite systems are prohibitive for responsive deployment of technology and tactical capability to orbit. To mitigate this, a novel spacecraft architecture based on the coordination of multiple pre-designed functional modules is needed. Interoperability is key to making responsive modular spacecraft a reality. An interoperability standard that is too strict limits the architecture to compatibility with those technologies foreseen at the time that the standard was created. Conversely, a standard that is too loose offers no actual advantage, since the level of effort above and beyond that required by the standards that is necessary to get a system to work essentially requires a customized design, resulting in an unpredictable and potentially unreliable end product. For example, a standard that is limited to star trackers, sun sensors, and horizon-crossing indicators will have difficulty when a new (say, GPS-based) attitude determination sensor is developed. Conversely, a standard that simply allows for a generic attitude determination device does not accommodate the fact that a star tracker will not work when pointed near the Sun, Earth, or Moon. The AstroLogic software suite – a key, enabling component of AeroAstro’s SMARTBus modular spacecraft architecture – attempts to find a compromise between these two extremes in that it defines the standards to a level that best facilitates their implementation. In the example above, AstroLogic would define an attitude determination device with an envelope of orbital conditions within which it can operate successfully.

AstroLogic is based on the well-known and understood Plug-and-Play concepts of the personal computer industry. In these Plug-and-Play systems, a piece of equipment identifies itself and its capabilities on a common bus, and all of the information required to operate that piece of equipment is contained within the equipment itself. However, in an environment in which the physical attributes are as important as the capabilities of the device, identification of capabilities alone is insufficient. Therefore, AstroLogic expands the implementation of Plug-and-Play to support spacecraft operations, including identification of the device’s physical characteristics, such as mass, size, shape, center of gravity, moments of inertia, position within the vehicle, and so forth. Thus, AstroLogic is referred to as a Plug-and-Sense system. The specific details of AstroLogic are complex, but the idea is simple and straightforward: it allows a set of spacecraft devices to autonomously and unambiguously understand how they are configured and what services they can perform as a whole, and to communicate essential parameters to other modules within the system.

The Universal Plug and Play (UPnP™) architecture was considered for AstroLogic, but it was ultimately rejected because it is too verbose for efficient use on a real-time, resource-limited system like a satellite. The broadcast client/server model of computer networks was deemed more appropriate, so it was used as the basis for resource allocation. Just as a client announces a request for a given resource on a typical LAN or WAN network – memory, processing time, a given application executable – and any server capable of

responding does so, in AstroLogic, a device announces its request – for power, downlink, or attitude knowledge, for example – and any device capable of providing this resource responds to the request.

The concept that devices and resources transcend the device’s primary function or its physical location in the vehicle is also important. That is, while a given module typically provides a certain set of resources, the set of resources need not be limited to the module’s primary function. A solar array module may also provide attitude determination information, and an attitude determination module can offer spare memory. Any device that can provide a resource should do so; and it does not matter, in AstroLogic, where that resource might physically lie within the overall system.

The key to successfully deploying a complex architecture such as AstroLogic is ultimately in the usefulness, friendliness, and robustness of the final product. In creating AstroLogic, the AeroAstro team focused on five main design drivers that governed the trades and decisions of the team. These drivers were: implementation independence; compatibility with Space Plug-and-Play Avionics (SPA); breaking the physical device barriers; leveraging existing concepts and tools; and the efficiency of the engine in which the logic runs. Each is described in turn later in this paper.

AEROASTRO’S SMARTBUS™

AeroAstro originally developed AstroLogic to support its SMARTBus modular spacecraft architecture. Although AstroLogic exists independently of a specific

hardware or physical implementation, SMARTBus is an exemplar of AstroLogic’s utility. The SMARTBus modular spacecraft architecture comprises mechanical, electrical, and logical (interoperability) standards. These enable complete spacecraft to be assembled from a series of stackable modules (see Figure 1). It was designed to strike a balance between standardization and flexibility, mandating enough of the form and function to enable smooth integration, while leaving enough flexibility to allow for expansion, creativity, and the unanticipated. The specifications for a SMARTBus module are strict and specific without being too restrictive, so that modules created earlier will interface smoothly and properly with modules created in the future, and may be compatible with nearly any launch vehicle. Once these requirements have been met, a module designer may implement that module’s functions in any way desired.

With a stackable modular architecture, individual subsystem capabilities may be selected independently of other subsystems without the significant overhead associated with integrating a new spacecraft design each time a new set of subsystem requirements is encountered. For example, the designer can choose a given bit rate at a given frequency band; so much attitude control authority with a particular control method; this many Watt-hours of energy storage; etc., without redesigning the spacecraft from scratch each time a new spacecraft with a different set of capabilities is required. If there is only a minimal difference between two spacecraft – for example, one spacecraft requires a horizon-crossing indicator while the other requires a star tracker – with SMARTBus, only the

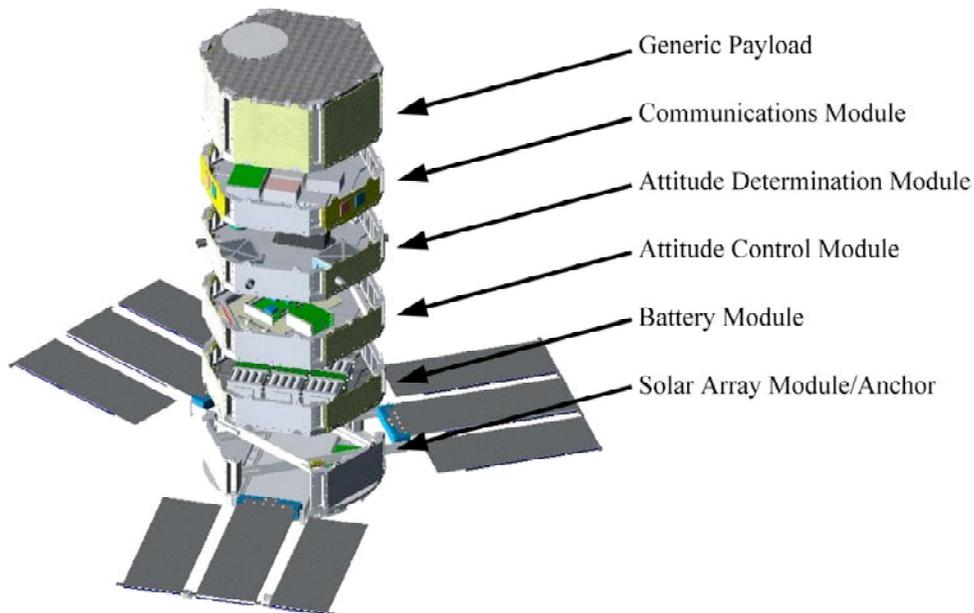


Figure 1. A Typical SMARTBus Modular Vehicle Stack.

corresponding module (in this example, the attitude determination module) must be swapped. While the impact of such a change might require significant packaging redesign in a traditional satellite, it has a negligible impact with the SMARTBus architecture.

SMARTBus offers several key advantages: rapid system integration, low recurring cost, and improved reliability. Subsystem redesign and new system design, which has become inherent in satellite manufacturing, largely disappears by virtue of the fact that modules are individually designed in accordance with the defined SMARTBus standards and can then be used interchangeably. Activities that previously required significant effort – such as structure design, fabrication, and test; harnessing creation, routing, and assembly; subsystem interface development and control; and myriad other “from the ground up” satellite development efforts – can now be avoided. This promotes focus on the mission as a whole and the existing modules that may be combined to achieve that mission. Since proven designs will frequently be reused, the unanticipated problems inherent in new designs will be avoided, decreasing integration time, and allowing incorporation of lessons learned.

The goal of SMARTBus is to enable the use of “off-the-shelf” spacecraft modules at a product-level (rather than development-level) price point, stack them, test them, and be ready to integrate a payload in a matter of hours, rather than months or years. An ambitious goal such as this does not come easily, and these advantages come with commensurate cost, in terms of volume, mass, performance, and other critical factors. While SMARTBus is not the answer for every mission, if integration time or spacecraft price are driving factors, these tradeoffs will likely make it a preferred approach.

There are a number of terms used in describing the SMARTBus architecture that warrant definition and explanation. These terms are defined below.

- A **module** or **slice** is the basic building block of a SMARTBus spacecraft. It performs a specific and

well-defined set of functions. It is hexagonal in shape and may have any of a discrete set of heights.

- A SMARTBus **device** is any component that follows applicable electrical and logical interface standards, but is not necessarily a module. A module is a specific kind of device.
- Devices may provide or request **resources**. Spacecraft resources include power, pointing, communications, memory, and so forth. Devices interact by requesting resources from each other.
- **AstroLogic** is the particular set of logical interface requirements that facilitate device interaction. It is a Plug-and-Play type of architecture, which includes physical attributes like center of gravity, moment of inertia, height, shape, etc. This approach is known as **Plug-and-Sense**. AstroLogic is a specific Plug-and-Sense system for spacecraft.
- The combination of a set of modules is referred to as a **stack**. Creating a stack is the core of the integration process for a SMARTBus vehicle (Figure 2 shows a prototype stack). The choice of stack elements and stack order is the spacecraft designer’s primary task, and certain stacks will perform better than others for a given mission.
- The **Anchor** is the bottom module. It is the module that attaches to the launch vehicle interface. The Anchor has certain specific responsibilities and requirements. On most SMARTBus missions, the Anchor will be a solar array module.
- Modules interact electrically over the **Backbone**, a power, data, and signal bus that runs up one vertex of the modules’ hexagonal form. Each module has one connector on its bottom and a complementary connector on its top to permit attachment of modules and connection to the Backbone.
- One of the devices in the spacecraft functions as the **Arbiter**. This is a computer that governs spacecraft-wide resources, such as power and pointing, and manages resource conflicts (the function of the Arbiter will be detailed further later in the paper).

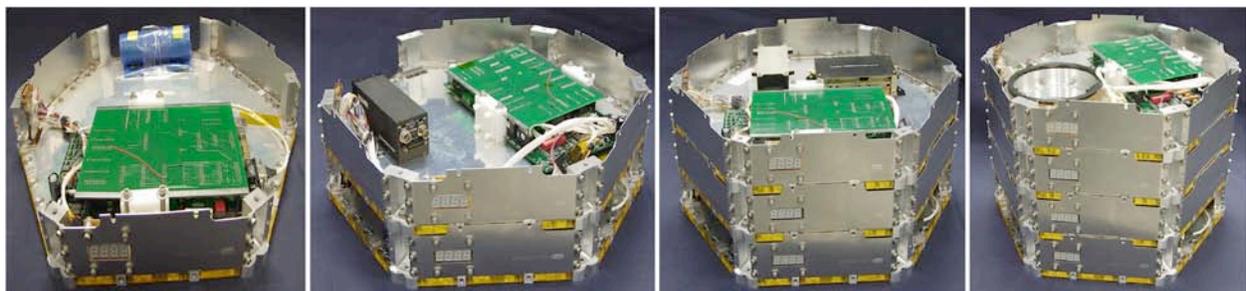


Figure 2. Stacking of Prototype SMARTBus Modules.

With these terms in mind, the sequence of a SMARTBus mission generally runs as follows:

1. Establish initial mission parameters. These include power, communications, attitude knowledge, attitude control, memory, processing power, and so forth. They are referred to as “initial parameters” rather than “requirements” since they will likely change according to the capability of the available modules; this is one of the tradeoffs of the modular architecture.
2. Determine the set of SMARTBus modules that come closest to achieving these initial parameters. Existing modules are preferable, even if they if they have somewhat lesser (or greater) capabilities.
3. Establish and verify the stack order of the chosen modules. Optimize power, communications, thermal, and sensor view considerations (preferably using automated tools).
4. Adjust mission parameters to accommodate the modules selected.
5. Procure the selected modules, as well as the interface for the selected launch vehicle.
6. Assemble the selected modules into a stack in its defined stack order.
7. Verify stack operation using automated tools.
8. Attach the payload, if any.
9. Load mission-specific software, including any information required to interface with the payload or the ground station, onto at least one computer onboard the spacecraft, typically the one selected as the Arbiter.

10. Functionally test payload interface and mission operations. The more compliant a payload is with SMARTBus standards, the more straightforward and easily automated this will be.

11. Environmentally test the assembled spacecraft as required by the launch vehicle or mission manager.

12. Deliver the spacecraft for integration with the launch vehicle and then launch it.

One of the primary goals of the SMARTBus architecture is to perform spacecraft integration through environmental testing in a matter of hours, assuming all the equipment (including the payload) adheres to the SMARTBus standards and that the required modules are on hand.

THE IMPORTANCE OF LAYERING

The means to address such an extensive architecture as SMARTBus, as is the case with responsive space in general, is centered on breaking up the problem into manageable elements. Throughout the development and refinement of this modular architecture, the concept of layering has persisted as the best tool to control complexity, which can ultimately be cumbersome and dangerous to the utility of the architecture.

Layering requires breaking down a complicated system into its sensible, more manageable, constituent parts, and then dealing with each layer individually. The particular set of layers selected is not canonical, however, certain themes have consistently arisen in the development of SMARTBus and AstroLogic. The primary layers that have been incorporated into the architecture are shown in Figure 3, and Figure 4 shows how they interrelate.

APPLICATION	<i>Subsystem function: attitude control, communications, etc.</i>
TRANSPORT	<i>How to identify and communicate particular applications</i>
NETWORK	<i>How to locate and communicate with devices that host applications</i>
LINK / MAC	<i>Means for sharing a common resource like a data bus or power rail</i>
PHYSICAL	<i>The electrical data connection itself: data, power, and signals</i>
CONNECTOR	<i>Wires and their housings conducting electrical signals among devices</i>
STRUCTURAL	<i>Connecting devices physically to tolerate launch and other loads</i>
THERMAL	<i>Ensuring that connected devices remain thermally stable</i>

Figure 3. Layering in Design.

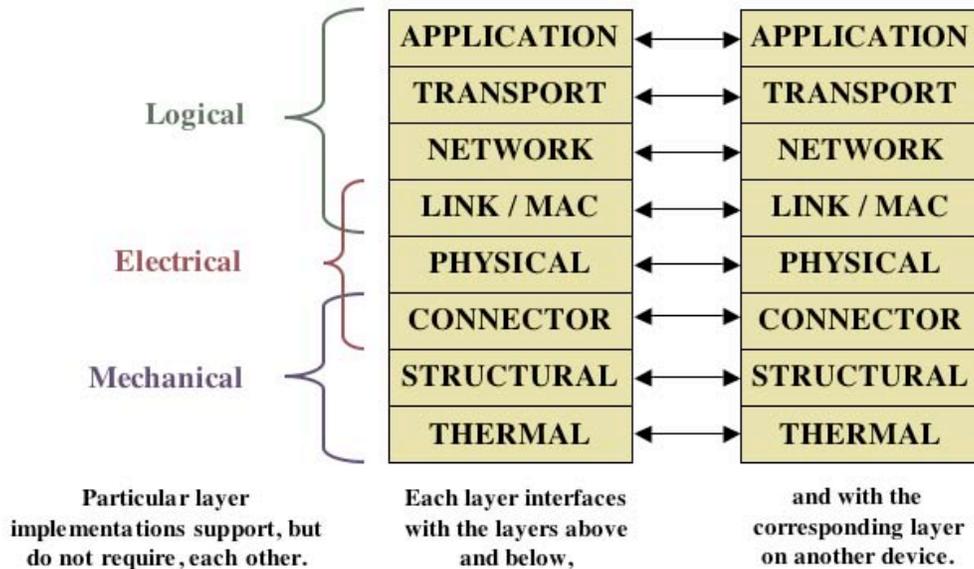


Figure 4. Interaction Among Layers.

The primary rule of layering in AstroLogic is that no layer may be dependent on, or even be aware of, the implementation of any other nonadjacent layer. If this rule is abided by, each element is kept relatively simple, and different implementations can be dropped into a given layer – even simultaneously, as one might do with the Link layer, which is likely to have many implementations – with minimal impact on the whole, large, complex system. If this rule is ignored – for example, if the Application layer starts taking into account how the Network layer might route data, or if the Connector layer cares about how Medium Access Control is performed over the wires within the connector – the entire design begins to falter and become mired in its own interrelationships.

It is important to note that a given implementation may well be aware of the *existence* of all the other layers, but it must never be dependent on the *implementation* of any nonadjacent layer. For example, a Transport layer can know that a Physical layer exists and therefore, physical data transfer delays are introduced and the response may not be immediate; but which Physical layer(s) are involved in communication should not be visible to the Transport layer. The single most critical lesson learned in the SMARTBus development has been to keep the layers distinct and their interfaces well defined.

Another lesson learned in the implementation of layers in the SMARTBus architecture is that for any given layer, there will tend to be an already-established, industrially supported implementation that is readily available for incorporation into the design. These

existing approaches should be used whenever possible, even if they seem to be suboptimal at first glance. This topic is explored further in a later section.

Finally, the use of disciplined layering enables new technologies and new approaches to be incorporated smoothly and efficiently. Different structural approaches may be used for a varying sizes or shapes of spacecraft without the loss of electrical and logical interoperabilities designed into the system. New data buses and new applications can be integrated with comparable ease since the interdependencies are virtually nonexistent beyond the immediately adjacent layers. The development of standards in the aerospace industry can thus be made mutually advantageous at a variety of levels: different ideas from different approaches can be merged into a smooth and workable system applicable to a particular class of missions.

THE PATH TO ASTROLOGIC

Implementation Independence

Traditional custom spacecraft are designed around the needs of a payload or mission objectives. Relatively simple tasks can take orders of magnitude longer than they ought to in a mature industry. Many work hours, are often consumed in calculating power budgets to size solar arrays and batteries, creating thermal models based on vehicle’s configuration and orbit, and updating all of these data as development proceeds. Other, more delicate calculations – such the vehicle’s moments of inertia and center of gravity – are not accurately known until the vehicle is mostly designed.

One of the unique challenges that face SMARTBus module developers is that the final configuration of the vehicle, or what other modules will be interacting with theirs, is not known at the time of module design. They will not even know what other modules will be physically adjacent to them. Oftentimes, this final configuration will only be known weeks, days, or hours before it needs to be on orbit and operational. The “luxury” of mission conceptual design as we know it is non-existent. The launch readiness review must be trivialized to a handful of go/no-go decisions based on results from a simple tool. Figure 5 depicts a SMARTBus attached to its Multi Use Ground Support Equipment (MUGSE), which is an automated tool for performing this binary determination. In other words, module developers have minimal information and must design the module independently of how it will be used and what its interaction with other modules will be – instead they must rely on the standards and design guidelines provided.

As with modularity in any system, the rationale for modularity in spacecraft is simply to be able to use two individual devices together in order to perform a task, even if those devices were not originally designed specifically to work with each other. Rather, the devices are designed to be compatible with a standard interface, and the implementation on either side of the interface is invisible to the other side. In the mechanical layers of the spacecraft, these interfaces are expressed in terms of such things as bolt-hole patterns, thermal conduction and radiation paths. In the electrical layers, these interfaces are expressed in terms of such things as voltage, current, and rise time. In the logical layers, they are expressed in terms of software.

After devices are connected physically, they must be able to communicate with each other. In a typical spacecraft, this is accomplished in a very specific,

customized way. For example, the command & data handling (C&DH) system has a software driver for interacting with the momentum wheel, another for interacting with the star camera, and an attitude control algorithm for allowing the two to work together. If a new momentum wheel is selected, or if a differential-GPS system is used in place of a star camera, new software must be written for this specific configuration. While this is not necessarily a bad approach, since it keeps the software focused and streamlined and allows for very little emergent (unanticipated) behavior, it is extremely labor-intensive and does not support rapid or low-cost mission development.

Through its Plug-and-Sense capability, the SMARTBus standard has taken a very different approach. Instead of designing the software around the equipment to be used in a system, AstroLogic was designed around the resources offered by a system. Rather than commanding a particular brand and model of momentum wheel, the attitude control algorithm requests that any device capable of controlling pointing do so. Instead of implementing different software for a differential-GPS (dGPS) system than is implemented for a star camera system, the attitude determination algorithm simply asks for information on attitude from any device that can supply it. Plug-and-Sense also goes a step beyond the more familiar commercial technology of Plug-and-Play in that it is able to identify not just a module’s functional characteristics but its physical characteristics as well. This is critical, since a spacecraft is, at its core, a vehicle whose size, shape and mass matters. Thus, the integration process is shortened, since it becomes enormously less involved and less labor-intensive.

Compatibility with Space Plug-and-Play Avionics

AeroAstro has actively participated in the Air Force Research Laboratory’s (AFRL) Space Plug-and-Play



Figure 5. An Assembled, Functional SMARTBus Stack with Ground Support Equipment.

Avionics (SPA) initiative since its inception. As its name indicates, SPA is focused on avionics and computer-component interaction. The nature of SMARTBus has forced the development team to address a broader set of issues, mainly physical attributes of modules, location on the spacecraft, and thermal dissipation.

As mentioned earlier in this paper, one benefit of having a layered architecture is the ability to seamlessly incorporate new technologies as they become available. As such, each xTEDS, an XML-based device definition developed by SPA, may be incorporated as a Semantic software layer implementation for the given device type. This is complementary to the inter-module interfacing that AstroLogic must also take into account. For example:

- Sensors – magnetometer, star tracker, sun sensors – attach and self-configure to the module’s core electronics according to SPA standards

- Core Electronics Blocks (CEBs) interact, using the same SPA standard data dictionary (xTEDs), to share information among modules in an Internet Protocol (IP) environment
- Modules provide quick mechanical assembly; SPA standards provide Plug-and-Play functionality

Breaking Physical Device Barriers

As discussed earlier, AstroLogic was designed around resources (logical view) rather than equipment and the principal function that equipment provides (physical view), as shown in Figure 6. AstroLogic devices interact by requesting and providing resources.

In support of this approach, AstroLogic has been built on the peer-to-peer multicast client/server approach used in network computing. In such a network, the specific situation dictates whether computers act as a client or as a server. A given computer with a need – to

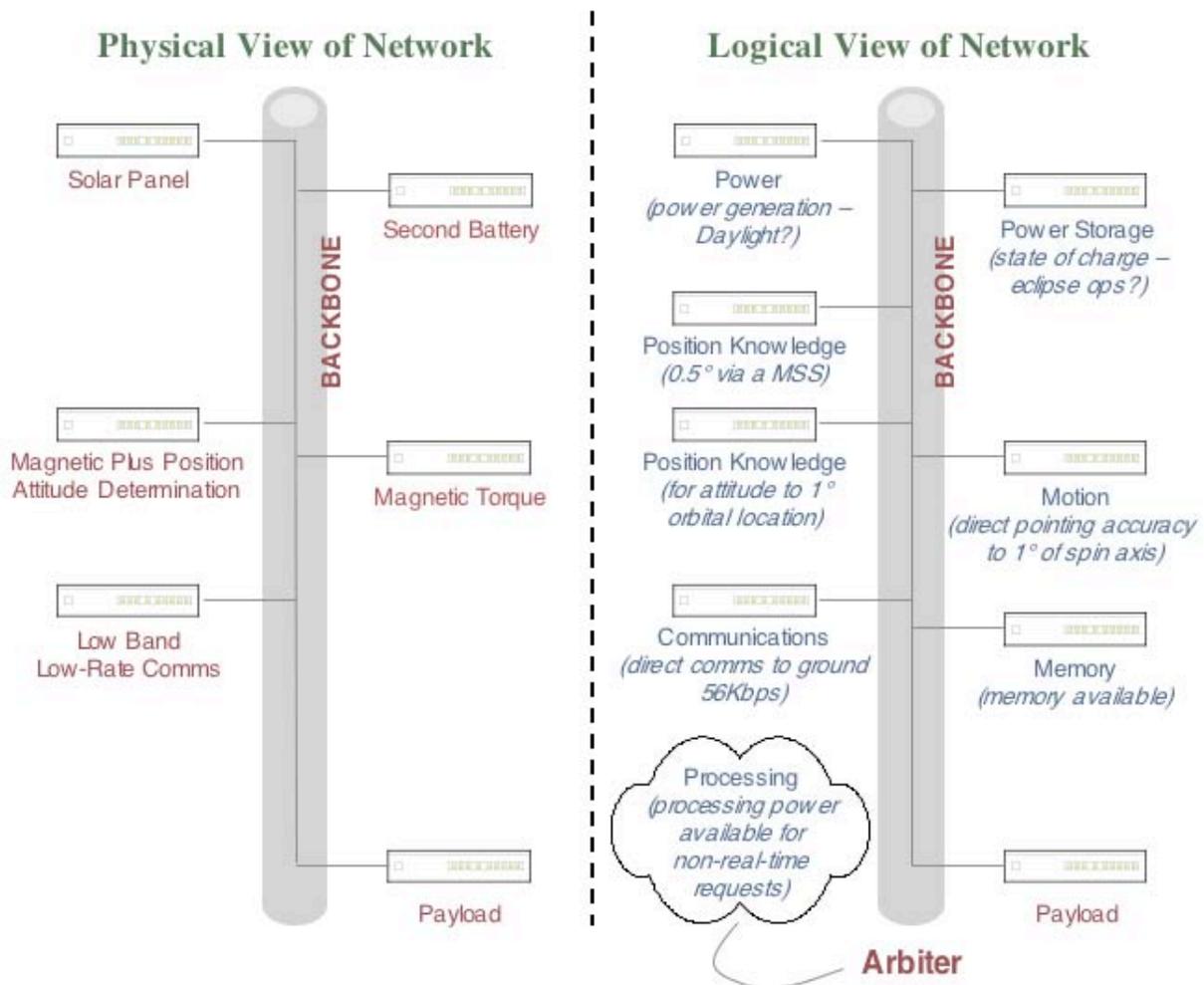


Figure 6. Physical versus Logical Views.

print a document, for example – sends a request to other computers on the network for access to a printer. Any servers available for printing respond, giving the user a choice among them, and the document is printed. Peer-to-peer multicast client/server is a mature technology with characteristics that serve well as the basis for AstroLogic.

Similar to the client/server distinction, AstroLogic’s approach is to divide a spacecraft into *resource providers* and *resource consumers*. Any given device may be a provider of some resources and a consumer of others. A device listens for resource requests on a published port, and identifies those types of resources it is capable of supplying, as shown in Table 1.

Table 1. UDP Listen Ports for SMARTBus.

Request Listen Port	Resource Type	Description
8000	Device	All devices listen to Device requests
8001	Communications	Direct communications (space-ground or space-space)
8002	Motion	Direct pointing and propulsion
8003	Position Knowledge	Attitude and orbit location
8004	Power	Direct power supply
8005	Arbitration	Organizing power, motion, and communications needs overall
8006	Memory	Mass data storage
8007	Processing	Non-real-time data processing
8008	Anchor	Bottommost module in SMARTBus stack
8009	Other	Resources distinct from those listed
8010	Imagery	Cameras on-board the spacecraft
8011	Payload	Special category for mission-unique components
8012	Time	Spacecraft timekeeper and source of sync pulses
8013	Reserved	
8014	Ground Operations	Anything specific to ground test or operations equipment
8015	<i>Reserved</i>	

With few exceptions, the same basic themes recur consistently across all message exchanges. A typical message exchange consists of a request, response, acceptance, possible data flow, and a withdrawal. An example of a message exchange for the Physical Parameters resource is shown in Figure 7. The Physical Parameters exchange allows the responder to provide the requester with the size, shape, physical protrusions, and manufacturer-specific data that this device contains.

For the most part, the different resource types that have been defined are fairly self-explanatory. Attitude determination and orbital position are conflated into Position Knowledge. Attitude control and propulsion are fused into Motion. The Device resource type is applicable to all resource exchanges for which the physical characteristics of a device – its size, shape, position, mass, mass properties, and electrical characteristics – are relevant. The Anchor resource type

primarily allows for message exchanges that enable a device to determine its physical position within the stack relative to other devices; the Anchor, the bottom module in the stack, defines the “zero datum” position in the vehicle from which all other devices measure their position. The Imagery resource type recognizes that some sort of imagery – whether it is imagery from a camera, star tracker, or another multidimensional and directional sensor array – is available and exchanges imagery-related information. Imagery is prevalent enough on modern spacecraft to warrant specific exchanges related to sharing that kind of information. The Payload resource type has only a few pre-defined exchanges and is largely open to mission-specific definition. The Time resource type allows a single timebase and synchronization system to govern all time recording on a spacecraft; the timebase and synchronization might be provided initially by a module with a precise clock, and then later be taken over by another device receiving GPS signals.

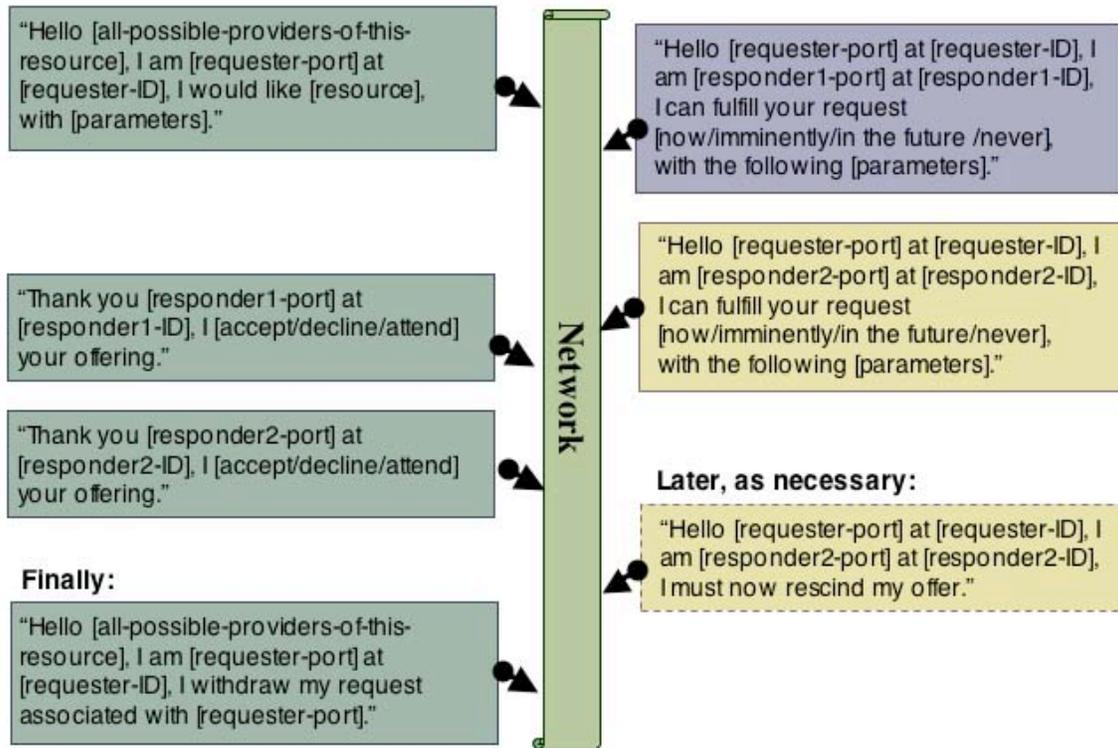


Figure 7. The Request/Response Exchange.

The presence of Power, Motion, and Communications resource types create a new issue that must be addressed: how should each of these resource types respond to conflicting requests, or more accurately, how should different devices that provide the same resource types avoid conflicting with each other when responding to a request? In this system, the answer is the creation of the Arbiter resource type. The Arbiter resource type is unique in that it must *be* unique; that is, there must be only one Arbiter on the spacecraft. The Arbiter manages power, communications, and motion resources for the entire vehicle. A device that requires one of these three resources requests it of the Arbiter, and the Arbiter in turn issues a request to particular providers of that resource and decides how best to fulfill the request. The use of an Arbiter is necessary because: there may too many providers of a resource (such as the pointing of the vehicle, which can be provided either by a momentum wheel or a propulsion device) and they must not be allowed to operate to contradictory ends; or, there is no single provider of a resource that can sensibly provide it alone (such as is the case with power, which is typically provided by a combination of solar panels and batteries, neither one of which can independently offer power to a device).

The software system governing a given resource type is considered to be at the Application layer (see Figure 8,

described in detail later). A given host (physical machine) is capable of responding to more than one resource type by having more than one application running, taking advantage of all the same layers below the Application layer.

Leveraging Existing Concepts and Tools

As mentioned earlier, for many of the defined layers, there is an already-established and industrially supported implementation (either within aerospace or in other industries), and this established implementation, or implementations if multiple exist, should be used in the appropriate layer if at all feasible. While this pre-existing implementation – whether it is a connector, or a link protocol, or a means for identifying applications on devices – will almost never appear to be the optimal solution for the problem at hand, utilizing a pre-existing standard has proven to have numerous secondary benefits. These benefits range from the existence of many professionals offering support and assistance via mailing lists, to the absence of the unforeseeable and subtle problems inherent in any new design.

In the process of developing the SMARTBus standards and the subsequent implementation of these standards on SMARTBus modules, AeroAstro found that many of the problems encountered with pre-existing layer

implementations were related to an unrecognized attempt to customize or modify the standard implementation in some way, in an effort – later found to be misguided – to make the standard “better” in some capacity.

AstroLogic makes use of standard implementations for as many layers as feasible. This is most evident in the Syntax, Transport, and Network layers (Figure 8). Basing the implementation of these layers on established network protocols such as TCP, UDP, x3d, etc. has brought about a number of unexpected benefits, including:

- No additional development was needed to support a distributed spacecraft rather than a physically connected one; the networking protocols worked correctly in either case.
- The use of Internet-based protocols has made AstroLogic robust and reliable. Each of the intramodule data links is treated as a separate subnet, able to provide a bridge between themselves at every module. A broken bus connection is routed around as automatically on the spacecraft as it is on the ground.
- Other communications systems such as Cisco routing, Win-T battlespace comms, and Wi-Max fragmented spacecraft networks can be incorporated seamlessly.
- Ground station systems no longer need a special protocol to communicate with the spacecraft. The ground systems are just viewed as additional nodes that are accessible to the spacecraft. The spacecraft’s communications system simply provides a bridge between the on-board data bus and a “wireless” subnet.
- True remote testing of different modules can be achieved. One need only provide the module to the Internet and give it an address.

Our experience has been that existing protocols and the tools that support them quite simply “just work” and that using existing tools confer real and consistent benefits.

Through use of Internet Protocol (IP), the logical world of Plug-and-Sense is bridged to the physical world of actual data buses and communication systems. It provides a unique identifier – an IP address – which enables a device to be accessed independently of its physical location and independently of the reason the access is required. This abstraction makes the Internet Protocol quite powerful. By using standard IP tunneling

techniques, packets can be routed to devices that are not physically located on the spacecraft’s data bus – slave picosatellites, for example. Remote integration is thus enabled, with devices in entirely separate facilities during functional checkout. It also facilitates simple device simulation using personal computers on any private or open network. If a device has one or more IP addresses (corresponding to the number of data buses it uses), it is ready to begin operating as a SMARTBus unit.

AstroLogic uses the simple User Datagram Protocol (UDP) as its transport layer for several reasons:

- Very little overhead, both on the physical data bus and within the computer software, is required.
- A majority of the communications are inherently intra-spacecraft, so packets do not travel over anything other than local links, reducing or possibly even eliminating the benefit of packet-level acknowledgements and sequence sorting.
- By handling responses/acknowledgements and multi-packet message handling at the Application layer when needed, such functions at the Transport layer are made redundant.

UDP is defined in RFC 768 of the Internet Engineering Task Force (<http://www.ietf.org/>), which remains the defining body. UDP appends an 8-byte header to the data packet and passes the resulting message to the IP layer for transmission. The appended header contains information on source and destination ports to allow each communicating device to support many distinct connections. This is the essence of the Transport layer. This ability to identify particular processes within a device is used in commonly known Internet services such as telnet (port 23) and http (port 80). It is used in SMARTBus, in the Transport layer, to identify spacecraft resources that a given device can supply, as shown in Table 1.

Resources are allocated via a Request/Response Exchange (RRE), an approach based on well-established client/server systems. In an RRE system, a requester announces the need for specific resources across the network, intended for all providers of that resource type; of those resource providers, those that can fulfill the request respond. For each of those responders, the requester either accepts or declines their resource offer.

With SMARTBus, the content of the Request/Response Exchange messages themselves are structured as eXtensible Markup Language (XML) messages, which

contain both information itself and the meaning of that information in the same message. By “tagging” each piece of information with contextual information, new features are enabled, optional features can be utilized or ignored, and integration difficulties are minimized, because a change in information does not necessitate a complete change in interface definition.

AstroLogic Engine Efficiency

The most difficult task in developing AstroLogic is creating the Command/Request engine. The engine, at its core, has been optimized to perform one and only one job: to make it easy for applications to exchange data. Object-oriented design lent itself naturally to the needs of the AstroLogic engine. A properly constructed set of classes, sensible inheritance, and clear public methods make a system that is robust, maintainable, expandable, and very simple to implement.

Figure 8 is intended to demonstrate the concept of what software layers are being used when building a SMARTBus module, or anything similar. It focuses on the pieces that AstroLogic requires but also shows how other standards protocols – such as a login system or the Spacecraft Command Language (SCL) – would fit into these layers. The concept of a “message” is what

traverses these layers; one ought not to think of a message as a block of XML or as an IP packet, but as the more general concept of “data to be communicated between boxes.”

Each of the colored boxes shown in Figure 8 associated with a layer is referred to as an implementation of that layer. Implementation names in black indicate items that will be implemented in the short term for SMARTBus and AstroLogic. Grey names represent other examples of implementations of that layer that are not currently on a schedule to be implemented but could be used for various software interaction purposes on the vehicle. The following guidelines apply when reading the chart:

- **Down the Chart:** Any implementation of any layer may choose which implementation of the immediately lower layer it will use at any given time to send a message. It has no visibility beyond the immediately adjacent lower layer.
- **Up the Chart:** Any implementation of any layer decides based on message contents which implementation of the immediately higher layer the message should go to. It has no visibility beyond the immediately adjacent higher layer.

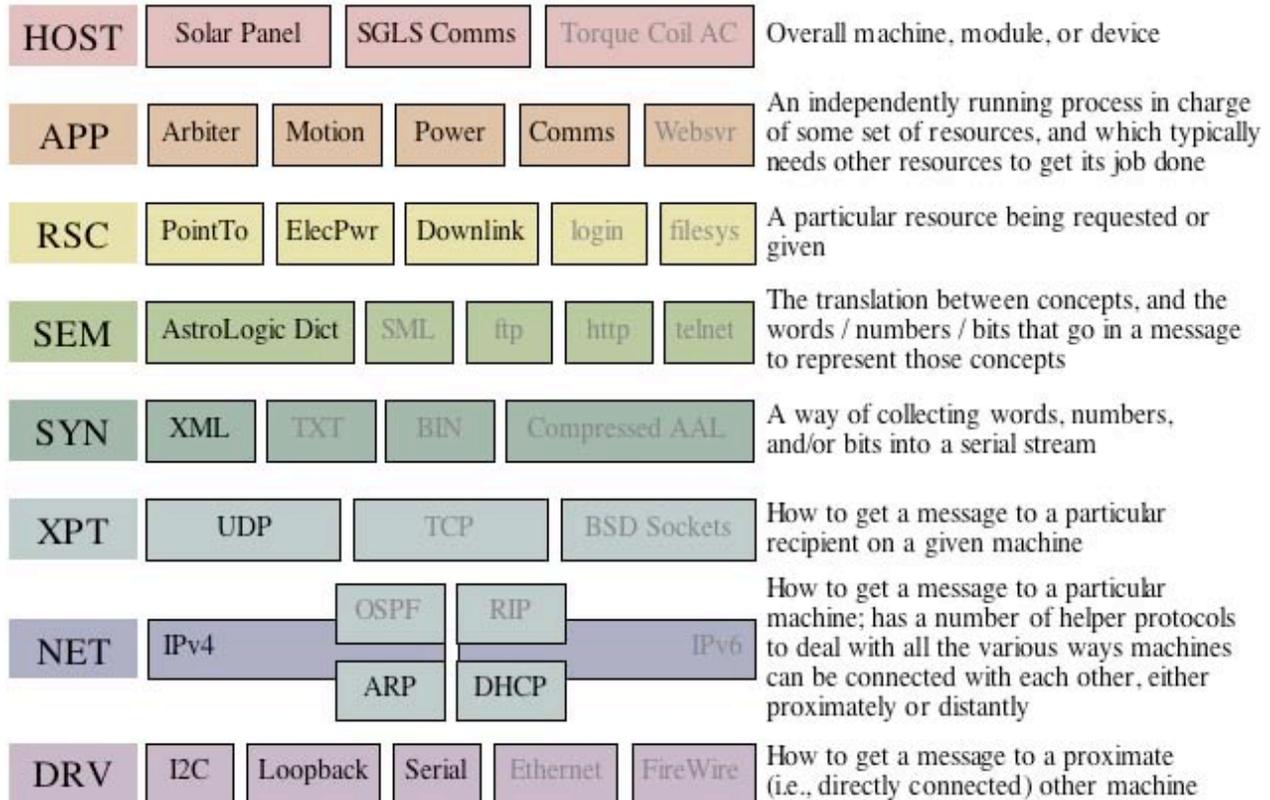


Figure 8. Software Layers for SMARTBus and Other Examples.

➤ **From Chart to Chart:** Any message at any layer is ultimately on its way to or from the same layer on another device (or even on the same device in the case of loopback). Syntax layers talk to Syntax layers, Application layers talk to Application layers. At the Semantic layer and below, the message is actually going from one implementation of the layer to the same implementation of the layer on the other device. At the Resource layer and below, different implementations can talk to each other.

The Resource layer knows that there are other entities on other devices, even if it does not know that those entities might be referred to by a UDP port and an IPv4 address. The Resource layer needs some way of saying “send this message here,” even if it doesn’t know how here is encoded. Since this abstraction applies at many layers, the whole process can be wrapped into a class called “Message”, which takes care of moving information up and down the stack.

A second layer-traversing tool recognizes that each software entity does not want to constantly ask itself “Do I have something to do?”, but rather, wants software or a daemon running independently to wake it up when something that is of interest happens. This gives rise to the `EventManager` class, of which there is typically one instantiation per computing device. Anyone can use the `EventManager`, but typically it is of most use to the higher layers.

Some classes and their relationships can now be defined, but first some basics need to be explained. The one and only `EventManager` on a device is instantiated as the object `EventManager`. AstroLogic messages are a particular subclass of `Message`, called `aalMsg`,

which take care of the message’s routing down through most of the layer hierarchy. Currently, that routing is from XML to UDP to IPv4; however, if any of those choices should change, only `aalMsg` will know about it.

With those basics in hand, Figure 9 depicts how AstroLogic message exchange works. It starts with an application somewhere with a need for a resource. Once the connection is made, subsequent interactions occur between the `ResourceRequester` and the `ResourceResponder`, with the respective `Application` and `ResourceGovernor` being notified when relevant (see Figure 10).

One interesting point is that the `ResourceGovernor`, which is ultimately implemented as specific to a resource (such as `PointTo` or `Downlink`), is shown as being an Application-layer entity. This is due to the fact that while specific exchanges between a Requester and a Responder exist in ignorance of how the resource is being supplied (i.e., are at the Resource layer), the Governor of the resource – even though it is a specific resource – functions only in the larger context of the Application layer.

For example, a governor of the `PointingKnowledge` resource must gather current pointing information from the attitude determination (or in AstroLogic parlance, the `Position Knowledge`) system as a whole; if an attitude sensor goes off-line, the `PointingKnowledge Governor` needs to know about it. The `PointingKnowledge ResourceResponder`, however, knows what `PointingKnowledge` is but does not know where it comes from. The information just appears, specifically, from the Governor.

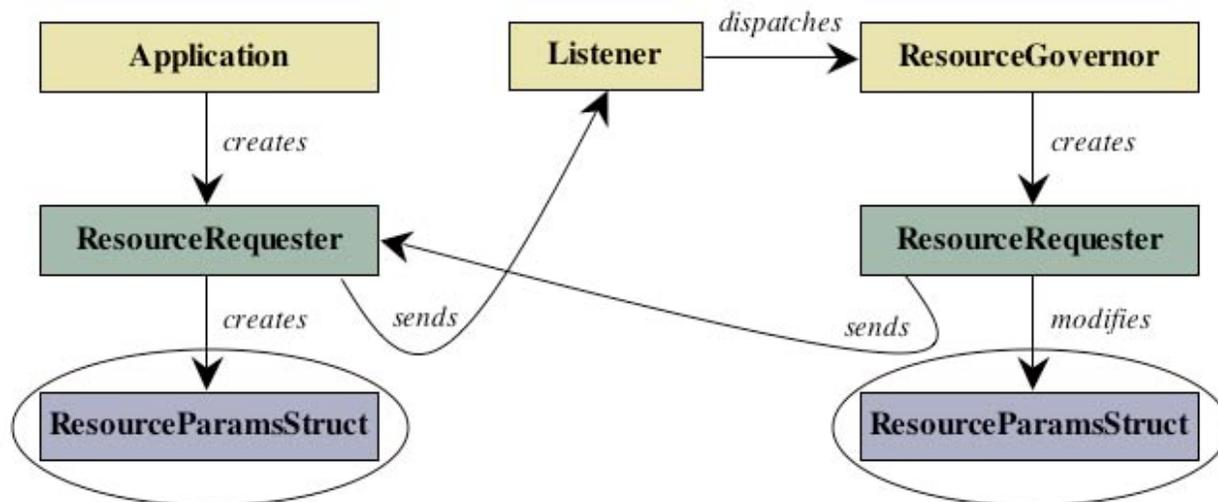


Figure 9. Depiction of an Application Requesting a Resource.

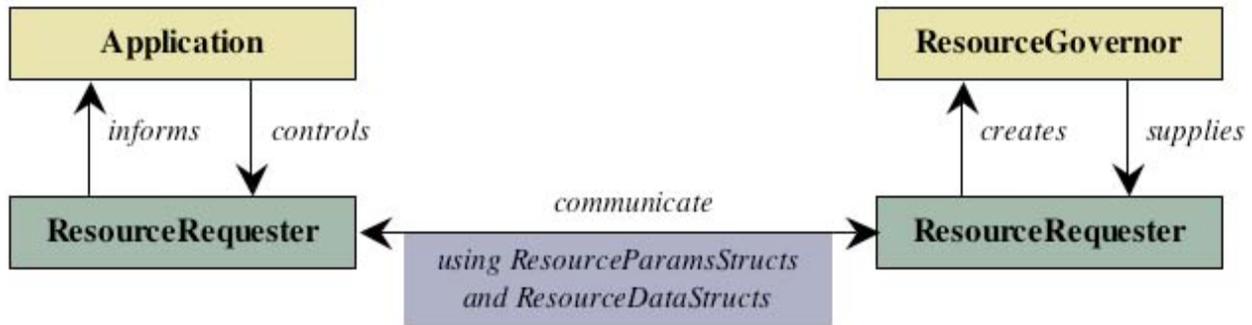


Figure 10. Requester/Responder Interaction.

THE ASTROLOGIC DICTIONARY AND MESSAGING

An AstroLogic message is made up of one or more packets. A given set of messages representing a request, response, data transfer, and/or request withdrawal are known as an exchange. Each exchange has a unique resource name within a given resource type.

The actual software message exchanges are defined in a growing document called the AstroLogic Dictionary. The dictionary is essentially a list of defined exchanges. Needless to say, new exchanges may be added and clarifications offered relatively easily as new device types come available. As with any other messaging system, a most serious commitment to maintain backward compatibility should be undertaken. This dictionary is, however, only one layer of the whole software system (the Semantic layer), and other data dictionaries can fit in as well. Conversely, the AstroLogic Dictionary can be used as a drop-in layer implementation in another system.

The dictionary is separated into the various resource types available in AstroLogic as described in Table 1. A given resource-provider must implement all exchanges marked "MANDATORY" for its resource type. All devices must implement all exchanges marked "MANDATORY" for the Device resource type. Implementing an exchange means that the device must provide properly formatted and valid responses to a request for that resource type, even if the response is a negative one.

There is a certain set of common elements that thread through the exchanges that ought to be used consistently in new exchanges for ease of implementation. In addition to the common elements detailed below, a new exchange should, in general, conform to the patterns set down by existing similar exchanges. The use of XML is intended to encourage and ease this commonality.

Message Structure

The overall themes of message exchanges will recur consistently across all exchanges, with few exceptions. There are requests, responses, and data. Each of them looks, by and large, like this:

```
<RequestIssued [priority= "SpacecraftEmergency | VehicleCritical
| MissionCritical | Noncritical"]>
  <Resource name= "RequestedResource">
    [<Time [start= "PT10M"] ... >]
      [<qty name= "qtyname" val= "qtyval" ... />]
      [<qty name= "qtyname" val= "qtyval" ... />]
      [<LocalFace.../>][<PointTo.../>][<X3Dobj .../>] [...]
    [</Time>]
    [<Time [start= "PT20M"] ... >] [...] [</Time>]
  </Resource>
</RequestIssued>
```

```
<ResponseOffered [option= "Immediate | Imminent | Possible |
Unavailable"]>
  <Resource name= "RequestedResource">
    [<Time [start= "P10M"] ... >]
      [<qty name= "qtyname" val= "qtyval" ... />]
      [<qty name= "qtyname" val= "qtyval" ... />]
      [<LocalFace .../>] [<PointTo .../>] [<X3Dobj .../>] [...]
      [<Cost>
        [<Time [stop= "P15M"] ... >]
        <Resource name= "Power">
          <qty name= "PowerCost" val="11" units="W"/>
        </Resource>
      [</Time>]
      </Cost>]
    [</Time>]
    [<Time [start= "PT20M"] ... >] [...] [</Time>]
  </Resource>
</ResponseOffered>
```

```
<ResponseData>
  <Resource name= "RequestedResource">
    <qty name= "qtyname" val= "qtyval" ... />
    [<qty name= "qtyname" val= "qtyval" ... />]
  </Resource>
</ResponseData>
```

```
<RequestWithdrawn>
  <Resource name= "RequestedResource" />
</RequestWithdrawn>
```

These patterns recur in nearly every message exchange: the RequestIssued contains a great deal of parametric information; the ResponseOffered contains the same type of parametric information, plus potentially costs (see below); ResponseData is more concise; and RequestWithdrawn includes only the name of the resource. Quantities, times, and costs elements constitute the most basic concepts needed by an AstroLogic application to make sensible decisions. Special consideration has been given to these messages as they may appear at almost any level of an exchange. These elements are discussed in detail below.

Quantity

The `<qty />` element is the workhorse of an Astrologic exchange. It allows concepts such as units, precision, reference frames, and other simple but essential expressions to be wrapped into a single data structure. 14 Watts, 7 centimeters plus or minus a tenth of a millimeter, or the +Z face of a module (or of the spacecraft as a whole) can all be expressed as a `<qty/>`.

Some examples of `<qty />` elements follow.

```
<qty name="IPAddress" val="10.128.0.1" />
```

(a unitless IP address of 10.128.0.1)

```
<qty name="TotalMass" val="8.2" units="kg" />
```

(total mass of the device is 8.2 kilograms)

```
<qty name="ModuleHeight" val="6" units="cm"
precision="0.001" />
```

(module height is between 5.999cm and 6.001cm)

```
<qty name="CG" val="[1.4 0 -22]" units="cm"
ref="ModuleRelative" precision="[0.1-0.2 0.01 0.01]" />
```

(module center of gravity is 1.4cm (+0.1, -0.2) up, 0cm (± 0.01) laterally, and 22cm (± 0.01) back, from the module origin)

Times

There are a broad variety of expressions of time that may be needed in a system like AstroLogic. Absolute time; relative time from now; relative time from an event; relative time from each occurrence of an event; durations; repeated moments and durations, all have a place and a need in the control of an embedded system. A concise and consistent description of time is critical if the message exchanges are to be efficient and not cumbersome.

Time is fundamentally described in accordance with the W3C XML Schema definitions of time, which in turn is

based on ISO 8601. In this framework, there are two basic time representations: an absolute time, in the format:

```
[-][Y*]YYYY-MM-DDThh:mm:ss[.s[s*]][[TZD];
and a relative time, indicated by a preceding "P", in the
format:
[-]PnYnMnDtnHnMn[.n[n*]]S.
```

For absolute time, negative years and years with greater than four digits are not entirely useful in AstroLogic applications, so research into those features is left to the reader. On the other end of the scale, arbitrary sub-second precision is allowed by the `ss[.s[s*]]` expression – that is, 04:16:22 and 04:16:22.4 and 04:16:22.428663 are valid times.

For relative time, any contiguous subset of the expression is allowed as long as it starts with P. Thus, P3Y is plus three years; P3Y2M is plus three years two months; P85D6H is plus eighty-five days six hours; PT-4M16.1S is four minutes sixteen point one seconds ago. Note that if any time (minute or second) element is included, the "T" marker must precede these elements. Relative times are based on some event; events such as "now" and "sunrise" are predefined.

With these expressions, an indication of all the combinations of time described above can be defined. All times are expressed as a periodically repeating duration, that is, a start time, an end time, and a repeat period. A time described with all three of these items will start at the start time, end at the end time, and then restart every repeat period after the original start. Subsets or degenerate configurations of these three values can define moments, single events, or isolated durations. That is, a moment is a time where stop equals start; singular occurrences have no repeat period; times which have a start but no end can be defined as having an end arbitrarily far in the future, or use the particular end event of "forever".

Some examples of `<Time />` elements follow.

```
<Time start="2004-09-14T19:28:55" stop="2004-09-15T07:00:00"
/>
```

(single timespan with absolute endpoints in GMT)

```
<Time start="2004-09-14T19:28:55" stop="P11H31M5S" />
```

(same timespan with absolute start, relative stop)

```
<Time startEvent="sunrise" start="P30S" stop="P10M" />
```

(single timespan describing the duration from 30 seconds to 10 minutes after the next sunrise)

```
<Time startEvent="sunrise" stopEvent="sunset" />
```

(the entirety of the next orbit day)

```
<Time startEvent="contactstart" stopEvent="contactend"
repeat="every" />
```

(the entirety of every communications contact)

```
<Time repeat="P5S" />
```

(a single moment recurring every five seconds)

```
<Time startEvent="sunrise" stopEvent="sunset"
repeatstopEvent="start" repeatstop="PID" />
```

(the next 24 hours' worth of orbit days)

Costs

In many situations, problems can be solved in a multitude of ways, and many resources can be provided through a number of different means. Knowing the cost of providing a resource, in terms of other resources, is often essential to determining which method is most preferable.

The `<Cost />` element does not have any unique attributes of its own, but collects a set of `<Resource />` elements to describe the cost of providing the encapsulating resource. Some of the `<Resource />` elements that exist within the `<Cost />` element will tend not to exist in any other context, so they are described here. The resources which will tend to be collected within a `<Cost />` element are:

- Electrical power, `<Resource name="Power">`
- Propellant or other consumable fuel, `<Resource name="Fuel">`
- A consumable other than fuel, `<Resource name="Consumable">`
- Requesters that will have to be denied a resource, `<Resource name="Requester">`

For each of these, specifics can be optionally provided using the `<Means />` element. The Requester of a resource should always indicate the priority of the request so that it can be evaluated by the responder should there be a conflict. An example `<Cost />` element using all of these types costs is as follows; this might be the cost of using a propulsion system to do attitude control:

```
<Cost>
  <Resource name="Power">
    <qty name="PowerCost" val="11" units="W" />
    <!-- It will take 11W to run the propulsion system -->
  </Resource>
  <Resource name="Fuel">
    <Means name="Propellant" />
    <qty name="FuelCost" val="0.01" precision="0.2-0.01"
units="%/s" />
```

```
    <!-- Every second, up to 0.2% of the fuel will be used,
with an average of 0.01% -->
  </Resource>
  <Resource name="Consumable">
    <Means name="ValveFatigue" />
    <qty name="ValveLife" val="0.3" units="%/s" />
    <!-- By using the propulsion system for AC, valves will
wear out faster than fuel -->
  </Resource>
  <Resource name="Requester">
    <RequestIssued priority="Noncritical">
      </RequestIssued>
    <!-- Some Requester had been granted a resource to
fulfill a non-critical need. This resource will have to be taken
away from this Requester to fulfill the new request. -->
  </Resource>
</Cost>
```

Service Message Types

As described earlier, the AstroLogic Dictionary is separated into various resource types. The following list of messages appear on specific services provided by the different resources:

- **Bus Reset** – A Bus Reset packet tells one or all participants on the bus that something extreme has happened (a power reset, malfunction of the Arbiter, or suspected malfunction of a specific recipient) and the device(s) are to start again from some condition, defined in the first parameter
- **Echo** – An Echo (or “Are-You-There”) exchange is the simplest packet transaction. The requester sends an Echo Request packet; the receivers send an Echo Response packet. The requester sends a Request Withdrawn packet to close the transaction.
- **Capabilities** – The Capabilities exchange lets the responder tell the requester the basic functionality this device can provide. It has two portions: a standard list of capabilities, and a custom list that can be defined by the manufacturer to describe specifics about the device.
- **Standard Telemetry** – The Standard Telemetry packet is a concise description of the physical device’s current state. It offers generic information that any physical system (such as a SMARTBus module) will contain, regardless of what functions that module performs.
- **Request For Request** – The Request For Request is a way for one device to encourage another to initiate an exchange. It can be a means for a more “intelligent” device to control a less capable one, or can allow a responder to supply a resource of its own volition. Often, a Request For Request is used to order events sensibly, as in a deployment

sequence (e.g., a communications module may not need to deploy its antenna yet, but a solar panel module may need the antennas to be deployed to get them out of the way of the solar arrays). For this reason, the receiver of a Request For Request should analyze the contents of the request being encouraged, to actually set off the chain of events required to make that request sensible

- **Physical Parameters** – The Physical Parameters exchange lets the responder tell the requester the size, shape, physical protrusions, and manufacturer-specific data that this device contains.
- **Mechanical Parameters** – The Mechanical Parameters exchange lets the responder tell the requester the height, position, and mass properties of the device.
- **Electrical Parameters** – The Electrical Parameters exchange primarily lets the responder tell the requester the power in and power out of a given module.
- **Logical Parameters** – The Logical Parameters exchange lets the responder tell the requester what data connections this device contains.
- **View Need** – The View Need exchange allows a module to ask other modules if a desired field of view is going to be blocked by another module's appendage. Ideally, no one will respond to this request, indicating the space is and is expected to remain free of obstruction. However, if another module has hardware that is obstructing that view – or if it has a deployable that will obstruct that view – that module must warn the requester. Note that this packet exchange is primarily for ground use to warn the user that interference is occurring, as little can be done on-board to avoid the obstruction. Since view needs are typically conical, view requirements are expressed as cone segments.
- **Deployment Plan** – A Deployment Plan packet announces to the spacecraft that at some point in the future, a hardware object is intended to move through a region of space external to the vehicle, and then occupy a region of space external to the vehicle. These regions need to be kept free of obstruction. Note that this packet exchange is primarily for ground use, to warn the user that an interference is occurring, as little can be done on-board to avoid the obstruction.
- **Point** – The Point exchange is the primary means of asking a Motion resource provider to aim the

spacecraft at a point in space. It allows the requester to specify a maximum spin rate that it can tolerate in the interest of getting the desired pointing precision. Pointing vectors can be described in two ways: an instruction to point toward a point in space referenced from some other point in space; or an instruction to point in a given direction in inertial space.

- **Pointing Knowledge** – The Pointing Knowledge exchange allows an attitude determination system to convey its knowledge about vehicle attitude to a requester that needs to know this information. Pointing in absolute space, or relative to some celestial body like the Sun, is available through this exchange.
- **MOSI** – The MOSI exchange is unique to situations (such as SMARTBus) where there is a basis from which other devices can measure their distance. This is in some respects the function of the Anchor, to be the one fixed point everyone else references from.
- **Conflict Warning** – A ConflictWarning exchange is the result of responses to a DeploymentPlan or ViewNeed exchange. It advises the ground system of a potential conflict. These warnings will occur in three different types of situations: first on the ground pre-launch, where the operator has the opportunity to adjust the module stack order, change the times that deployments occur, or recognize that the conflicts are unimportant; second is on-orbit repeats of these conflicts already known from pre-launch checkout tests; and third, improbable conflicts that occur unexpectedly during the mission that the spacecraft will attempt to deal with autonomously but the ground team must ultimately understand and resolve.

CONCLUSION

The development of standards – physical, mechanical, and logical – that find a balance between rigor and flexibility are critical to successful implementation of modular spacecraft. However, standards alone do not solve the problem. AstroLogic, a Plug-and-Sense technology, is a key enabler of modular, rapid-response spacecraft such as SMARTBus. Through careful layering of software, new technologies and approaches can be incorporated with minimal effort, and dangerous complexity that can often emerge in extensive software development efforts can be avoided.

The AstroLogic approach utilized on SMARTBus takes a very different approach than that of the traditional

spacecraft – it is driven by the resources provided by the system rather than the equipment selected for the system. By breaking the paradigm and shifting into a resource-based architecture, virtually any set of components to be integrated into a system – as long as they are compliant with the defined standards – without rewriting the software for each new design and tailoring it specifically to command the selected components for that particular configuration.

The Plug-and-Sense technology provides additional capability beyond what the more commonly known Plug-and-Play capability offers. It provides its physical characteristics – size, shape, center of gravity, etc. – in addition to the functional characteristics of each module. This distinction is critical, because these parameters are vital to the successful configuration of a spacecraft.

The application of this Plug-and-Sense capability to modular spacecraft allows spacecraft integration time to be shortened from what would normally take months on a standard spacecraft to something on the order of days

or even hours on a SMARTBus spacecraft. The successful implementation of AstroLogic could drastically change the way spacecraft integration is performed and ultimately enable the deployment of a truly rapid-response spacecraft.

ACKNOWLEDGMENTS

This material is based upon work administered under Contract Numbers DAAH01-03-C-R068 and W31PQ-04-C-R071 (DARPA), and F29601-03-M-0151 and FA9453-04-C-0218 (AFRL).

The authors wish to thank the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory (AFRL), and the Department of Defense Small Business Innovation Research (SBIR) Program for their support of the Phase I and II contracts under which this work was conducted. AeroAstro would especially like to acknowledge the contributions of Lt. Col. James Shoemaker, Ph.D. and Mr. Jeff Ganley, of DARPA and AFRL, respectively, for their continued consideration and support