Infotech@Aerospace 2011
29 - 31 March 2011, St. Louis, Missouri

AIAA 2011-1597

Space Plug and Play Avionics Application Programmers Interface for Radio Devices

Scott Dossey[1], Kevin J. Lynaugh[2], Matthew Davis[3] and Richard Middlestead[4]
Vulcan Wireless Inc. Carlsbad CA, 92008, USA

and

James Lyke[5], James Crane[6] and Calvin Roman[7]
Air Force Research Laboratory, Kirtland Air Force Base, Albuquerque NM 87117, USA

This paper describes a template for the embedded self-description of radio equipment for use in Space Plug-and-Play Avionics (SPA) systems. This template is a proposed standard that seeks to define a canonical description that might be universally applied to a variety of both simple and complex radio devices, including software-defined radios. The authors believe such template approaches promote the engineering of composable systems. A template API for communications is proposed in which a great variety of different radios (from any conforming vendor) might be interchanged and still produce an effective mission result (even as a personal computer might interchange a number of approaches for connecting to the internet without re-engineering the web browser or internet applications). This common API can be used regardless of communication medium, protocols, waveforms, or operating requirements. The challenge of designing a template is in striking a balance between simplicity in the representation of a canonical (generic) interface that still supports the rich expressiveness needed to accommodate sophisticated radio equipment.

Nomenclature

API	=	Application Programmer's Interface
SDM	=	Satellite Data Model
SDR	=	Software Defined Radio
SPA	=	Space Plug-and-Play Avionics, an emerging standard for plug and play satellite architectures
TEDS	=	Transducer Electronic Data Sheet
USB	=	Universal Serial Bus
UTF-8	=	UCS (Universal Character Set) Transformation Format – 8-bit. A Unicode text format that is backwards compatible with ASCII (American Standard Code for Information Interchange).
XML	=	eXtensible Machine Language
xTEDS	=	eXtensible Transducer Electronic Data Sheet

I. Introduction

Space Plug-and-Play Avionics (SPA) [1] is a building block approach to simplify and accelerate the pace of creating complex systems. It is based on a composability concept in which hardware and software elements can be treated as interchangeable "black boxes" that can be flexibly and scalably arranged to form systems. The key concepts include:

- Single-point electrical interfaces;

[1] Senior Software Engineer, Vulcan Wireless Inc., 1935 Camino Vida Roble, Suite 150A.
[2] President, Vulcan Wireless Inc., 1935 Camino Vida Roble, Suite 150A.
[3] Principal Engineer, Vulcan Wireless Inc., 1935 Camino Vida Roble, Suite 150A.
[4] Director of Communications, Vulcan Wireless Inc., 1935 Camino Vida Roble, Suite 150A.
[5] Technology Advisor, U.S. Air Force, Principal Electronics Engineer, Space Vehicles Directorate, 3550 Aberdeen Avenue SE, Associate Fellow, AIAA.
[6] Electronics Engineer, Space Vehicles Directorate, 3550 Aberdeen Avenue SE.
[7] Electronics Engineer, Space Vehicles Directorate, 3550 Aberdeen Avenue SE.

- Self-describing and automatically discoverable components;
- Self-organizing networks that link power, electrical signals, timing, and software data between components automatically;

These features have the potential to greatly simplify the process of system integration. However, self-describing datasheets cannot address all the API interoperability needs for devices. A computer cannot read a self-described datasheet and then automatically write application software for a self-described device. A human must still code these applications, and must code to a specific API. The eXtensible Transducer Electronic Datasheet (xTEDS) of SPA can be used chiefly to define formal interfaces which can be electronically enforced by SDM. However, there is no requirement that two similar devices have similar xTEDS API. The open-ended nature of the XML-based self-description approach used in xTEDS allows an effectively infinite number of ways to implement device functionality. Each vendor, left to their own devices, is likely to come up with a unique xTEDS API for each device. This leads to a situation where each device effectively "plugs", but doesn't "play". The solution is to define a strict API for each different type of device.

Many commercial plug-and-play standards, such as the universal serial bus (USB), have strict device API definition. USB achieves this with the notion of *class* specifications for commonly used peripheral devices. Each class specification defines precisely what application programming interface (API) functions are available and how they are used for a particular type of device. In the *USB Mass Storage Device Class*, for example, an interfacing standard is defined for all block storage devices. This class concept allows the same driver code to be used for USB drives of all kinds (from "thumb drives" to redundant disk arrays). Similarly, the *USB Human Interface Device Class* defines an interfacing standard for keyboards, mice, and game peripherals. The USB class specification concept is powerful; only a few drivers have had to be written to accommodate a vast range of devices.

In this paper, we propose a SPA-based interpretation of a generalized communications API, a framework that may permit a diversity of radio devices to be accommodated for the flexible engineering of rapidly composable systems having communications requirements. This API template at one level serves as a unified, canonical representation of communications devices. It can be used in radios regardless of communication medium, protocols, waveforms, and operating requirements. The challenge of designing a template is in striking a balance between simplicity in the representation of a canonical (generic) interface that still supports the rich expressiveness needed to accommodate sophisticated radio equipment.

The paper is organized as follows. In the next section, we briefly describe the xTEDS concept. We then introduce the SPA "CommunicatorInterface" as an API format for use in xTEDS. The rest of the paper is devoted to more detailed definitions of the API.

II. eXtensible Transducer Electronic DataSheet (xTEDS)

The xTEDS is an embedded electronic datasheet, inspired by the pioneering work of Kang Lee (NIST) and the team who developed the IEEE 1451 smart sensor standards [2]. Embedded datasheets were inspired by complex systems, such as factories having thousands of disparate sensors that over time might fail and need replacing. It is conceivable that vendors making such sensors stop making them or go out of business, etc. Standardization across a wide diversity of sensor components was considered impractical, leading to a potential crisis when replacing one sensor with an analogous one from another supplier that would likely be incompatible, forcing a redesign of the equipment interface. On the scale of thousands of sensors, such an approach is undesirable, leading to the notion that smart sensor standards might be created to simplify and automate the replacement and integration of disparate sensors of many types from many vendors.

In SPA, the concept of xTEDS was created to do a similar thing with complex systems. The primary difference between TEDS and xTEDS is that while TEDS is best suited for simple, scalar sensors, such as thermometers and pressure gauges, the xTEDS is intended to support arbitrarily complex components, as well as simple ones. The original TEDS predated the existence of XML and was intended for implementation on simpler microcontrollers having limited, fixed memory maps. In xTEDS, flexibly is achieved by using an open-ended concept, in which XML is used to describe the properties, measurands, and commands (the "knob-ology") of SPA devices.

To illustrate the structure of a typical xTEDS template, a very simple xTEDS for a thermometer (from []) is shown in Figure 1. The xTEDS is based on a header, component "device" declaration, and one or more *interface* definitions. These interfaces contain one or more *message* definitions, which are comprised of references to one or more *variables*, declared elsewhere within the same overall interface definition. Interfaces, devices, applications (pertaining to a software xTEDS, not shown in Figure 1), messages, and variables are structural *elements* within an xTEDS hierarchy (see [2] for a complete list of xTEDS elements). The distinction between attributes and qualifiers

is structurally clear. Attributes (such as “kind”) are delineated in the head of an element declaration. Qualifiers are themselves elements, which are composed of attributes. The semantic distinction between qualifiers and attributes is less clear.

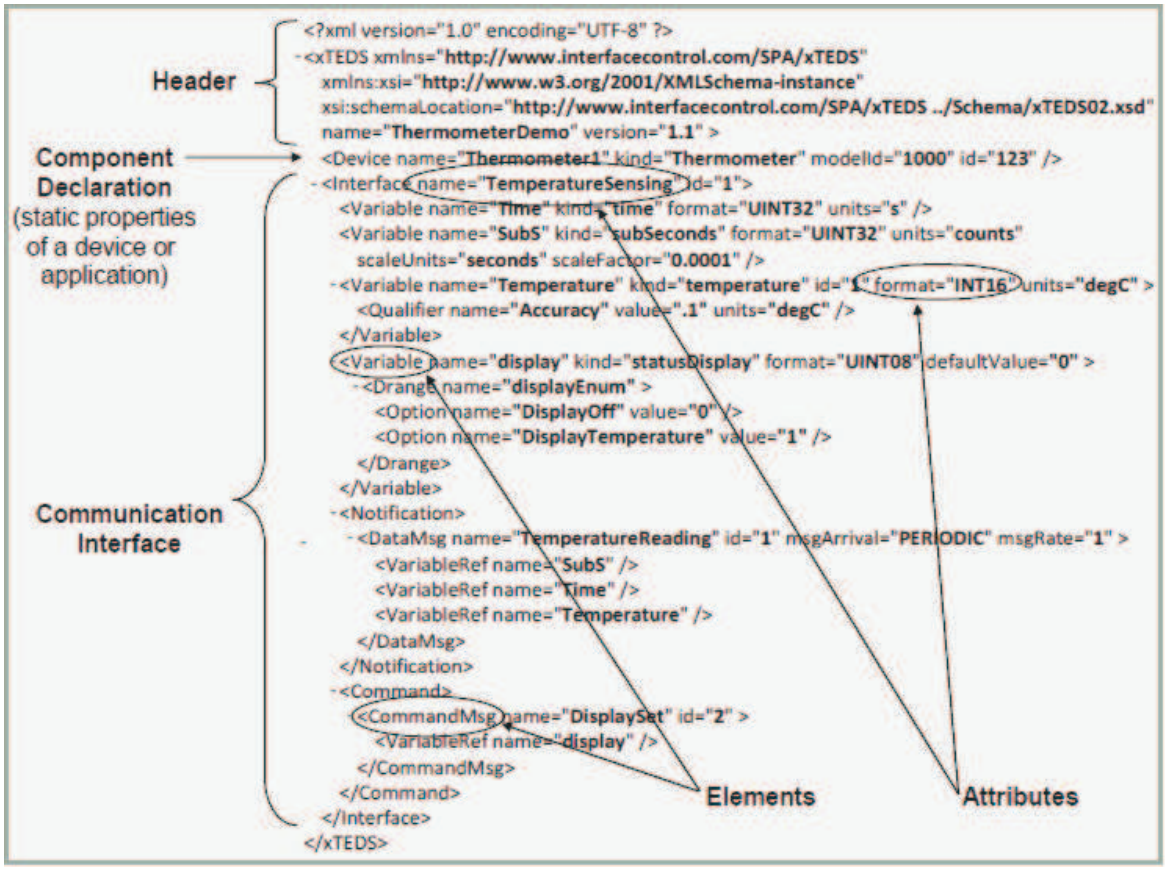

Figure 1. An example eXtensible Transducer Electronic Datasheet.

As such, the notion of xTEDS, while having some structure, can be variously defined. Without regulation, the structure can be rendered in many different ways, and the names of elements can be arbitrarily chosen. The notion of a common data dictionary emerged early on in an attempt to regulate at least the naming consistency of the “variable” elements, especially pertaining to units of measure, but the potential of abuse without strong discipline remains. In reference 2 a number of xTEDS reformation concepts have been proposed, and many have percolated into the discussions of SPA standards and technology development.

II.A. “Class” Approach for SPA

Under the banner of “xTEDS reformation”, one approach that has been discussed is the notion of a canonical template for SPA devices. This is an approach that amounts to a regularized API that is controlled for specific devices. As previously discussed, USB employs a class specification approach to constrain the open-ended-ness of device specifications. Similar to the USB class specification, where a “mouse is a mouse” (or a *Human Interface Device* at least), the template API makes the notion of a reaction wheel, for example, consistent across the widest variation of reaction wheels. This means that the reaction wheel API is specified through a regulated xTEDS interface element. All conforming reaction wheels use the same interface element “template” or “API”. This “canonicity” is important in that it makes design of the application software that relies on devices more predictable. Without canonicity, applications must be written to anticipate all possible stylistic variations of a (ex. reaction wheel) interface, making the application brittle to any new reaction wheel interface that might be written in the future.

Canonicity is not a panacea. Since even an occasional computer mouse may have an odd button or feature (deviating from the canonical notion), it is important to accommodate idiosyncrasies in SPA template APIs. For this need, one can use a special *OEM Interface*, in which the differences from the canonical “norm” can be fully

expressed. Analogous to a mouse with extra buttons, which ultimately requires a special driver to exploit those extra features (but still actually functions as a normal mouse without it), the reaction wheel with odd "bonus" features, special interior temperature readings, or an infinite number of other unique things, can containerize these non-canonical features in an OEM interface.

The USB class specification approach is also not a panacea; it has limitations. It is generally only useful when many different devices replicate identical functionality. For point to point communications, the USB standard has a *USB Communications Device Class*. But there are many possible communications standards and modalities, and this class specification cannot magically support them all. The specification consequently makes no attempt to do so. As a result, it is tremendously complicated, having up to five distinct subclasses for commonly supported communications equipment. Each subclass has unique configuration needs and requirements, and the five subclasses still cover only a small subset of commercially available communication devices.

In SPA, given the vast breadth of types of missions and communication requirements to various possible systems, a similar situation exists. It seems difficult to define a common framework for radio communication devices given the diversity of radio types, waveforms, and data protocols. In the next section, a proposed SPA communications framework API is described that attempts to strike the balance between expressive capacity and simplicity.

III. The CommunicatorInterface

The canonical template or "API" for SPA-based communications devices is an xTEDS description. The xTEDS description will be referred to as the *CommunicatorInterface*. The goal of the *CommunicatorInterface* API is to generically allow for many possible radio devices to be created as plug-and-play modules. In being generic, many different communications configurations can be implemented with a consistent structure, supporting variations to buffer handling, flow control logic, and frame handling logic between implementations. It is expected that the API is also fully capable of handling multiple communication devices and multiple channel communications setups. Each physically separate communication device (e.g., radio) contains an xTEDS having an interface element referred to as the SPA *CommunicatorInterface*. The presence of this interface signifies that the device conforms to a *SPA Communications Device Class*.

The API commands and notifications supported by the *CommunicatorInterface* consist specifically of a limited set of canonical commands (meaning that they are not to be amended or modified by an implementer). Idiosyncratic functionality that exists outside the scope of this interface would be relegated to the aforementioned "OEM interface".

IV. **Hierarchical Configuration Space**

To the degree possible, it is desirable to have radio devices to take on the appearance of simple black boxes that communicate.

Most non-trivial communications systems are built in a hierarchy of networking layers, as suggested in Figure 2. In this case, a SPA radio device contains a single *CommunicatorInterface*, defining two distinct physical radios within the same device. In the hierarchy of layers shown, the physical layers are "children" of a root layer that contains the general "configuration space" of the entire device.

Each layer may have its own configuration space, independent of other layers. For example, the following may need to be configured for a radio link:

- Which *CommunicatorInterface* is being used. This is implicitly specified by the use or non-use of the appropriate physical and logical software interfaces.
- For a particular radio device with one *CommunicatorInterface*, multiple internal radios may exist.
- Each internal radio may have multiple communication mediums that support different frequencies, channels, or ground test interfaces that it operates on perhaps simultaneously.
- For each medium, multiple waveforms and framing options may be possible, perhaps simultaneously.
- For each possible set of low level waveforms and framing options, the network protocol or protocols used over them may need to be configured.
- There may be possible nesting and layering of communication protocols.
- Each individual communications session under a protocol may have constraints that need tweaking.

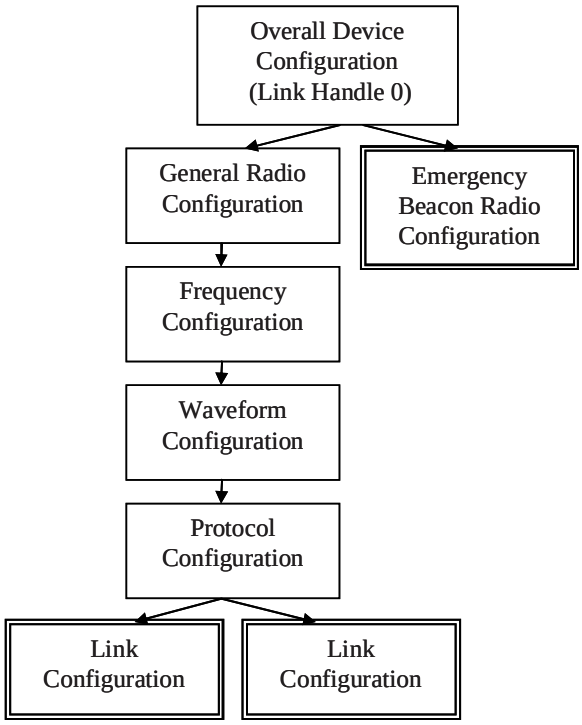

Figure 2. Possible Configuration Hierarchy. *This shows one device with two internal radios, one of which is general purpose, one of which is only used for emergency transmissions. Boxes with double lined borders represent configuration spaces whose handles can be directly used to initiate a radio link via a* ***LinkActivate*** *command.*

Each one of these configuration spaces may have unique configuration needs, and it is best to separate and modularly isolate each one if possible. Exactly how the configuration hierarchy is arranged can vary from device to device. Given that the configuration for different types of network layers varies wildly, there is a deliberate separation of configuration functionality from use functionality. Configuration parameters necessary to configure vary dramatically based upon protocol, waveform, and media used. The CommunicatorInterface standard API makes no attempt to standardize these configuration spaces currently (though they should be standardized in the future). Each can currently be customized with their own textual configuration parameters. Once a link's configuration space has been configured, however, the API defines a canonical interface for communications over a link.

V. **Link Handles**

Each configuration space has its own link handle. Link handles are dynamically assigned as integers, each associated to a configuration space. The integer "0" is a special link handle, being always reserved for the

commands and notifications of a particular overall SPA communications device (i.e., the topmost layer). Link handles are used to associate commands and notification messages with configuration spaces.

VI. Creating and Destroying Link Handles

Link handles can be created via the **CreateLink** command message and can be destroyed via the **DestroyLink** command message.

VI.A. CreateLink Command Message

This requests a new link handle to be allocated and returned via the **LinkCreated** Notification Message. This is used to request a new link handle to a new associated configuration space. Since the configuration space is hierarchical, a parent configuration space is specified via the **ParentLinkHandle** parameter. Parent configuration spaces do not always have to support **CreateLink** functionality, or may only support it under certain specific configurations.

Parameters:

- An integer **ParentLinkHandle** must be passed specifying an open parent link. Link Handle 0, which always represents the device as a whole as shown in Figure 2, is always available and does not have to be created.

Responses:

The device responds with either a **LinkCreated** notification, or an **Error** notification.

VI.B. LinkCreated Notification Message

This is sent either in response to a **CreateLink** command message, or sometimes can be sent spontaneously to signify that the device has spontaneously created a new configuration space. Spontaneously created configuration spaces are intended to be used to represent links for handling incoming unsolicited connections. The message returns a new **LinkHandle** that is unique to a *CommunicatorInterface*.

Notification Fields:

- An integer **ParentLinkHandle** specifying the configuration space the new **LinkHandle** is created in.
- An integer **LinkHandle** specifying a newly allocated link handle to a new configuration space.
- A boolean **SpontaneousFlag** specifying whether the creation was requested by a **LinkCreate** or not.

VI.C. DestroyLink Command Message

This frees resources associated with a configuration space specified by a LinkHandle. This also implicitly destroys all children configuration spaces under the specified one. Generally speaking, this should free resources immediately, and not attempt to cleanly close down links by communicating closing state information to remote stations (see DeactivateLink).

Parameters:

- An integer **LinkHandle** must be passed specifying a link to destroy. Link handle 0 is a special case which destroys all configuration space except the one specified by link handle 0.

Responses:

The device responds with either a **Success** or **Error** notification message as appropriate.

VII. Identifying Configuration Space Capabilities

Each configuration space has certain associated capabilities. These can be identified via the **GetIdentity** command message. The identity information is returned via an **Identity** notification message.

VII.A. GetIdentity Command Message

This requests identity information for a specific link handle. Identity information describes key attributes of a the configuration space that the link handle references.

Parameters:

- An existing integer **LinkHandle**.

Responses:

The device responds with either an **Identity** or **Error** notification message as appropriate.

VII.B. Identity Notification Message

This is sent in response to a **GetIdentity** command message.

Notification Fields:

- An integer **LinkHandle** specifying which **LinkHandle** was passed to the **GetIdentity** call that triggered this Identify notification.
- An integer **ParentLinkHandle** specifying the configuration space the new **LinkHandle** was created in.
- A string **ConfigurationSpaceName** identifies the type of configuration space. Configuration spaces that have not been standardized be of the form: <VENDOR NAME>:<CONFIGURATION SPACE NAME>. For example "VULCAN WIRELESS:LEOPARD PROTOCOL" would be the name of a configuration space before standardization. After standardization it would just be "LEOPARD PROTOCOL".
- An integer **ConfigurationSpaceVersion** which will specify a single revision number for the configuration settings of a configuration space.
- A string **ImplementationName**. For the top level radio configuration space (link handle 0) this would be the name of the radio device. For a protocol configuration space, this might be the name of the software that implements the protocol, etc.
- The three integers **ImplementationVersionMajor, ImplementationVersionMino**r, and **ImplementationVersionRelease** specifying major, minor, and release version numbers for a configuration space implementation.
- A **FramingType** enum which can specify framed, unframed, or unconfigured packets. Framed data is packetized and aligned to some boundary (like an IP packet), whereas unframed transmission is a continuous stream of data (like a serial port). Technically some systems are partially framed (like a serial port frames each specific byte with start and stop bits), but in general, this refers to whether data arrives in cohesive chunks, or as a continuous stream.
- An integer **MinFrameSize**
  -1 is a special value meaning not applicable or dependant on situation.
- An integer **IdealFrameSize**
  -1 is a special value meaning not applicable or dependant on situation. This value otherwise must be between MinFrameSize and MaxFrameSize.
- An integer **MaxFrameSize**
  -1 is a special value meaning not applicable or dependant on situation
- A **CommunicationCapability** enum which can specify none, tx_only, rx_only, half_duplex, full_duplex, or unconfigured as appropriate to configuration space.
- A **Checksummed** enum which specifies data integrity checking strength. This can be unconfigured, none, checked, or authenticated. "Checked" is used for any system that checks payload data (not just headers) against a checksum or cyclic redundancy check of any strength. "Authenticated" should be reserved for systems that use a cryptographic authentication process that guarantees the data hasn't been tampered with.
- An **Acknowledgement** Boolean flag which indicates the communications system used in the configuration space sends back acknowledgements guaranteeing that data has been sent. This is set to false if the configuration space does not handle links.
- An integer **InstanceCount** which specifies a limit to configuration spaces instantiable under the one specified by **LinkHandle**.
  -1 is a special value meaning resource limited, many, or currently undeterminable.
- An integer **OpenInstanceCount** which specifies how many configuration spaces are instantiated under the one specified by **LinkHandle**.

VIII. **Configuration of a Context**

Each configuration space may need configuration (unless it has suitable defaults). Possible configuration parameters are endless, but some examples are:

- Radio operational mode
- Frequencies
- Spread Codes
- Waveform types
- Data Rates
- Station IDs
- Terms of Service
- Transmit power Level

This API makes no attempt to capture all possible configuration options into a massive conglomeration of parameters. Instead, it merely specifies that each configuration space can be configured via a configuration string.

Some simple examples:

- Media configuration:
  - "tx_frequency=375000000 rx_frequency=421000000"
  - "channel=6 doppler_search_range=10000"
  - "channel=diagnostic_port"

- Waveform configuration:
  - "waveform=mil181 options=137,143,144"
  - "tx_waveform=gsm tx_bps=9600 rx_waveform=soqpsk rx_bps=19200"
  - "waveform=soqpsk viterbi=1/2 data_rate=auto_scale"

- Protocol configuration:
  - "protocol=tcp local_host=192.168.1.1"
  - "protocol=mil184 local_station_id=1 fec_rate=3/4 probing=on arq=crc retries=5 compression=none satellite_delay=20 radio_delay=5"

- Link configuration under a protocol:
  - "remote_host=tacsat12.afrl.gov port=375"
  - "remote_station_id=234"

Ideally each possible configuration space type would ultimately have a standardized specification for specifying options. Additionally, general guidelines for formatting and parsing configuration strings exist. Full details will be published with the full specification of this API. In general configuration strings are made up of fields of the form "parameter=value" separated by spaces. Textual configuration strings were chosen because they are easy to communicate and relate to device vendors when troubleshooting products.

The API configuration command messages are: **SetConfiguration**, **GetConfiguration**, **SetConfigurationParameter**, and **GetConfigurationParameter**.

VIII.A. SetConfiguration Command Message

This sets a configuration string for a configuration space. This is intended to set up all configuring parameters for a configuration space at once. Configuration from previous **SetConfiguration** or **SetConfigurationParameter** commands must be reset. The validation of the configuration at the time of a **SetConfiguration** should be to the syntax used by the configuration space only, and no attempt should be made to validate it against parameters or resources used by other configuration spaces unless it changes the behavior of an already activated link. Otherwise full validation of viability should not occur until an appropriate **LinkActivate** occurs.

Parameters:

- An existing integer **LinkHandle** to the configuration space to be configured
- A **ConfigString** and **ConfigStringLength** specifying a UTF-8 configuration string and its byte length.

Responses:

The device responds with either a **Success** or **Error** notification message as appropriate.

VIII.B. GetConfiguration Command Message

This gets a configuration string for a configuration space. This is intended to read back all configuration state for a configuration space at once. All possible configuration state information that can be set with **SetConfiguration** or **SetConfigurationParamete**r commands must be represented in the response.

Parameters:

- An existing integer **LinkHandle** to the configuration space whose configuration is to be read

Responses:

The device responds with either a **ReturnConfiguration** or **Error** notification message as appropriate.

VIII.C. ReturnConfiguration Notification Message

This returns the configuration string for a configuration space in response to a **GetConfiguration** command message. The string returned does not have to exactly match that sent by an earlier **SetConfiguration**, and should contain explicit values for default configuration parameters.

Notification Fields:

- An existing integer **LinkHandle** specifying the link for which configuration information is to be returned.
- A **ConfigString** and **ConfigStringLength** specifying a return UTF-8 configuration string and its byte length.

VIII.D. SetConfigurationParameter Command Message

This sets a single configuration parameter for a configuration space, changing the existing configuration one parameter at a time. Validation of **SetConfigurationParameter** command should be limited to syntactic validation only of the resulting configuration. No attempt should be made to validate it against parameters or resources used by other configuration spaces unless it changes the behavior of an already activated link. Otherwise full validation of viability should not occur until an appropriate **LinkActivate** occurs.

Parameters:

- An existing integer **LinkHandle** to the configuration space to be configured
- A **ParameterNameString** and **ParameterStringLength** specifying a UTF-8 parameter name string and its byte length.
- A **ConfigString** and **ConfigStringLength** specifying a UTF-8 configuration for a parameter string and its byte length.

Responses:

The device responds with either a **Success** or **Error** notification message as appropriate.

VIII.E. GetConfigurationParameter Command Message

This gets the configuration of a particular configuration parameter for a configuration space. This is intended to read back the configuration of one parameter in the configuration state only.

Parameters:

- An existing integer **LinkHandle** to the configuration space whose configuration is to be read
- A **ParameterNameString** and **ParameterStringLength** specifying a UTF-8 parameter name string and its byte length.

Responses:

The device responds with either a **ReturnConfigurationParameter** or **Error** notification message as appropriate.

VIII.F. ReturnConfigurationParameter Notification Message

This returns the configuration string for a configuration space in response to a GetConfigurationParameter command message.

Notification Fields:

- An existing integer **LinkHandle** specifying the link for which configuration information is to be returned.
- A **ParameterNameString** and **ParameterStringLength** specifying a UTF-8 parameter name string and its byte length.
- A **ConfigString** and **ConfigStringLength** specifying a return UTF-8 configuration parameter string and its byte length.

IX. Activating Links

A link is not ready to be used just because its configuration space has been set up. In order to use a link it must be activated. Furthermore, since configuration occurs hierarchically, and a configuration for a particular configuration space may depend on multiple layers of configuration spaces, final configuration checking for a link may have to be deferred occur until activation is attempted. For the purpose of controlling link readiness there are the LinkActivate and LinkDeactivate command messages. For the purpose of signaling link readiness there are the LinkInitiated and LinkTerminated notification messages.

IX.A. LinkActivate Command Message

When a link has been configured, and is ready for use, the link is activated using **LinkActivate**. Note that before data can be received on the link you will need to use the **ReceiveReady** command. It is not necessary to activate parent link handles for configuration spaces which do not allow actual communication links to be created in them. It is however, required to activate lower layer network layer links, when lower level network layer links are capable of being used directly by the API to instantiate lower level links. Even link handles created via a spontaneously received **LinkCreated** messages must be activated before use. If spontaneously created link handles are not activated, no signal should be sent to the remote system of any response to their attempt to communicate. For some implementations, **LinkActivate** needs to be called rapidly after a spontaneously received **LinkCreated** command or the possibility of connection may be lost.

Parameters:

- An existing integer **LinkHandle** specifying the link you are activating.

Responses:

The API responds with either a **LinkInitiated** or an **Error** notification message as appropriate.

IX.B. LinkInitiated Notificaiton Message

This is the successful response to a LinkActivate command. It shall only occur in response to a LinkActivate command.

Notification Fields:

- An existing integer **LinkHandle** specifying the link activated.

IX.C. LinkDeactivate Command Message

This cleanly closes or deactivates a link, notifying the other side of the link that the link has been closed if such a mechanism exists. **LinkDeactivate** can in some cases take considerable time to try to disconnect cleanly before timing out.

Parameters:

- An activated integer **LinkHandle.**

Responses:

The API will eventually respond with either a **LinkTerminated** or an **Error** notification message as appropriate.

IX.D. LinkTerminated Notification Message

This can be either the successful response to a LinkDeactivate command, or can occur spontaneously in response to the link being closed on the other end or timing out. Receiving a **LinkTerminated** message means that the user can no longer transmit or receive on the link.

Notification Fields:

- A previously activated integer **LinkHandle** that has been deactivated/closed and can no longer be used for transmission or reception of data.
- An enum for **TerminationType**:
  - clean – The link was closed due to a successful LinkDeactivate (that notified the other side if possible), or by the other sides notification of closure.
  - unclean – The link was terminated due to a transmission timeout or a **LinkDeactivate** (that attempted to notify the remote side of termination, but was unable to do so properly).

X. Flow Control

One critical aspect of many communications is flow control. When communicating there is often the possibility of one person transmitting data faster than the receiver's capacity to handle the data. There are several basic strategies for dealing with flow control issues:

- Insure that the sender cannot send data faster than the receiver's ability to process the data.
- Notify the sender to stop or slow down before the receiver runs out of capacity for buffering the data. The exact nature of how this is done will be protocol specific.
- Keep retransmitting until acknowledgement (usually with a back-off timeout to avoid jamming the channel and wasting power). Note for protocols that use acknowledgements this is often the ultimate fallback mechanism for flow control.

Since an underlying protocol may change its flow control strategy based upon how much buffer space is available (in the case of throttling speeds), this information needs to be sent through the API. Flow control is accomplished via the **ReceiveReady** command message, and the **TransmitReady** notification message. These commands are discussed before discussing basic Transmit and Receive commands because they fundamentally control when and how data is transferred.

X.A. ReceiveReady Command Message

This command message lets the radio know how much the user is prepared to receive on a particular link. The information sent by this command is then used primarily to implement flow control. You must use this command on a link to enable the receiving of data. The parameters to the **ReceiveReady** command tell the API the maximum permitted amount of data that it can send **Receive** notifications (until another ReceiveReady updates this quantity). Data that may be received by the implementation before it can send flow control data back should be buffered internally to the implementation (unless there is no flow control). Also, an implementation may elect to buffer some data internally that it cannot yet send to the API because a **ReceiveReady** command has not been sent. In the case that the implementation has no flow control, data can be lost if this command is not sent in a timely fashion.

Parameters:

- An integer **LinkHandle** specifying a link.
- An unsigned integer 32 bit **ReceiveSafeLength** specifying how much more data it is now safe to receive. This number should be set to the maximum buffer space left on the receiver minus the amount of previous **ReceiveSafeLength** still authorized. Setting it lower and then enabling more little by little may cause protocols to thrash under excess flow control. Note that this value can also be set to zero, in which case there is no extension of available receive buffer space. It can also be set to the special value 0xFFFFFFFF which turns off all flow control, setting the receive data limits to infinity.

Responses:

The device responds with either a **Success** or **Error** notification message as appropriate

X.B. TransmitReady Notification Message

This notification message lets the user know when it is safe to transmit on a link, and how much data can be safely transmitted. This message controls when and how much you can transmit on a link for several purposes. First,

it prevents buffering overflows as during transmit. It is also tied in with protocol flow control. Furthermore, it will prevent you from filling a frame without sending a **TransmitFrameEnd**. In very raw "bits in/bits out" modes it can inform the user of time windows when communication is possible (time slice transmission window, Doppler lock, etc). Note that a **TransmitReady** may also come with a time limit, specifying how long data can be sent before the window of opportunity closes. This time limit may represent many things, including when buffer underflow might occur, when a transmission window ends, or the end of a doppler estimate's validity. After the time limit, another **TransmitReady** will be sent with an update. The new **TransmitReady** may have **TransmitSafeLength** of zero, specifying that transmission is not ready at this time. The time limit is always slightly underestimated, to prevent another **TransmitReady** coming in while a **Transmit** command is taking place. As long as the underestimated time limit is followed, this will not happen. A given **TransmitReady** is only valid until the next one comes, which can happen for one of three reasons: time limit expiration, transmission update, or size/time extension. In the case of time limit expiration, a new **TransmitReady** is sent reflecting the new time limit and a new safe length for transmitting. Transmission updates occur after a **Transmit** command has been sent and reflect the new amount of data safe to transmit. Size/time extensions occur when the **TransmitSafeLength** or **TimeLimit** can be safely extended. In the case of a size/time extension, the **TransmitSafeLength** can only go up or stay the same, and the **TimeLimit** cannot go down more than the amount appropriate to the time interval since the previous **TransmitReady.** Users of the API should note that there may of course be small discrepancies between their internally tracked time limit left, and the time limit returned, but they can safely choose the larger of the two as the remaining time, given that they have at least 2 millisecond precision.

Notification Fields:

- An integer **LinkHandle** specifying the link for which the **TransmitReady** status is being updated.
- An unsigned integer **TransmitSafeLength** specifying how much data it is now safe to transmit. Note that this can also be zero. Zero means that it is not safe to Transmit at all. It is however, safe to perform a **TransmitFrameEnd** in this case, if and only if **FrameEndFlag** is set.
- An integer millisecond **TimeLimit** limit setting the amount of time that this **TransmitReady** is valid. This is stored as an unsigned 32 bit value. This value is always underestimated relative to the real time limit. Also it is possible for the **TimeLimit** to be 0xFFFFFFF, which actually means that the user has an infinite time to perform the Transmission in, or in the case that the **TransmitSafeLength** is 0, that the time limit is infinite until you receive a new **TransmitReady**. It is also possible when the **TransmitSafeLength** is zero for the **TimeLimit** to be suddenly updated with a decreased amount.
- A **TransmitUpdateType** enum that specifies what type of update has occurred (spontaneous, extension, or transmission_update):
    - A spontaneous_update occurs when a link is first initiated, a time limit is expired, or any time after the system leaves a case where **TransmitSafeLength** is zero (other than by **TransmitFrameEnd)**.
    - An extension occurs when there is a size/time extension. The new values of the other parameters represent the new greater size or time left.
    - A transmission_update occurs after each **Transmit** and **TransmitFrameEnd** command
- A boolean **FrameEndFlag** that specifies that when **TransmitSafeLength** is exhausted, no more data will be made available by **TransmitReady** until a **TransmitFrameEnd** for the **LinkHandl**e occurs.

XI. Transmit and Receive

Two commands are used to send and receive buffers over a communications link, the **Transmit** command message, and the **Receive** notification message.

XI.A. Transmit Command Message

This command transmits data for a particular link handle. **Transmit** commands can only be used with activated link handles (activated via **LinkActivate**), and for which a **TransmitReady** notification has been received. After each successful **Transmit** a new **TransmitReady** notification message will arrive with an update as to the status of how much can be transmitted.

Parameters:

- An integer **LinkHandle** specifying a link.
- A **TxBuffer** and **TxBufferCount** specifying a byte buffer and its size to transmit.

Responses:

The API will respond with either a **TransmitReady** or **Error** notification message. In the case of a **TransmitReady** the **TransmitUpdateType** will be set to transmission_update, and the transmission can be considered a success.

XI.B. Receive Notification Message

When data is received, it comes in via this message. A link must be activated with **LinkActivate**, and flow control must be configured with **ReceiveReady** command before this notification message will start arriving.

Notification Fields:

- An integer **LinkHandle** specifying the link that the data is coming in on.
- An **RxBuffer** and **RxBufferCount** specifying a byte buffer and its size.

XII. Handling Framed Communication

Many communication transport layers have some sort of natural packet or burst boundary, whereas others don't. Packets are often limited in size, and the alignment of data inside packets can be critical. For this reason, it becomes critical to export the existence of framing and/or packet boundaries in this general purpose communications API. This is done via the **TransmitFrameEnd** command message, and the **ReceiveFrameEnd** notification message.

XII.A. TransmitFrameEnd Command Message

This message, regardless of underlying implementation, should act to send any buffered outgoing data over the radio link. For implementations that require framing or packetization, it will also act to end a frame or packet. Depending on the implementation, data sent via a **Transmit** command may be transmitted before a **TransmitFrameEnd** is sent, and in such cases it may be possible to have a transmission underflow error occur if internal implementation buffers are not filled fast enough via **Transmit** commands before a **TransmitFrameEnd** is sent.

After each **TransmitFrameEnd** command, a new **TransmitReady** notification message will arrive with an update as to the status of how much you can transmit safely.

Parameters:

- An integer **LinkHandle** specifying the link handle whose frame should be ended and/or whose buffered data should be sent out on the radio link.

Responses:

The API responds with either a **TransmitReady** or **Error** notification message. In the case of a **TransmitReady** the **TransmitUpdateType** will be set to transmission_update, and the transmission can be considered a success.

XII.B. ReceiveFrameEnd Notification Message

When data is received, it comes in via this message. A link must be activated with **LinkActivate**, and flow control must be configured with **ReceiveReady** command before this notification message will start arriving.

Notification Fields:

- An integer **LinkHandle** specifying the link that the data is being transmitted on.
- An **RxBuffer** and **RxBufferCount** specifying a byte buffer and its size.

XIII. Status Updates

Maintained with each configuration space is a set of status variables common to all configuration spaces. It is assumed that these variables change after every **Transmit** command (and associated response), **Receive** notification, **TransmitFrameEnd** command (and associated response), **SetConfiguration** (and associated response), **SetConfigurationParameter** (and associated response), and **ReceiveFrameEnd** notification. For these changes in status, the user of the API is not notified explicitly. Any change that occurs otherwise, or is not related to normal processing of these commands, triggers a **StatusUpdate** notification message. To read status, the **GetStatus** command message and **ReturnStatus** notification message are used. Additionally, for status and telemetry that is not covered by the ReturnStatus notification message and is specific to a particular configuration space type,

GetTelemetry and **ReturnTelemetry** are available. Generally, this telemetry should not be essential to the operation of the link, but is made available for instrumentation purposes.

XIII.A. StatusUpdate Notification Message

Notifies the user of a configuration space that its status, configuration, or Telemetry has changed due to an event that the user has not initiated or been notified of, or which is not normally or consistently updated by such events.

Notification Fields:

- An integer **LinkHandle** specifying the configuration space whose status has changed.
- A **StatusChanged** boolean. This is true if status variables have been updated unexpectedly.
- A **ConfigurationChanged** Boolean. This is true if the configuration changed unexpectedly, rather than the status.
- A **TelemetryChanged** boolean. This is true if telemetry specific to a particular implementation of a link that is not covered by that return in ReturnStatus has changed.

XIII.B. GetStatus Command Message

This command is used to request status information for a configuration space. This command can be called on any created (but not destroyed) link handle, even ones for whom a **LinkTerminated** notification has been received signifying deactivation.

Parameters:

- An integer **LinkHandle** specifying the link handle for whose configuration space the user is requesting status.

Responses:

The API responds with either a **ReturnStatus** or **Error** notification message.

XIII.C. ReturnStatus Notification Message

Returns back status variables in response to a **GetStatus** command. These status variables are common to all configuration spaces. This command can be called on any created link handle, and **ReturnStatus** should return valid data for a link even when the link has been deactivated. Using these fields for returning status on things other than what they are specified for is prohibited.

Notification Fields:

- An integer **LinkHandle** specifying the link handle for whose configuration space the user requested status.
- A timestamp for the last **RXActivity –** set to **CreateHandle** time if never updated.
- A timestamp for the last **TXActivity –** set to **CreateHandle** time if never updated.
- An integer **RxFrameCount**. This can be -1 if does not make sense in context.
- An integer **TxFrameCount**. This can be -1 if does not make sense in context.
- An integer **RxBitCount**. This can be -1 if does not make sense in context
- An integer **TxBitCount**. This can be -1 if does not make sense in context
- An integer **RxFrameDrops**. This can be -1 if does not make sense in context, or drops are always undetectable.
- An integer **RxDetectedBitErrors**. This can be -1 if does not make sense in context, or single bit errors are never detectable.
- An integer **TxFrameAcks** specifying how many transmitted frames have been acknowledged. This can be -1 if it does not make sense in context, or there is no acknowledgement mechanism.
- An integer **TxRetransmits** specifying how many times transmitted frames have needed to be retransmitted. This can be -1 if it does not make sense in context, or there is no retransmit mechanism.
- An integer **TxConfirmedBitCount** specifying how many bits have been confirmed by acknowledgements as having been sent.

XIII.D. GetTelemetry Command Message

This command is used to request telemetry information specific to a configuration space. This command can be called on any created (but not destroyed) link handle, even ones for whom a **LinkTerminated** notification has been received signifying deactivation.

Parameters:

- An integer **LinkHandle** specifying the link handle for whose configuration space the user is requesting status.

Responses:

The API responds with either a **ReturnTelemetry** or **Error** notification message.

XIII.E. ReturnTelemetry Notification Message

Returns back an buffer of bytes specific to a particular type of configuration space in response to a GetTelemetry message. If there is no telemetry to report, a zero length return size should be sent.

Notification Fields:

- An integer **LinkHandle** specifying the link handle for whose configuration space the user requested status.
- A buffer of bytes called **TelemetryBuffer** and its size as **TelemetrySize.**

XIV. Error Handling

The **Success** and **Error** notification commands are referenced several places in this document. The Success notification is used when a command requires feedback that it has successfully completed, and there is not already a mechanism in place. **Error** is used to signal an error has occurred. Both return a LinkHandle to associate them with a configuration space.

XIV.A. Success Notification Message

This notification is sent to notify of the successful completion of a wide range of command messages. It is never sent spontaneously, or when the command message responds with some other type of notification indicating success.

Notification Fields:

- An integer **LinkHandle** specifying the link handle for whose configuration space the user is requesting status.

XIV.B. Error Notification Message

This notification is sent to notify of the unsuccessful completion of any command message. It can also be sent spontaneously. Note that more extensive documentation regarding appropriate sending of standardized error messages will be addressed in the complete specification.

Notification Fields:

- An integer **LinkHandle** specifying the link handle for whose configuration space the error is to be sent to.
- A boolean **SpontaneousFlag**. This will be false if error is in response to a command message.
- An integer **ProblemLinkHandle** which is usually the same as **LinkHandle,** but can alternatively indicate that some context higher up the configuration hierarchy has had an error that affects its descendants. The **ProblemLinkHandle** must be the same as **LinkHandle**, or **LinkHandle** must be a descendant in the configuration hierarchy from **ProblemLinkHandle**. Note that when such an error occurs, every descendant in the configuration hierarchy must be called, and children configuration spaces in the hierarchy are notified before parent configuration spaces.
- An enum **ErrorResponse** specifying a recommended handling method for the error and/or nature of the error. Possible values are:
    - ignore – best course of action is probably to ignore this warning.
    - retry – best course of action is to retry last action before error.
    - restart_link – best course of action is to teardown and restart the link that caused the error if possible and it's children.

    - close_link – shutdown the link that caused error and its children
    - reset – reset device entirely.
- An enum **ErrorType** specifying the type of error:
    - bad_choice – A bad parameter has been sent that conceivably could make sense in another configuration space implementation. This indicates programmer error, or that the program is probing for certain functionality that is missing. Handle appropriately.
    - bad_usage – The api has been used in a way that cannot possibly be due to functionality probing and indicates a real programmer error.
    - fault – This indicates a suspected hardware or software failure.
    - situation – This indicates that an unusual link state or situation has triggered the error.
- An unsigned integer **ErrorCode.** This specific numeric error code for the error. Error numbers below 0x40000000 are reserved for standardized and future standardized error codes. Error numbers equal to or above 0x40000000 are specific to a particular configuration space.
- A textual **ErrorMessage** and **ErrorMessageSize** describes the error.

XV. **Conclusion**

As currently defined, the API documented is not intended to be a complete and comprehensive answer to plug and play radio functionality for SPA systems. That will ultimately require standardization of configuration spaces and telemetry options. It is, however, a step toward providing a general purpose and extensible framework for future development, collecting common radio functionality into one set of API messages. If this API, or some similar API is ultimately standardized, it will prevent every vendor from coming up with a completely new SPA interface for their radio, and facilitate the process of reaching true plug and play radio integration.

XVI. **Acknowledgements**

Portions of the work reported in this paper were performed under contract and with cooperation of Air Force Research Laboratory Space Vehicles Directorate.

XVII. **References**

[1]Lyke, James. "Plug-and-play as an Enabler for Future Systems", *Proceedings of the AIAA Space Conference*, September 2010.

[2]Lanza, D., Vick, R. and Lyke, J., "The Space Plug- and- Play Avionics Common Data Dictionary - - Constructing the Language of SPA ", *Proceedings of the 2010 AIAA Infotech Conference*, 20-22 Apr 2010 Atlanta, GA.