Remote Function Select (RFS) Protocol Suite

Interface Design Description

sparton
NAVIGATION AND EXPLORATION
5612 Johnson Lake Rd • Deleon Springs, FL • 32130
800.824.0682 • 386.985.4631
www.sparton.com/ne • www.thedigitalcompass.com

130-7545-001, Rev A

REVISION HISTORY:

DATE	DESCRIPTION OF CHANGE	INITIALS
7/03/07	Original	CMN
5/2/08	Reformatted	CMN
7/23/09	Added BitArrays and Show and Format Messages to allow for generic status message handling by GUI's	CMN
10/27/09	Minor corrections, add figure numbers, table.	CMN
4/12/10	Added fixed, floating point specifiers	CMN
6/24/10	Added get_value, etc.	CMN
15SEP10	Added floating point, name modification to RFS	CMN
5/10/11	Significant rewrite to introduction, added new data type documentation, corrected some errors.	CMN
8/25/11	Included TFTP, SAPP as part of the document. Added rev 3 stuff. Documented the bit construct message. Added the documentation for 2d arrays. Document was cleaned up and re-organized.	CMNJr
1/5/12	Clarification added to SAPP bytecount calculation. Corrected Construct command byte error. Documented Get Next/Previous Value commands (10 and 11).	CMNJr.

Table of Contents

Table of Figures

1 Introduction

The purpose of this document is to describe a communications interface such that a software developer can construct a system to control and monitor an embedded device or implement the embedded device side of the protocol. The target audience for this document is software developers. This communications interface has the purpose of creating a user interface on a remote device for embedded devices that do not have, or have a limited form of, a user interface.

This document describes several stand-alone items that are used together to implement a full command and control communications interface.

The first item is the Remote Function Select (RFS) protocol itself. The RFS protocol is a transport layer independent client-server, query-response command and data mechanism. RFS does not specify a transport or link layer. RFS expects to transmit and receive entire RFS packets through a mechanism provided by a lower protocol layer. As such RFS cannot be used by itself, but must be transported through some communications “pipe”.

As the RFS protocol is designed to transport small blocks of information (i.e. configuration settings), a second protocol is defined to allow transfers of bulk data to the embedded device. Bulk data is transferred as a file. The Embedded Trivial File Transfer Protocol is defined as a way to transfer large “blocks” of data to the embedded device, for example GPS ephemeris data, or filter coefficient tables.

In addition to the RFS protocol, this document describes a packet transfer mechanism that can be used on a simple serial port (or any similar byte wide transmission scheme). This mechanism is called Sparton Asynchronous Packet Protocol (SAPP). SAPP simply provides a mechanism to transfer blocks of binary data from one point to the other. This packet protocol is self-synchronizing, provides error detection, provides a provision for acknowledgement, carries a protocol identifier and provides binary transparency.

A common implementation of RFS is the RFS protocol encapsulated in SAPP protocol that is carried over a Personal Computer (PC) serial port to a UART on some embedded device. The relationship of the packet transfer mechanism with the SAPP encapsulated RFS, File transfer and other protocols on a serial port is shown in the following diagram:

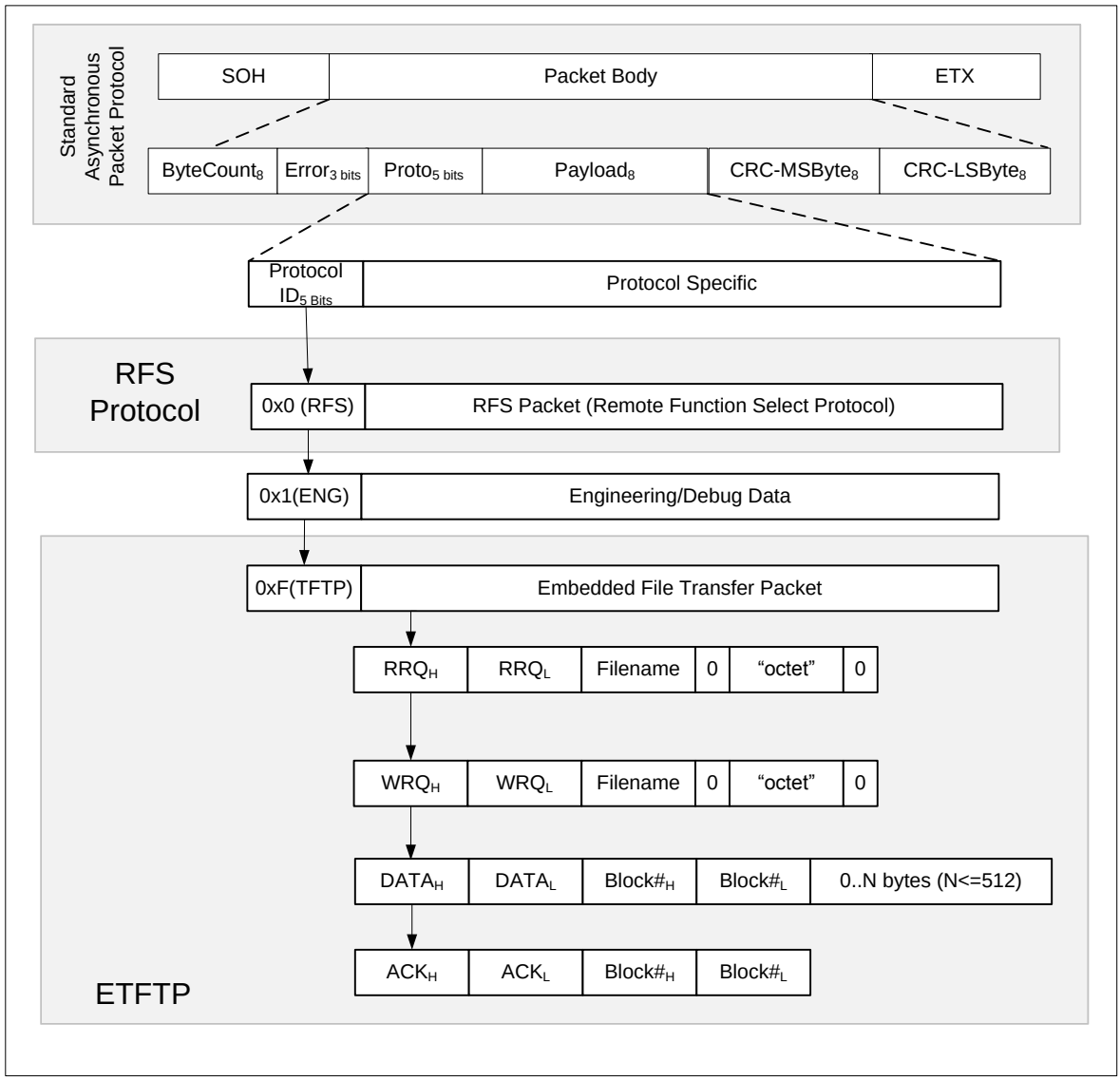

Figure 1-1 RFS Protocol Layering Relationships

The exact details of each of these protocols and layers are contained in this document. For the purposes of this introduction the details are irrelevant. However the bigger picture of using a link layer (SAPP) with a protocol designator that carries binary payloads that implement multiple protocols is the "take away" information. Readers familiar with the layered protocol model and/or IP based protocols will recognize the protocol layering being described.

Another common implementation of RFS is the RFS protocol encapsulated in SAPP which in turn is transported over an infra-red link using the IrCOMM virtual serial port. In this application the IrDA protocol stack presents a "stream of bytes" interface that from a protocol perspective behaves like a serial port. The SAPP layer uses the IrCOMM interface just as it would a "hard" serial port. The IrDA protocol stack handles all connection, data rate negotiation, error detection and retries "underneath" the RFS/SAPP protocol/link layer communications.

Another implementation is RFS carried directly in a custom radio interface. The radio interface provides the packetizing, binary transparency and error detection functions directly so the SAPP packet driver is not used.

Other implementations have included RFS via UDP/IP.

These examples are provided to orient the reader as to the steps needed to implement the RFS protocol in a new application. The implementer must decide upon a physical and transport layer method for delivering RFS packets to the RFS application, and must implement the RFS application to manage the control and configuration functions associated with the device data.

This document provides the SAPP packet level description to allow the reader to implement RFS on a serial port. Other interface descriptions are available.

The purpose of the core RFS protocol is to create a client-server architecture for configuring and monitoring embedded devices. The goal is to implement a protocol suite that allows an unaltered client to access multiple different types of embedded devices and, more importantly, for that client to be able to access different versions of the same device type, without software changes on the client side.

2 Scope

This document describes a protocol, that is, a structured method for performing some activity. In this case the protocol describes a method for configuring, controlling or monitoring an embedded device. An embedded device is one, for the purposes of this document, which does not have an intrinsic user interface. The protocol in this document allows a virtual or remote user interface to be presented to the user of the embedded device on a host platform (usually a Personal Computer). The virtual user interface communicates with the embedded device over some kind of communications channel. This document does not specify what kind of communications channel; just that it has certain characteristics. In practice the protocol has been carried over serial links, infra-red links, and various radio links. The reader should consult the appropriate section of this document for a discussion about implementation specifics.

The purpose of this document is to describe the protocol such that implementers may develop applications to control and monitor embedded devices. The intended audience for this manual is software engineers tasked with developing protocols stacks and end applications based on this protocol. This document is not targeted at hardware engineers tasked with implementing the physical communications channel that carries the packets needed for this protocol. In summary, this is a detailed technical document for software developers.

This document does not describe the parameter set for any specific embedded device. Rather, this document describes the method by which a client can query any device that implements this protocol to determine its parameter set. The reader should consult the documentation for specific devices to determine the exact parameters for a given device.

3 Overview

Remote Function Select (RFS) is a protocol suite that allows a client (user) to access control, status and data information on an embedded device. This usually takes place over some kind of half-duplex serial connection, but can take place on an IRDA serial port, a bespoke radio interface, UDP/IP/Ethernet, zigbee serial interface, or any communications channel capable of half or full duplex, bi-directional binary communications. The implicit requirements of the communications channel are that it delivers a stream of bytes from the client to the server and vice versa. The RFS protocol does not place large demands on the communications channel. The communication channel for Remote Function Select protocols must provide the following facilities:

- The channel must transmit a block of 8 bit bytes (a "packet") of approximately 256 bytes or less in either direction (device <> client). The protocol expects bytes as the basic data elements.
- The channel must present the packet in such a way as to identify the beginning of a packet and the size.
- The channel must provide binary transparency for the packet contents. For example, the channel must **not** convert non-printable characters to other forms or expand carriage returns to carriage return, line feed pairs.

RFS is a client-server protocol. The client (user) makes a request of the server (embedded device) and the server responds. The client may request that the server respond multiple (or until commanded to stop) times per request. This behavior would be used in a sensor type application.

RFS is designed to reduce the amount of software maintenance required on the client side software. Just as a web browser program can access new and varied content with minimal upgrades, the RFS functionality is similar. RFS accomplishes this goal by establishing a common set of descriptions for various kinds of data that would be found in an embedded device, and describing a message format that allows the various kinds of data to be communicated. Additionally, RFS defines messages which transmit the meta-data about each embedded device data to the host. The meta-data describes information ***about*** the data, not the data itself. This meta-data includes the name of the data, its type, its range limits, and/or the possible choices, and the names of the choices. This methodology allows the GUI application developer to implement the graphical interaction of the individual data types only once and then be able to access any and all data items in the embedded device's data base.

RFS does not specify any kind of device addressing or routing, nor any error detection or correction of packets (including retries). RFS assumes a blind transmission and reception interface. The user is responsible for implementing any transport layers, error handling and routing functions externally to the RFS packets.

The important noteworthy characteristic of RFS is that the meta-data describing the internal data contents of the embedded device is contained in, and transmitted from the embedded device itself. When new or modified embedded devices become available with dissimilar data contents the device is capable of describing these new contents to the client side device. This methodology was initially

designed for an environment where the embedded device is disposable or evolving and the client device is persistent. The methodology transfers the software maintenance task to the readily changeable device from the difficult to change persistent client device.

4 Protocol Description

4.1 Discussion

This protocol supports the ability to read ("get") or write ("set") a value in the embedded device. Values may be status, configuration or data (e.g. in a sensor type embedded device). No facility is made for the device to set or read any value on the interrogating (client) device. The embedded device always performs the server role. It will always be the server or resource in a two way communications with an interrogating device (the client). This is shown in the figure below.

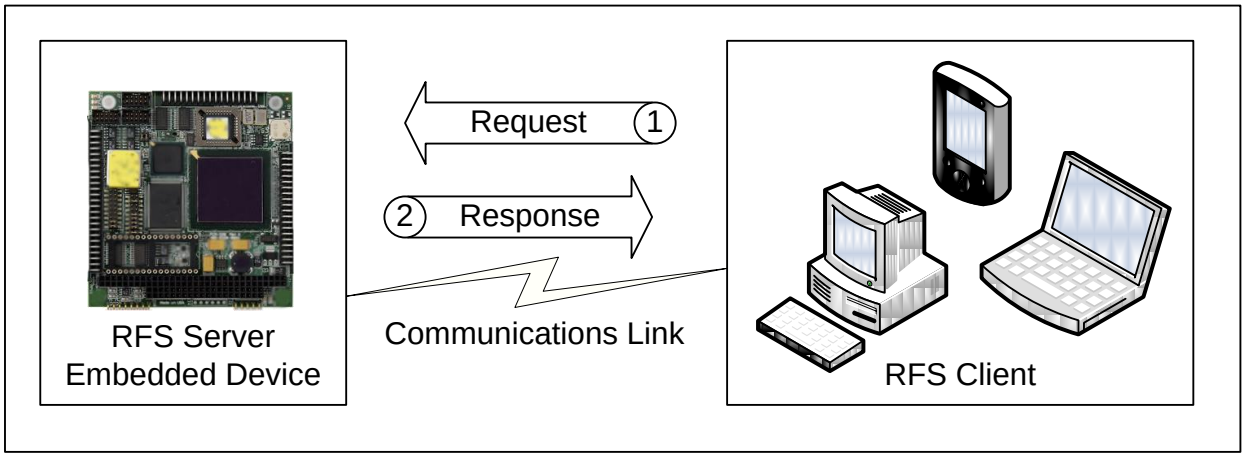

Figure 4-1 High Level Client Server Relationship

The protocol assumes an underlying physical, framing and transport layer. The underlying layers must provide packet level service from the client application to the embedded device. The transport layer must handle the connection, addressing and any error detection/correction and retries independently from the RFS protocol layer. The following transport layers have been used in the given combinations:

- Simple serial port: The underlying physical layer is an RS232 serial port that carries asynchronous binary data. The framing is handled by a packet layer that implements the Sparton Asynchronous Packet Protocol. RFS is carried as one of the protocols in the SAPP layer.
- IrDA port: The underlying physical layer is IrDA SIR serial communications. The IrDA protocol stack is implemented to handle the link layer and implements the IrCOMM virtual serial port functions. A middleware layer performs framing using the Sparton Asynchronous Packet Protocol.
- UHF Radio: The underlying physical layer is a burst mode UHF radio. The radio hardware performs the packet construction, framing and error detection. The RFS protocol is carried within the MAC layer framing implemented in the radio. The framing layer implements the

Sparton Multi-Drop Asynchronous Packet Protocol (MAPP) that implements addressing in a broadcast environment.

- Satellite Modem: The underlying physical layer is a satellite modem using the vendor proprietary RF technology. The MAPP protocol implements the addressing and framing layers. RFS is carried in the MAPP framing layer.
- UDP/IP: The RFS protocol was tunneled through UDP/IP on a standard local area network. UDP/IP was used for routing and addressing.

Some of these applications are illustrated in the following diagrams as protocol stacks. Note that in each of these applications at least one other protocol existed in parallel with the RFS protocol in the embedded device. The identification of and routing of the RFS protocol and the other protocols was handled at the "link" layer. This illustrates that RFS is a standalone protocol that can exist within a protocol "network" and is able to be independently described in isolation from the surrounding protocol layers.

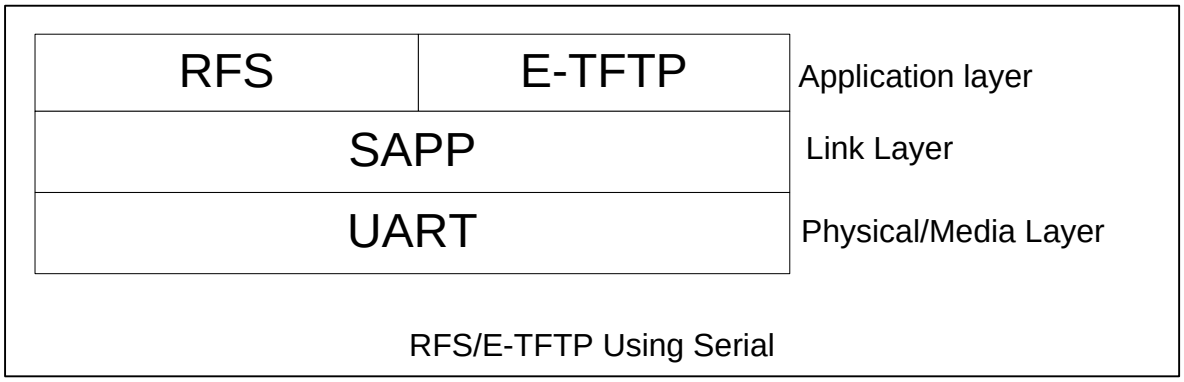

Figure 4-2 RFS Protocol carried over UART/Serial port.

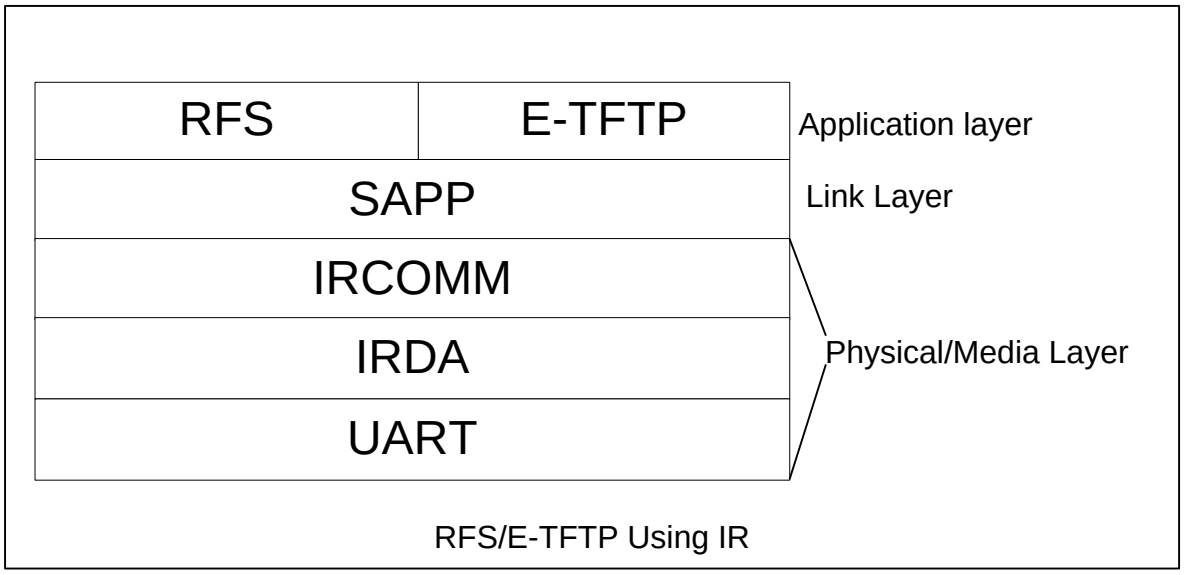

Figure 4-3 RFS Protocol carried over Infrared Port (IRCOMM/IrDA)

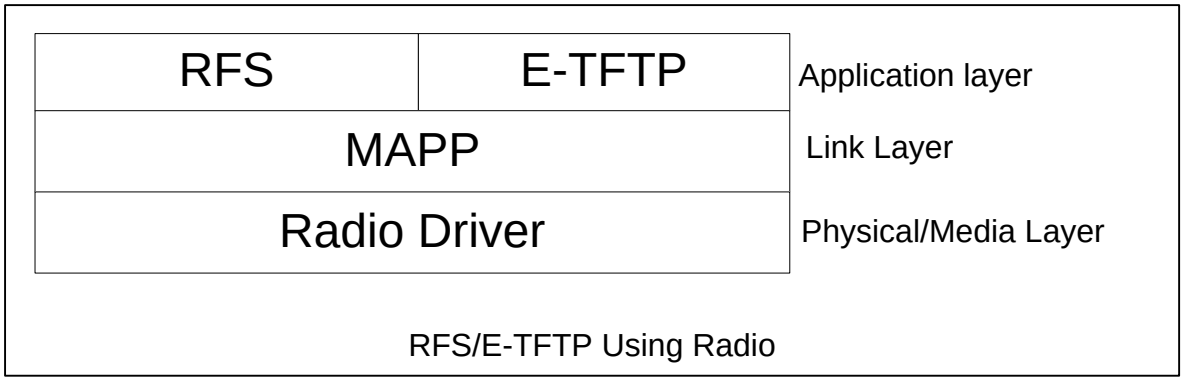

Figure 4-4 RFS Carried over bespoke radio interface.

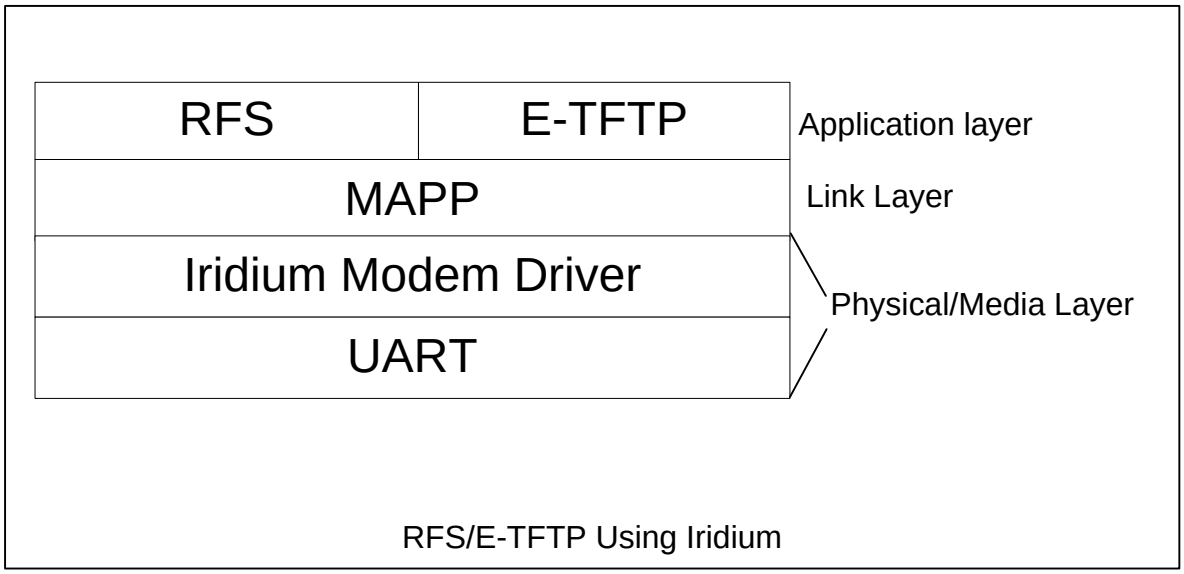

Figure 4-5 RFS Carried over Iridium Satellite Modem.

Note that in some cases the link layer is different in response to the network requirements of the application. This document contains appendices that describe some of the link layers. The ordinary serial port is described in detail in the section describing the Sparton Asynchronous Packet Protocol link layer.

4.2 Protocol Overview

4.2.1 Basic Operation

At the core level this protocol is a simple request and response protocol. The embedded device (the server) generally waits for requests from the external device (the client). The client can make one of several kinds of requests. The most common request is a “get” or “get value” request. The “get” request informs the server that the client would like to obtain information about one of the variables in the server. The variables are identified by a Variable IDentifier (VID). The VID’s are numerically increasing values that uniquely identify each item in the server. Two of the VIDs are reserved and always contain two data items needed by the host to interact with the server. VID 0 is always the count

of the number of VIDs in the device. VID 1 is always the name of the device. The number VID's and the device type carried in the name of the device allow the host to display a window title for the user and configure the client device for operating with the embedded device.

The "get" request carries minimal information other than the packet framing. The essential information is the "get" request command code itself and the VID of the requested item. The server's response to the "get" request is some form of VID value information depending on the "get" type issued by the client. Some responses contain the variable value and meta-data about the variable. Some responses contain only the values. The client has the option to request data or data plus meta-data by virtue of the type of get message that is transmitted.

4.2.2 Memory-less Operation

The RFS protocol is non-temporal. Message ordering is not required nor expected. No RFS message depends on the prior transmission of some other message. Each communication is a single request with a single response. Failure to receive a request or response does not constitute a failure of the protocol and may occur in noisy environments. Client applications should employ timeouts and retries as appropriate for the communications environment and intended reliability. Note that link/physical layer drivers may drop a message due to link errors such that the RFS protocol stack never receives the packet. Link layer software should detect and recover from this error per the system requirements of the system being designed.

4.2.3 Use Case

The remainder of this document contains the detailed description of the data types and message types needed to conform to the protocol. As with most protocol documents the amount of detail needed to resolve ambiguity generally creates something that is difficult to read. This section is designed to provide a high level "feel" for how the protocol is designed to be used. With a high level understanding of the general usage, the implementer may dig into the actual details of the protocol needed to complete an implementation. This section is intended to be informal and lacking in detail so that the reader may get the "big picture" of the use of the protocol.

This protocol was designed to solve a certain problem as illustrated in this figure.

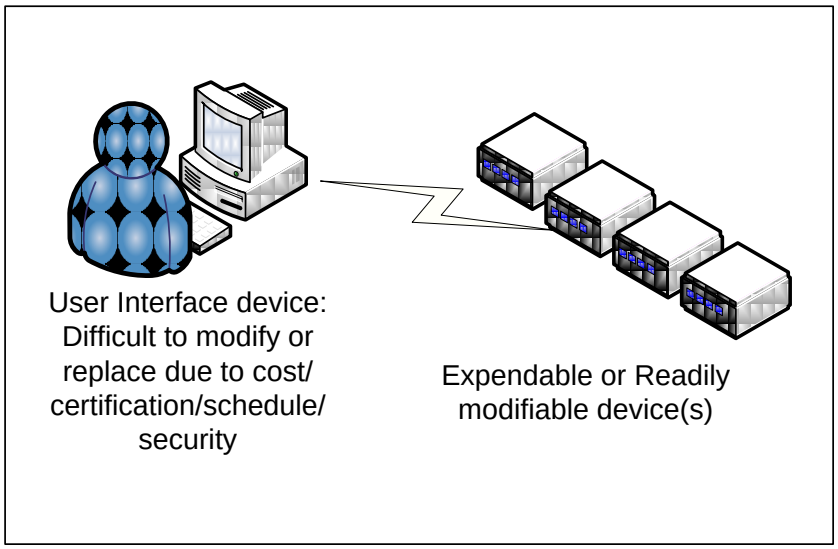

Figure 4-6 RFS Problem/Solution Domain

The problem domain is the situation where some kind of device (typically an embedded device that has limited user interface capability) requires configuration or monitoring by a human being. The historical paradigm for solving this problem was to create a very rudimentary, bespoke, command sequence for the embedded device. The human interface would then be tailored to send the bespoke commands to the embedded device for configuration. This solution is entirely viable in a relatively static domain where the functional needs of the communications and configuration do not require changes in either the human interface or the embedded device. However, if the requirements change, the embedded device could be readily changed with the next generation device. However since the user interface device cannot be easily changed for whatever reason, the system is essentially frozen with respect to new features. The embedded device may implement the new feature but then there would be no way to take advantage of the new features on the user interface side.

RFS was designed to overcome this restriction by expanding the custom interface into a generic one. As a further expansion, the characteristic knowledge of the embedded device's configuration would be stored in the embedded device, not the user interface device. This is modeled after the paradigm of the World Wide Web and an Internet Browser. In the WWW/Browser scenario, the browser does not maintain information about every web site; rather, the web sites' transmit information in a feature rich language (HTTP, et. al.) to the web browser. The browser simply presents the information to the user, regardless of the content.

RFS is in essence a very simplified HTTP-like language to describe and select configuration variables in an embedded device. Three different paradigms were envisioned for RFS. Paradigm one is that of a simple hand held device like a PDA. The PDA would contain a simple user interface program that processes RFS messages. The user would connect the PDA to the embedded device and scroll through the data items in the device one at a time. The PDA software would request one data item, present it to the user, then with a button press move to the next (or previous) data item. The PDA would not require any permanent memory for settings or configuration as that information would be contained in the

embedded device. In this paradigm the PDA (or client device) would simply be a "dumb" display device that removes the need to have a display and keyboard on the embedded device. The protocol supports this operational paradigm in that individual variables may be retrieved with not only the current value, but the name and other characteristic "meta-data" about the variable. This is enough data to display the variable and all of its choices on the simple PDA type device. The message pair that supports this paradigm is the Get and Get_Response message pair.

The second paradigm for which RFS is targeted is the set of client devices with more permanent memory that would be needed for the PDA implementation. (Although there is nothing to prevent this paradigm from being executed on the current range of PDA, IPAD, etc. like devices available.) In this situation the client device might first query all the data in an embedded device and construct a multi-valued display for the user. This application might include a priori assigned names for commonly expected variable types within a certain problem domain. The user would be thus presented with a more attractive or informative display from which to select configuration. Noteworthy in this paradigm is that the individual ranges and types of values would still be stored in the embedded device and transmitted to the client device on demand. In practice this means that if a new embedded device or an improved embedded device becomes available, the new features/choices are automatically presented to the user without changes to the client user interface software.

The third paradigm for RFS is similar to the second but applies to a bandwidth limited environment. In this case the client would query the embedded device to determine which make/model/revision is being queried. The client would then use this information to open a local description file (supplied in XML[1]). The local description file would contain the same database contained in the embedded device for the version being addressed. The client device would use the descriptions to create the user display and send the minimal commands to the device to retrieve and set the values needed. The Get_Value, Value_Is and Set commands are directed towards this paradigm. Note that in this paradigm, the client device automatically adapts to the new devices as the new device behavior is presented to the client device in the same "language" that is used to communicate with the device, the difference is that the entire device description is contained in a file supplied to the client device rather than by the client querying the device. Thus once the client device has been constructed to display and interpret the basic data types whether by file or query, any changes are automatically incorporated for the new device.

4.2.4 Vocabulary

This section describes some of the terms used to describe the protocol.

- Basic Data Type – A description of the binary format of simple data or meta-data. Basic data types describe the way integers, strings and other types of information are formatted for transmission.
- Integer – A whole quantity. Integers represent values that may vary in value across a great range.

[1] An example XML file describing the RFS database for a typical device is included as an appendix.

- Ordinal – A value that selects one of a small number. An ordinal is a selection amongst a small number of discrete choices. An ordinal would be used to describe the settings of a selector switch with, for example, the choices of high, medium and low. This is in contrast to an integer that represents actual temperature. In RFS each selection of an ordinal has a unique name associated with it. Integer values do not have a unique name for each differing value.
- String – A sequence of human readable characters in ASCII form plus the necessary framing and carriage control. A string is intended to be displayed for human readers.
- Variable – A piece of data that describes the state of a particular item. A variable describes one "thing". The state of the variable may be a single byte, a 32 bit integer, an array, a string, or a group of quantities. A single variable is the "subject" of all RFS messages. RFS manipulates one variable at a time.
- Variable Identifier – A variable identifier is a number. It uniquely identifies one variable in the embedded device. All variables are identified by the client and embedded device by the Variable Identifier.
- Data – The value of a variable.
- Meta-Data – Information about a variable, not including its value. Meta-data is the extra information about a variable that allows the client to display the variable to the user. The name of a variable is meta-data as are the range limits. Meta-data is described using basic data types.
- Message – A block of bytes that implements one RFS command or response.
- Object – A conglomeration of various bits of data and meta-data that describes a variable.
- Field – One of the components of an object or message. A field describes some subset of an object or message that can be identified as a "thing" itself; the name of a variable for example, is an identifiable "thing" that might be part of an object.
- Element – One of the primary components of a field or basic data type. For example, the string length byte is an element of a string field.
- Description – A description is information that describes meta-data. A description object contains the information to describe the meta-data for a variable. Description objects contain description fields. Description fields are composed of combinations of basic data types.
- Value – The current or desired setting for a variable, i.e. the actual information. Values are transmitted in value objects. Value objects contain data fields that are comprised of elements that are in turn comprised of basic data types.

4.3 Meta-Data

Intrinsic to the RFS protocol is the concept of meta-data. A common message exchange is the Get/Get_Response message pair. The message by the client is to "Get" the information about a variable identified by a VID. (Note that VID-0 always contains the total number of VIDs in a device, such that the client device can "walk through" all known VIDS in a device.) The "Get" message is a general query that in essence requests all information about a variable. The "Get_Response" message provides all the information that the embedded device can supply about the given variable. The exact details and

format of the message for various data types will be described later in this document; however for example, a simple 32 bit integer type will be used to describe data and meta-data about a variable. When the embedded device returns the "Get_Response" message, the message contains sufficient information to allow the client to display the value to a human user, and allow the human user to correctly select a new value for the variable. The meta-data for a simple 32 bit integer variable includes the name, in human readable text, of the variable, the basic data type of the variable (32 bit integer), the plurality of the variable, i.e. is it a single scalar, or an array, in this case a scalar (single), the range limits of the value, the default value and the current value. The meta-data is sufficient for a client device to display the variable to a human operator and allow an informed selection of value. Once the operator has made a selection, the client device can send only the new data value back to the device.

The meta-data can be retrieved one at a time for some classes of client devices, for example handheld devices that contain only a single line display, or can be retrieved in total for more feature rich client devices. Some more capable client devices may store the embedded devices' database settings for a particular configuration for automated transmission to the embedded device without human intervention. The protocol does not require the "get" before "set" behavior, thus the client device can uni-directionally send a group of "set" commands to the device in cases where the device is a well-known type.

4.4 Transactions

Two kinds of message transactions are allowed. The first kind of transaction is a poll-response exchange. The client emits a message request and the server responds with one or more response messages. A bit in the protocol header indicates whether or not the current message is contained in one transmission unit or continues into the next transmission unit. The second category is an unsolicited message from the server. An unsolicited message may be some kind of error condition ("trap") or more likely a repeating sensor output value. The unsolicited messages are controlled by the client by enabling or disabling settings that allow/disallow the unsolicited responses.

4.5 Variable Identifiers

Each unique data item (variable) in the embedded devices is identified by a unique number. This number is called the Variable IDentifier (VID). Each VID may be "gotten" or read. Some variables may be "set" or written. The server is responsible for disallowing any illegal values being set. The client must "re-get" a variable to determine if the value set was proper and accepted. The server will respond with an error message if an illegal VID is sent or the value of a set command is illegal. The error message will contain a text message indicating the nature of the error.

4.6 Variables

- Variables may be read/write or read only.
- A read/write variable is one in which the value may be read with a "get" command and may be written with a "set" command.

- A read only variable is one in which the device provides status or data. An attempt to set a read only variable is ignored.
- Depending on the device, some variables may be permanent (i.e. the value is maintained through a power cycle or reset) or temporary. This behavior is device dependent.
- Flags contained in the protocol indicate whether the variable is read only and/or persistent.

4.6.1 Basic Data Types

The protocol defines basic data types. These values are used to communicate VID values and VID meta-data. Meta-data describes information about the VID other than the value of the VID. For example, meta-data may include the name of the variable. Variables may be scalars, meaning a single value per VID or complex types. A complex type may be an array of basic types or an aggregation of dissimilar primitive types. **Note: All data types are defined in a CPU (endianness) independent manner. All data types are treated as arrays of bytes when transmitted. There is no guarantee that multi-word types will fall on native multi-word boundaries. The protocol assumes no alignment and no packing. Implementers are cautioned to use *machine independent* methods of inserting and extracting data and converting to and from native values. Do not assume that a "struct" can be typecast onto a message with predictable results.** The protocol only guarantees that the specified bytes in a message will be arranged in the order described. The basic data types are as follows:

- Int32, a 32 bit integer transmitted most significant byte first (big Endian). Int32's range from -2^32 to +2^31 (0x80000000 ... 0x7FFFFFFF).
- Ordinal, an 8 bit type, where each unique value has an associated string describing the meaning of this selection. Ordinals vary from 0 to the maximum possible value for the variable. An ordinal that allows the user to select from three choices will have the three values of 0, 1 and 2.
- String, composed of a length byte, followed by up to 200 bytes (including embedded 0's) and a final terminating null (0) (even if there is an embedded 0 immediately preceding). [2]
- Fixed Point Integers (Fixed32 or Fixed64). 32/64 bit values that represent fixed point values. The number of fractional bits is described in the description of the variable.
- Floating point single precision (Float32). 32 bit values that represent floating point values is IEEE standard format.
- Double Precision floating point (Float64). 64 bit values that represent double precision floating point in IEEE standard format.

The sections that follow describe the binary transmission format of the basic data types. The bytes are presented left to right in transmission order. The choice of most significant or least significant bit transmission for individual bytes is a hardware dependent choice of the system designer. The protocol only assumes byte ordering.

[2] Note that this does not match the common representaion of "C" style strings. The length byte is incorporated to aid in processing variable length data fields in a message.

4.6.1.1 Byte

A byte is a single 8 bit unsigned quantity. Bit transmission order is transport layer dependent. A byte is the smallest sized object that the RFS protocol expects to use in the transport layer.

4.6.1.2 Int32

An Int32 represents a 32 bit signed value. It is formatted/transmitted as follows:

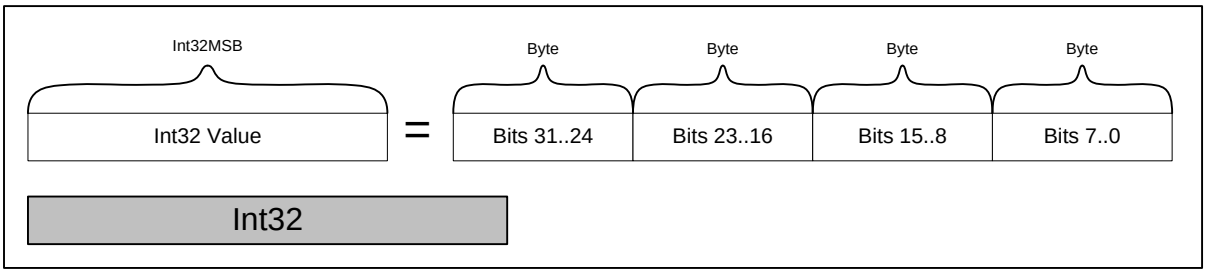

Figure 4-7 Int32 Data Format

For example the hexadecimal value 0x12345678 is transmitted as 0x12,0x34,0x56,0x78.

4.6.1.3 Ordinal Data

Ordinal data is transmitted as a single 8 bit unsigned byte. Since the value represents the maximum value selectable the range of Ordinal data is therefore 0...255. An ordinal value indicates a quantity that is a selection from a finite set of items. Each finite selection is represented by a string name that is transmitted as part of some messages as meta-data. However the actual data itself is a single byte quantity.

4.6.1.4 String Data

String data is formatted as shown in the diagram below.

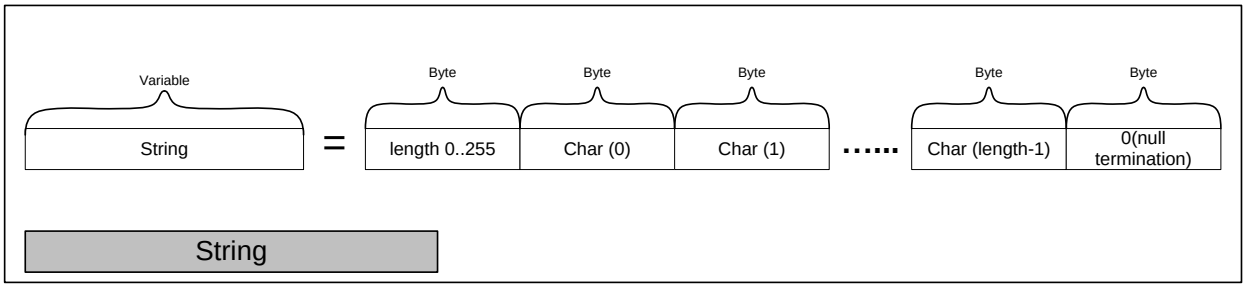

Figure 4-8 String Data Format

A string is an array of bytes. Byte 0 is always the length of the string and describes the number of bytes in the string **including the null termination** (Note : not the same as "C"'s strlen() function returns.) but does not include the length byte in computing the length. Thus the longest string possible is 254 characters including the trailing null (Since the maximum field size is 255). Strings may have embedded nulls including the last Byte before the terminating null.

For example the string “Hello World” is transmitted as
0xC,0x48,0x65,0x6c,0x6c,0x6f,0x20,0x57,0x6f,0x72,0x6c,0x64,0x00.

4.6.1.5 Fixed32

A fixed point value contains a whole value and a binary fraction contained in a 32 bit value. The format is SWWWW.FFFFF, where S represents the sign bit, W represents a binary digit of the whole part of the value and F represents a single bit of the fractional part of the value. Notationally this is indicated by X.Y where X represents 1(the sign bit) plus the number of whole bits and Y represents the number of fractional bits.). The indication of the partitioning of a fixed32 value into X and Y parts is part of the meta-data for a given variable. The range of the whole portion of a fixed point value is $-2^{(x)}$ … $+2^{(x-1)}$. For example if x =4, then the range of the whole portion is -8 to +7. A Fixed32 is transmitted as follows:

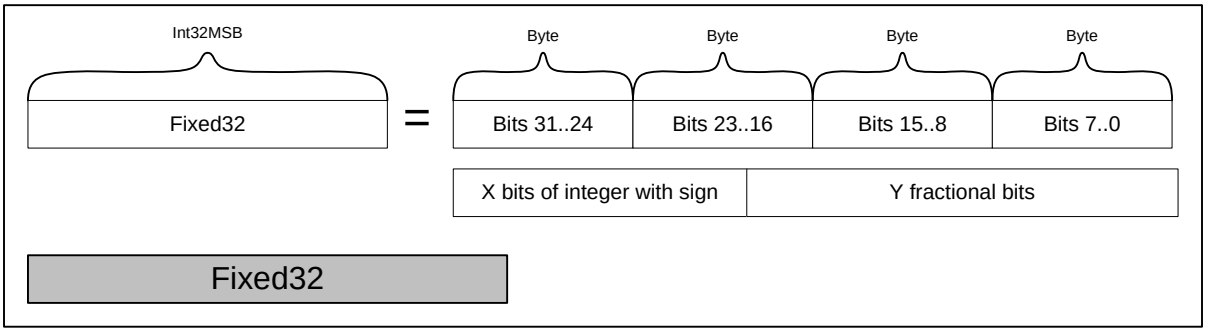

Figure 4-9 Fixed32 Data Format

For example the 9.23 fixed point value that might represent a latitude value of 29.3815 degrees is transmitted as (246357897 (0x0EAF1F89) is the integer value) 0x0E, 0xAF, 0x1F, 0x89. Similarly -29.3815 degrees (-246357897 (0xF150E077)) is transmitted as 0xF1, 0x50, 0xE0, 0x77.

4.6.1.6 Fixed64

A fixed point value contains a whole value and a binary fraction contained in a 64 bit value. The format is SWWWW.FFFFF, where S represents the sign bit, W represents a binary digit of the whole part of the value and F represents a single bit of the fractional part of the value. Notationally this is indicated by X.Y where X represents 1(the sign bit) plus the number of whole bits and Y represents the number of fractional bits.). It is transmitted as follows:

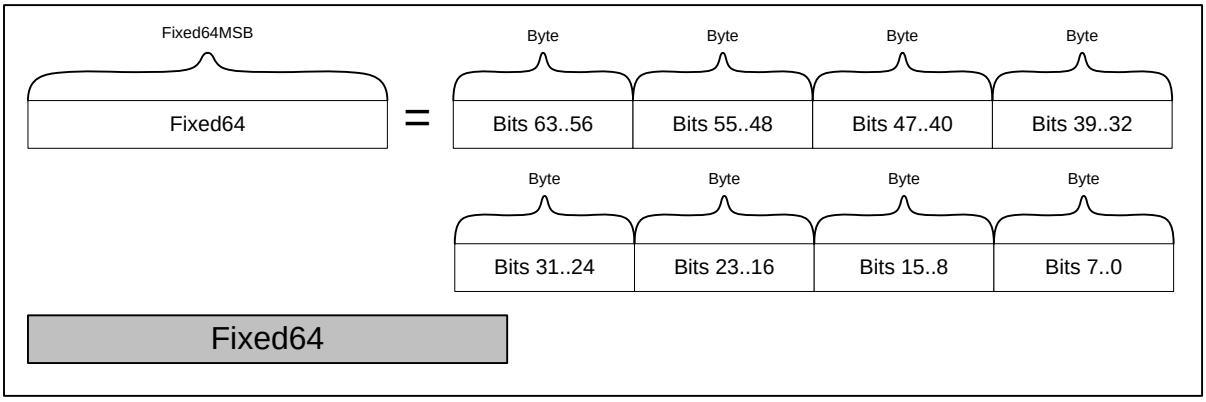

Figure 4-10 Fixed64 Data Format

4.6.1.7 Float32

A Float32 represents a floating point value. The format is IEEE 32 bit binary format. It is transmitted as follows:

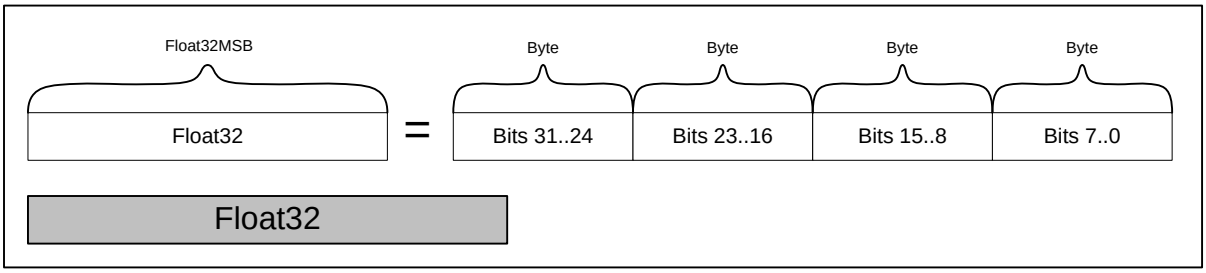

Figure 4-11 Float32 Data Format

4.6.1.8 Float64 Data

A Float64 represents a double floating point value. The format is IEEE 64 bit binary format. It is transmitted as follows:

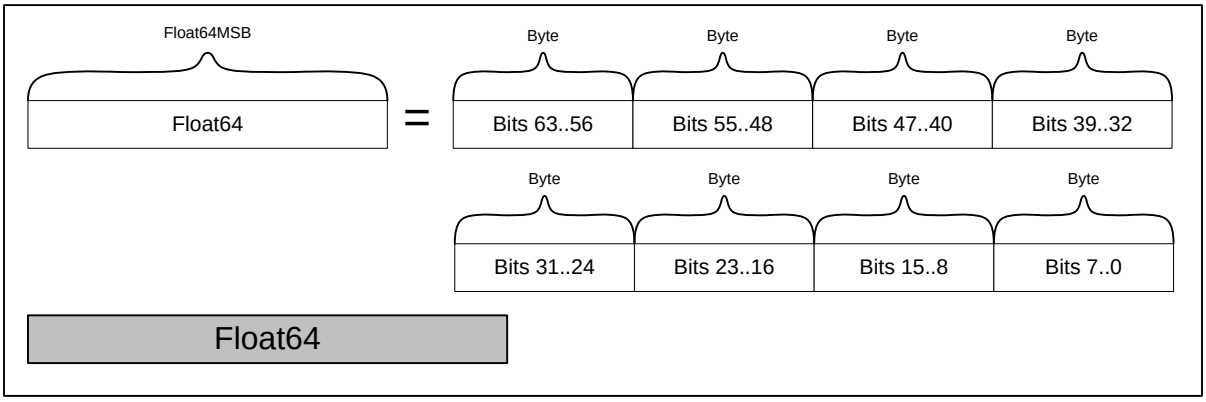

Figure 4-12 Float64 Data Format

4.6.1.9 BitArray Descriptor

The BitArrayDescriptor is a meta-data description item. It is used by BitArrays (described in more detail later in the document) to describe the contents of one field in a BitArray variable. BitArrayDescriptors are 32 bit quantities, sent most significant byte first. The BitArrayDescriptor value is formatted as shown in the diagram below.

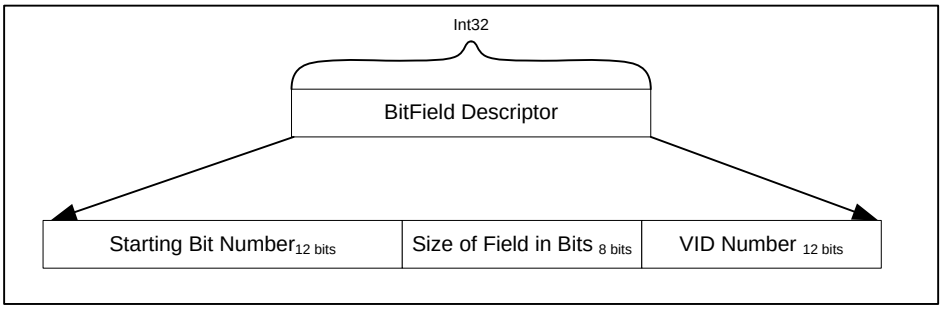

Figure 4-13 BitArray Descriptor Format

The upper 12 bits form the starting bit number in a BitArray, the next 8 bits designate how many bits are used for the quantity, and the lower 12 bits describe which VID is to be packed into this field.

4.6.1.10 BitArray

Bit arrays are simply that, arrays of bits. A BitArray is transmitted as a sequence of bytes. The most significant bit of the first byte in the sequence is numbered bit 0. The second most significant bit of the first byte is numbered bit 1, and so on. The least significant bit of the first byte is numbered bit 7. The most significant bit of the second byte is numbered bit 8 and so on. Bit arrays are used to compact multiple values into a single byte array with minimum wasted bits. The bit numbering for a BitArray is shown diagrammatically below. The length of BitArrays is always rounded up to the next 32 bit boundary for transmission.

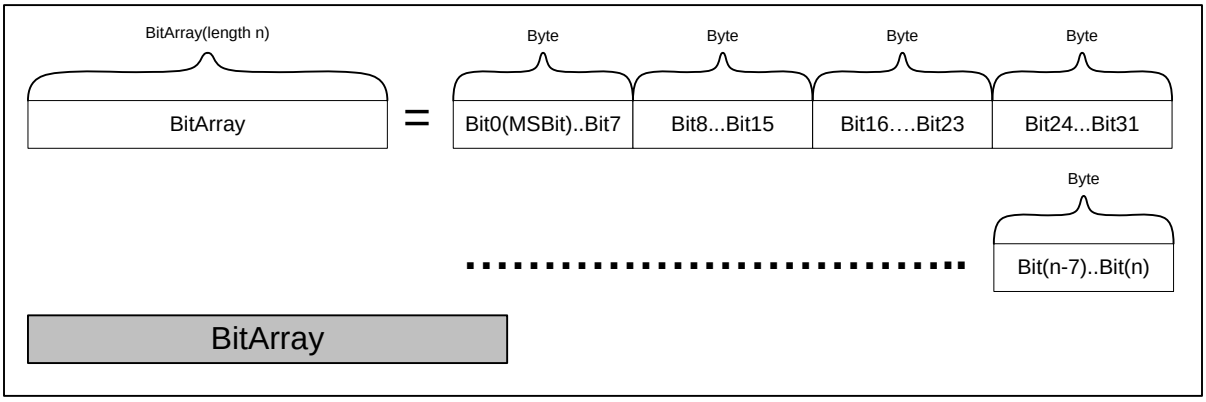

Figure 4-14 BitArray Format

4.6.2 Complex Data Types

Data may be combined together in a complex type and associated with a single VID. An array is a set of basic data type values, stored and transmitted sequentially in big endian order. A BitArray is a composite object formed by concatenating various scalar numeric basic data types into a combined object.

4.6.2.1 Arrays

Arrays are aggregations of a same data type. Arrays are used to carry vectors or rows of matrices. The array type consists of multiple items of the same base type. Arrays will be described in detail later in this document.

4.6.2.2 BitArrays

A packed data type called a BitArray is an aggregate data type. A BitArray does not convey any specific information associated with the VID itself; rather the BitArray contains values from other VIDS packed into an efficient transmission format. As the name implies a BitArray variable allocates groups of bits for other vid items and may use only the required number of bits to convey the required information. In one sense a BitArray is a “meta-vid”. For example, consider two VID values that are of type Int32. These two values may only have limited maximum and minimum values such that the actual range of values may be represented in a few bits, say 7 bits each. To read the two values without a BitArray, two separate transactions, each carrying a full 32 bits is required. The BitArray allows the aggregation of these two values into a single entity where only the required number of bits is used to represent each value. The two 7 bit values would be packed into a single 32 bit word. The BitArray thus allows the transmission of multiple values in an efficient manner. The protocol provides a mechanism for the client to define the format and contents of the BitArray variables, thus providing the client with the ability to construct a single message carrying all necessary variable values.

The exact format and construction of BitArrays will be described in detail later in this document.

4.7 Protocol Description

Each RFS message is one of the messages described in the figure below. If a message cannot fit into one transmission unit from client to server or server to client it will be segmented into 2 or more messages. Each message shall contain the same header information as to protocol, command and the same sequence number. All messages shall have the high order bit of the command byte set to 1 (i.e. OR'd with 0x80) excepting the last message where the high order bit shall be 0. A message that fits within the transmission unit shall have the high order bit of the command byte set to 0. The "breaks" in the payload field are not required to align with any field boundary. The receiver of a multi-packet message shall assemble the payload field in its entirety before attempting to parse any and all subfields contained therein. Under no circumstances shall any message body exceed 512 bytes. This restriction is provided to allow this protocol to operate on small memory footprint microcontrollers with limited RAM. The underlying transport layer must provide a minimum packet size of 8 bytes (for devices that have less than 128 VIDs) as the protocol header must be transmitted for each multi-packet message.

Messages are comprise of several parts.

- Header: The revision level and payload size are part of each message. This is part of the basic header and provides a basic sanity check on the protocol level and the size of the message.
- Command: The command indicates the desired action.
- Sequence Number: The sequence number is used to uniquely identify each RFS message or aggregate multi-transmission-unit messages. It is considered part of the header.
- VID: The VID field is variable length and identifies which variable that this message applies to. Note: The VID is not present for some messages, for example Get_Next.
- Data/Meta-data: This field contains the data and/or meta-data associated with this command. Note this field is not present in some messages.

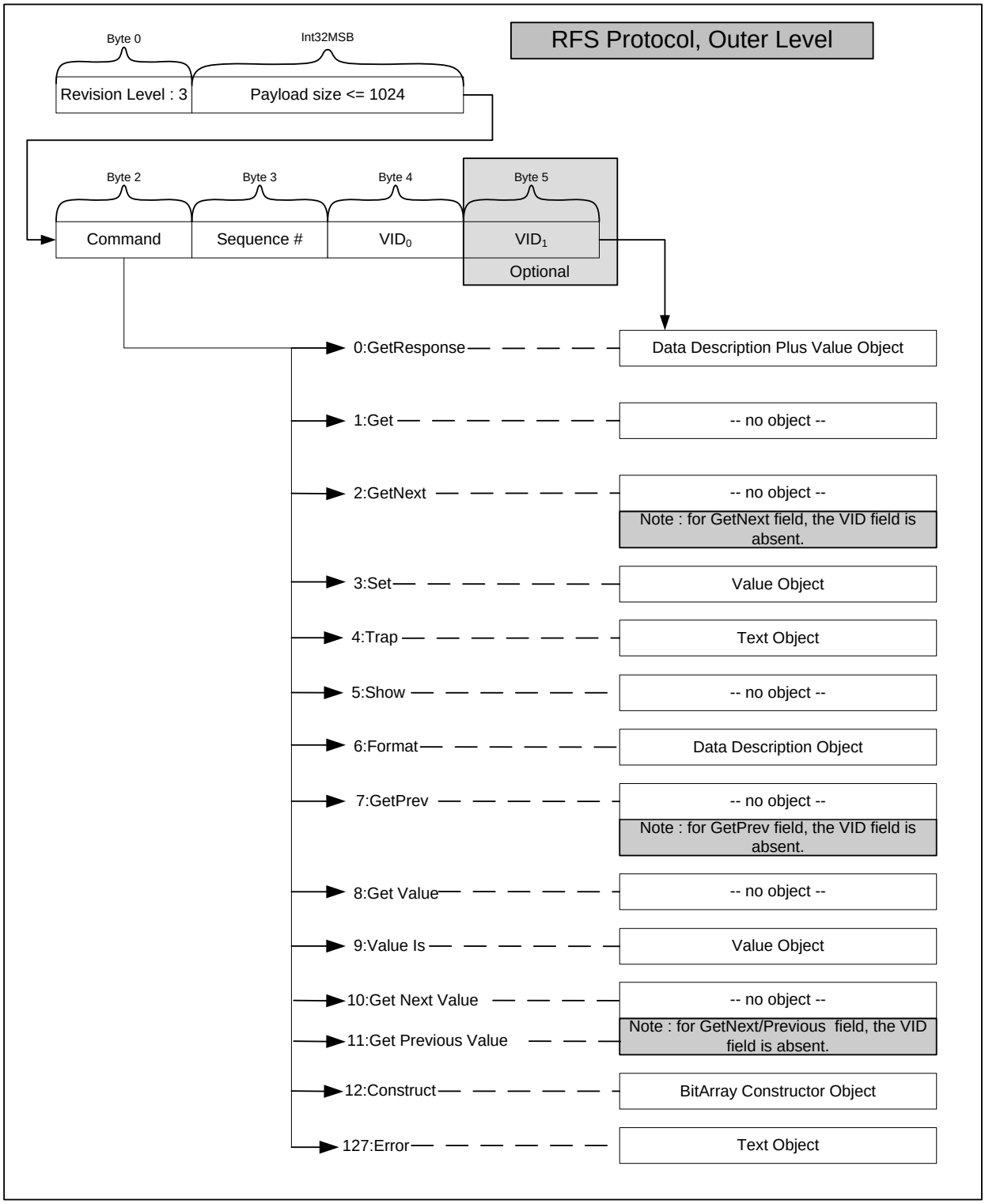

Figure 4-15 RFS Outer Layer Protocol Diagram

4.7.1 Message Header

Messages contain individual elements, roughly grouped as described in the previous discussion. This section describes the individual message header elements. Refer to the diagram in the previous section to locate the placement of each element in the message.

4.7.1.1 Protocol Version Number

The version number is the first byte of the RFS protocol and describes which version of this standard that the protocol adheres to. This document describes revision 3 of the RFS protocol. Other versions are reserved and may be used in the future to add additional capabilities to the RFS protocol. Any device and or client shall respond to version 0 of the protocol according to this document. Version 0 includes the basic data types of Int32, Ordinal and String. These values are sufficient to access the reserved VIDs and to identify the device. If a new device or client is developed and encounters a RFS device with a higher protocol version level than itself both sides shall default to the earliest version level between the two devices. For example if a device responds with version 2 of the protocol and the client responds with version 1 of the protocol then the device shall respond only to version 1 commands. The revision level in which commands or data types was introduced (and therefore permitted) is contained in parenthesis by each command and is also described in a table in the appendix. Note that devices that utilize RFS features beyond version 0 may have reduced controllability from clients that implement only RFS version 0 features.

4.7.1.2 Payload Size Field

Following the version number is a single Int32 value indicating the total size of the payload field. This value does not include the size of the header information or the count itself or the command, sequence number, VID. It does cover the message body whether it is in a single transmission unit or multiple transmission units. The purpose of this field is to verify that a complete message has been received when it was transmitted across multiple transmission units. In practice most messages fit within a single transmission unit.

4.7.1.3 Command Byte

Following the sequence field shall be the 1 byte command. If the command fits within one transmission unit for the communications interface the high order bit position will be 0. If the high order bit is a 1 another transmission unit (packet) will follow. For each packet the header, up to and including the VID will be repeated, followed by the message body parts. For multiple packet messages the payload counter value shall be the total bytes remaining to transmit, including this packet. So for a single packet message the count of payload bytes shall equal the count field. For a multi-packet message the first count shall be the entire message payload size, the second counter shall be the size of the payload minus the bytes sent in the first packet and so on until the last packet that will have a count equaling the number of bytes in the payload. This packet shall have the high order bit of the command byte cleared to 0. In this way the correct number of bytes in the payload shall be cross checked by the receiver.

4.7.1.3.1 Command Table

The table below lists the available commands. The commands are marked as client sourced, server (device) sourced, and the corresponding message type that is matched with the given command.

Command	Source	Command Pairing
Get	client	Get_Response
Get_Value	client	Value_Is
Set	client	none
Get_Next	client	Get_Response
Get_Previous	client	Get_Response
Trap	server	none
Show	client	Format
Construct	client	none
Error	server	none

The following commands are defined:

4.7.1.3.2 Get_Response (RFS0):

Command Byte Value: 0

This command is used to respond to the client with the description of a VID that was previously requested by a Get or Get_Next message. In the case of a Get message the VID of the requested variable must have been sent by the client requesting this variable. In the case of the Get_Next command the Get_Response is the next VID in the devices database. In either case the Get_Response message contains the description fields of the variable. The sequence number of the message shall be the same sequence number as the originating Get or Get_Next command.

4.7.1.3.3 Get (RFS0):

Command Byte Value: 1

This command causes the device to respond with the Get_Response message for the VID that is included in this message.

4.7.1.3.4 Get_Next (RFS0):

Command Byte Value: 2

This command causes to the device to respond with the description fields of the next variable in the database after the most previous Get or Get_Next command. The device shall automatically wrap the index associated with Get_Next in a circular fashion. If a Get_Next command is issued without a previous Get command the 0th variable shall be returned as the starting point. If a Get command is issued then the VID of the variable “gotten” shall form the basis for the Get_Next command.

4.7.1.3.5 Set (RFS0):

Command Byte Value: 3

This command writes a new variable into a device VID. The device performs any error checking and ignores invalid values/combinations. The Device does not issue a response message for this command. The client must re-Get the VID to determine the results of the Set command. Note that some variables may be read only and will ignore the set command. Some variables may be trigger variables that trigger an action, but do not change the value. This behavior will be described for an individual device.

4.7.1.3.6 Trap (RFS0):

Command Byte Value: 4

This command is issued by a device without prompting from the server. Trap messages indicate status that occurs relative to some condition. Variables may be "Set" to enable and disable traps.

4.7.1.3.7 Show (RFS1):

Command Byte Value: 5

This command is issued to the device for a single variable to cause the device to describe that variable. This command allows an external description of the device (e.g. XML configuration file) to be built that describes all the variables in a device type without necessarily describing the current values of a specific device.

4.7.1.3.8 Format (RFS1):

Command Byte Value: 6

This command is issued from a device in response to a "Show" command for a given VID. This command describes the format of the variable.

4.7.1.3.9 Get_Previous (RFS1):

Command Byte Value: 7

This command causes to the device to respond with the description fields of the previous variable in the database after the most previous Get or Get_Previous command. The device shall automatically wrap the index associated with Get_Previous in a circular fashion. If a Get_Previous command is issued without a previous Get command the 0th variable shall be returned as the starting point. If a Get command is issued then the VID of the variable "gotten" shall form the basis for the Get_Previous command.

4.7.1.3.10 Get_Value (RFS1):

Command Byte Value: 8

This command causes the device to respond with the Value_Is message for the VID that is included in this message. The packet format is identical to the Set message type with the exception that the Set opcode is replaced with the Value_Is opcode.

4.7.1.3.11 Value_Is (RFS1):

Command Byte Value: 9

The Value_Is message is used to convey only the value of a variable and not the variable's meta-data. The Value_Is message is issued in response to a Get_Value message.

4.7.1.3.12 Get Next Value (RFS3):

Command Byte Value: 10

The Get Next Value command allows an external host to "scroll" through all the VIDS and get just the values. The devices increments the VID number from the last Get Value Command to the next VID in the list and returns the value. This command wraps from the last VID to VID 0.

4.7.1.3.13 Get Previous Value (RFS3):

Command Byte Value: 11

The Get Next Value command allows an external host to "scroll" through all the VIDS in reverse order and get just the values. The devices decrements the VID number from the last Get Value Command to the previous VID in the list and returns the value. This command wraps back around from VID 0 back around to last VID.

4.7.1.3.14 Construct (RFS2):

Command Byte Value: 12

The Construct command is used by the client to create the desired contents of a BitArray variable. The device does not respond to this message. The client may issue a "Show" command to determine the results of the Construct command.

4.7.1.3.15 Error (RFS0):

Command Byte Value: 127

The contents of the message shall be implementation defined. The error message is used to indicate to the client if some previous command caused an error condition. The use of this message is optional.

4.7.1.4 Sequence number

Following the command field is a sequence number. This is a single byte that increments with each message to provide duplicate packet rejection or identify multiple transmission unit messages. The sequence number is a modulo-256 value that uniquely indicates a single RFS command. For RFS commands that can be contained in one packet, the sequence number increments for each one. If a RFS packet must span multiple packets the sequence number should be the same for all packets that are part of the same RFS command.

4.7.1.5 VID

Following the command field is one or more bytes describing the VID that is being set or gotten. If the VID is 127 or less the VID shall be contained in a single byte. If the VID is larger than 127 the VID shall be transmitted as multiple bytes, each containing 7 bits of the VID number. All bytes but the last shall contain the high bit (bit 7) of each byte set to one. The last byte of a multi-byte VID shall have the high

order bit set to 0. For most simple systems where 127 VIDS is enough, a single byte with the high bit set to 0 will be transmitted for the VID field. Multi-byte VIDs are transmitted most significant bits first.

4.7.2 Message Data

The message header communicates the desired VID and the desired action on that VID. The command type further identifies the contents of the message data portion of the RFS message. Some messages contain no message data (e.g. Get Next). Most messages contain some data. Message data includes values and meta-data. The values and meta-data are transmitted using the basic data types that have already been described.

Data fields contain one of the following objects:

- An empty object. (E.g. the Get command contains no data.)
- A data description object.
- A data description with value object.
- A value object.
- A BitArray constructor object.
- A text object.

4.7.2.1 Data Description Object

A Data Description Object (DDO) contains meta-data about a VID. The Format command contains Data Descriptions. Data descriptions contain information that is used by the client to display the name, range limits and other information needed by the user to correctly set the value of a variable. Data Description Objects contain the following fields:

4.7.2.1.1 Singularity/Access/Persistence Identifier (RFS0-3)

The first byte of a DDO contains a single byte known as the Singularity/Access/Persistence (SAP) identifier. The upper four bits of this byte identify the variable as a single value (a scalar) (RFS0)) (0x10), a single dimensional array (RFS2) (0x20), a 2 dimensional array (Implemented in RFS Revision 3 (RFS3)) (0x40), or a BitArray (RFS2) (0x80). Bit 0 of this byte has the following encoding: 0 = Read/Write, 1 = Read Only. Bit 1 of this byte has the following encoding: 0 = value is temporary, 1 = value is persistent (survives power cycle). All other bits and combinations are reserved. The remainder of the DDO varies depending on the SAP byte as shown in the following diagram:

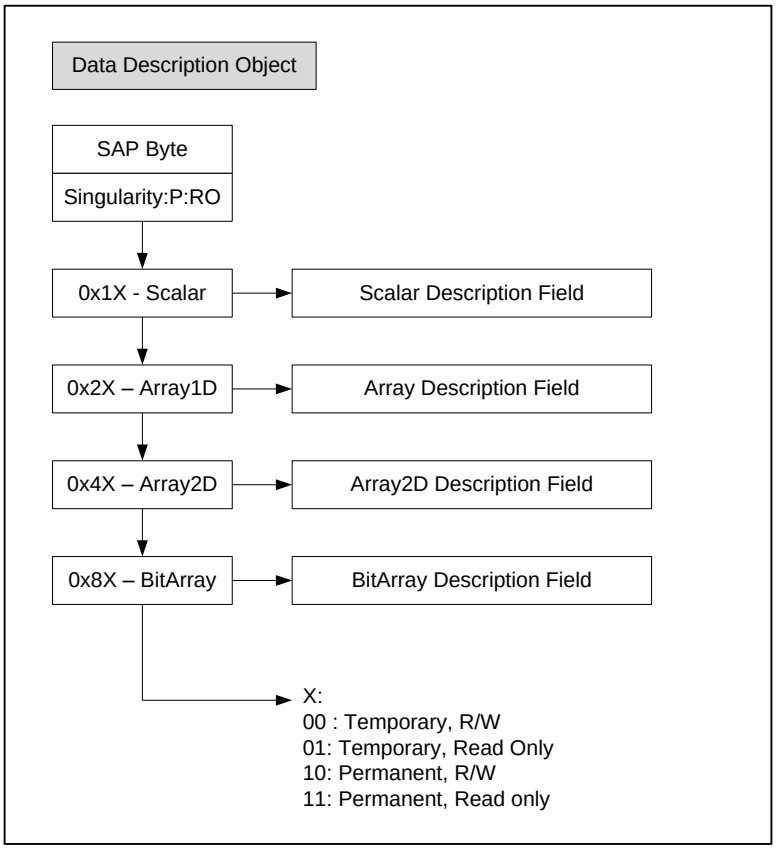

Figure 4-16 Data Description Object (Top Level)

The format of the Description Fields is shown in the diagrams that follow. The byte value in the diagrams labeled Field Size is a single byte that indicates the size of the rest of the field. This is used as an error check to confirm correct delivery of the required amount of data for a given field. This closes a possible hole where a transmission unit may have been properly received without transmission error but the sender incorrectly formatted the message. This prevents the receiver from attempting to decode garbage beyond the end of the message.

4.7.2.1.2 Scalar DDO Field

This diagram represents the format of a Scalar DDO. The first byte indicates the basic data type; the second byte is the field size and indicates the count of the number of bytes remaining to the end of the description (not counting this byte or the type byte). The following name field contains a string (as defined in this document) indicating the text human readable name of the variable. The remainder of the field varies according to the variable type as follows:

- Int32 – The values that follow are the minimum and maximum values allowed for this variable.
- String – The single byte that follows indicates the maximum length string that can be stored. This is inclusive of the count byte and terminating null.

- Ordinal – The first byte value is the maximum value allowed for this value (i.e. number of choices minus 1). Following the maximum value are "maximum value + 1" strings, where the ordering matches the ordinal choices of 0..maximum value.
- Float32/64 (RFS2) – The values that follow are the minimum and maximum values allowed for this variable.
- Fixed32/64(RFS2) – The values that follow are the minimum and maximum values allowed for the value (in signed x.y format). Following the min/max values is a single byte that indicates the number of whole digits in the fraction, *including the sign bit*, (i.e. the x in x.y). The y value is computed as 32-x (or 64-x for a 64 bit fixed value). The whole number range for a fixed point value

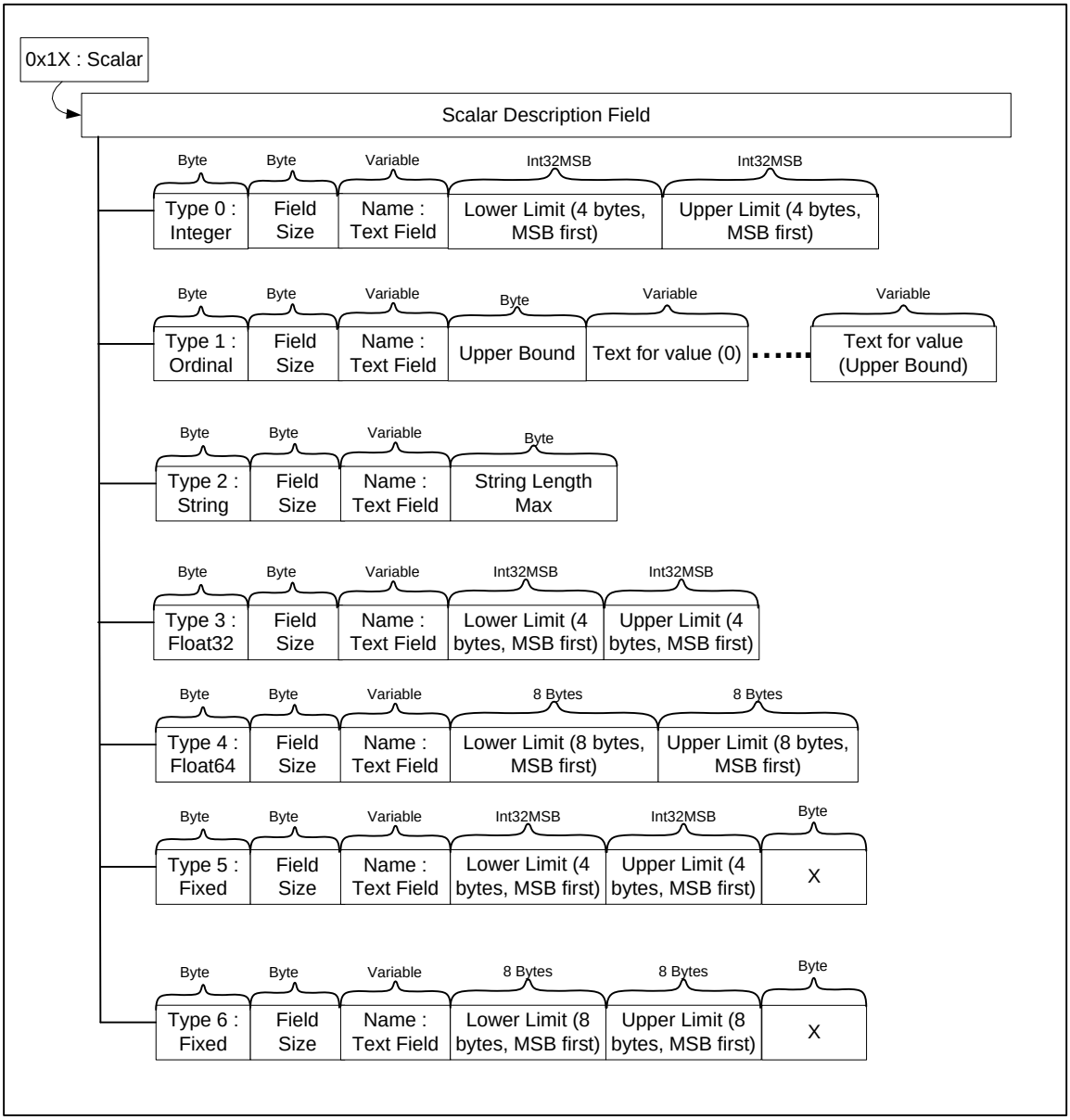

Figure 4-17 Scalar Description Field

4.7.2.1.3 One Dimensional Array (1D) DDO Field

Array DDO's contain the same starting fields (type, field size and name) as the scalars. The remainder of the field varies according to the type being transmitted. The meaning of these individual fields is:

- Elements —This is the number of items stored in the array. The element indices that are implied are 0...elements-1.
- Minimum -- Indicates the minimum value for any and all elements in the array.
- Maximum-- Indicates the maximum value for any and all elements in the array.
- X – Indicates the number of whole digits in a fixed point array.

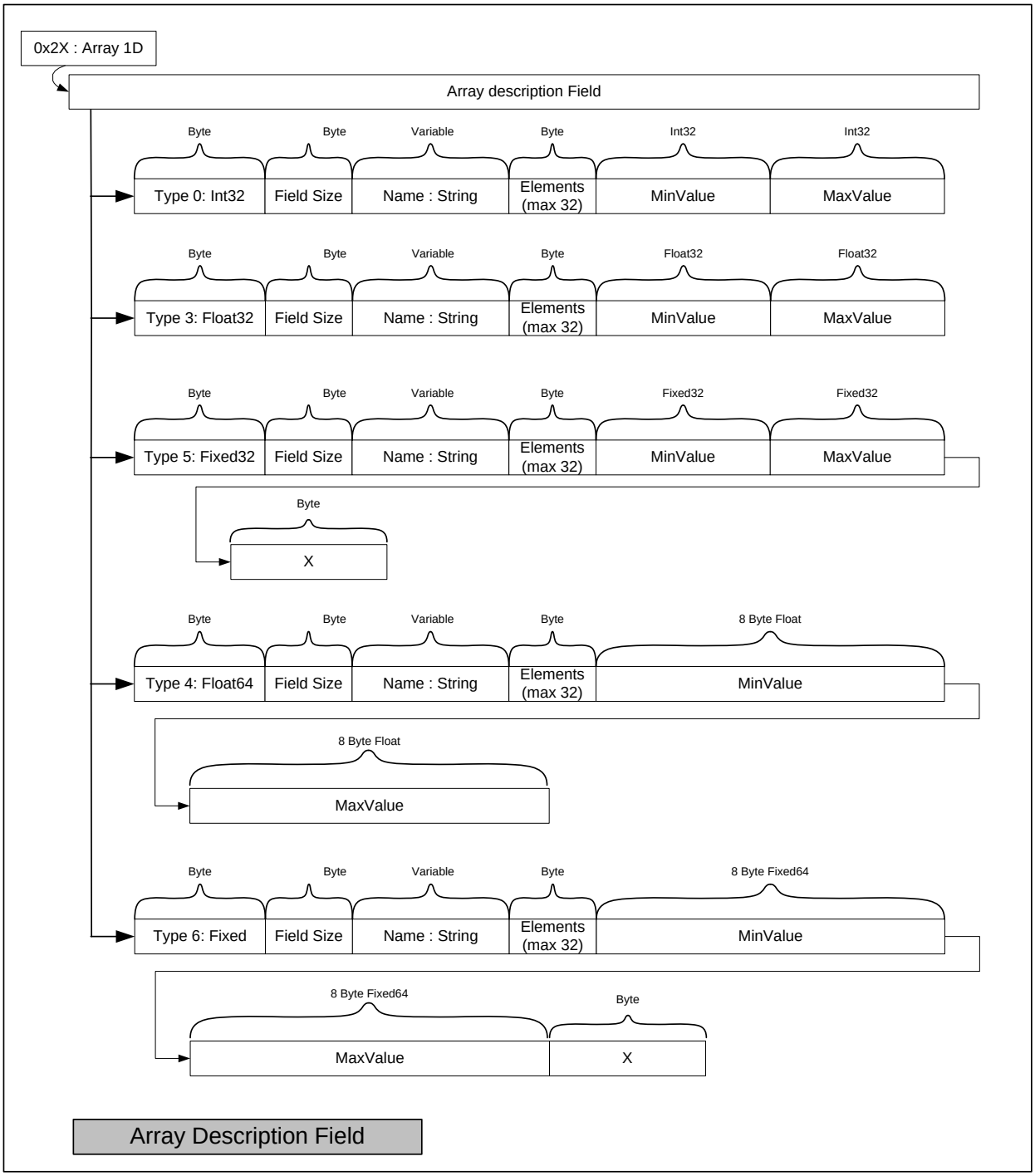

Figure 4-18 Array Description Field

4.7.2.1.4 Two Dimensional Array (2D) DDO Field

Two dimensional arrays are described similar to the one dimensional arrays. The only difference being that instead of a single dimension there are two dimensions, rows and columns. The RFS protocol assumes a row major ordering of 2 dimensional array elements. To clarify, the implication is that the rightmost index elements are stored closest to together in memory. As the array elements must be

transmitted sequentially, a convention must be adopted to avoid differences between computers and software implementations. In this protocol a 2D array declared as array[m][n] is defined to be an array with n columns and m rows. The elements of the rows are stored sequentially in the transmission. For example an array declared as array[3][3] will be transmitted as array[0][0], array[0][1], array[0][2], array[1][0], array[1][1], array[1][2], array[2][0],array[2][1],array[2][2]. The interpretation and usage of the array is application dependent. The protocol merely describes the method of transmission.

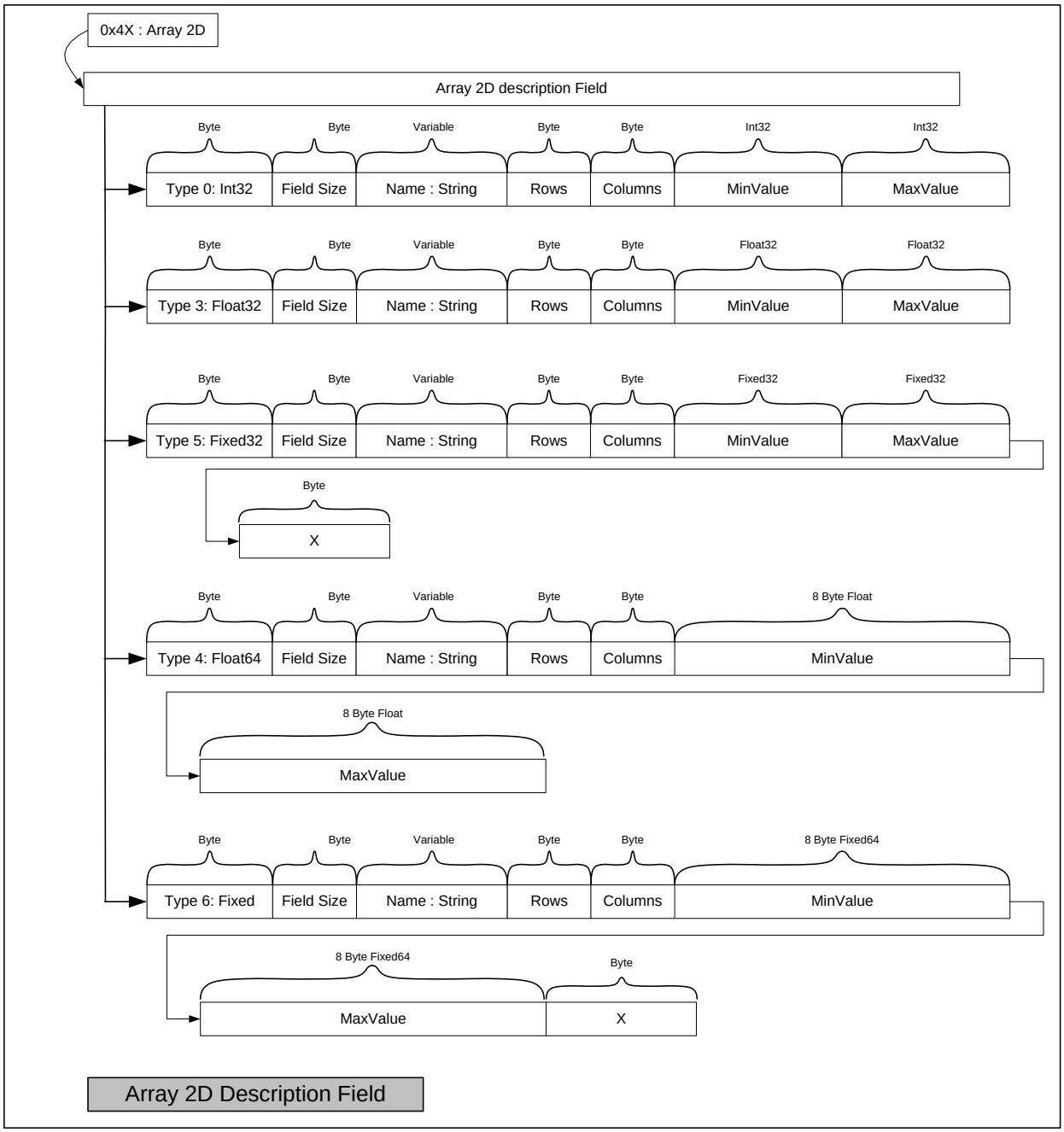

Figure 4-19 Array (2D) Description Field

4.7.2.1.5 BitArray DDO Field

BitArrays are used to transmit multiple VIDs in a single aggregate object. The BitArray DDO contains meta-meta-data. This means that instead of sending the meta-data for a VID, what is transmitted is information to indicate where to locate the meta-data. The BitArray DDO does this by describing a VID, the number of bits used to represent this VID and the bit position in the actual BitArray value when transmitted. The DDO contains a count of the number of possible VIDs that can be added to this BitArray, and a cross reference descriptor for each possible position. Unused positions will contain a cross reference that is all 0's. The number of non-zero descriptors indicates the number of VIDs that will be included in the Value message for this BitArray.

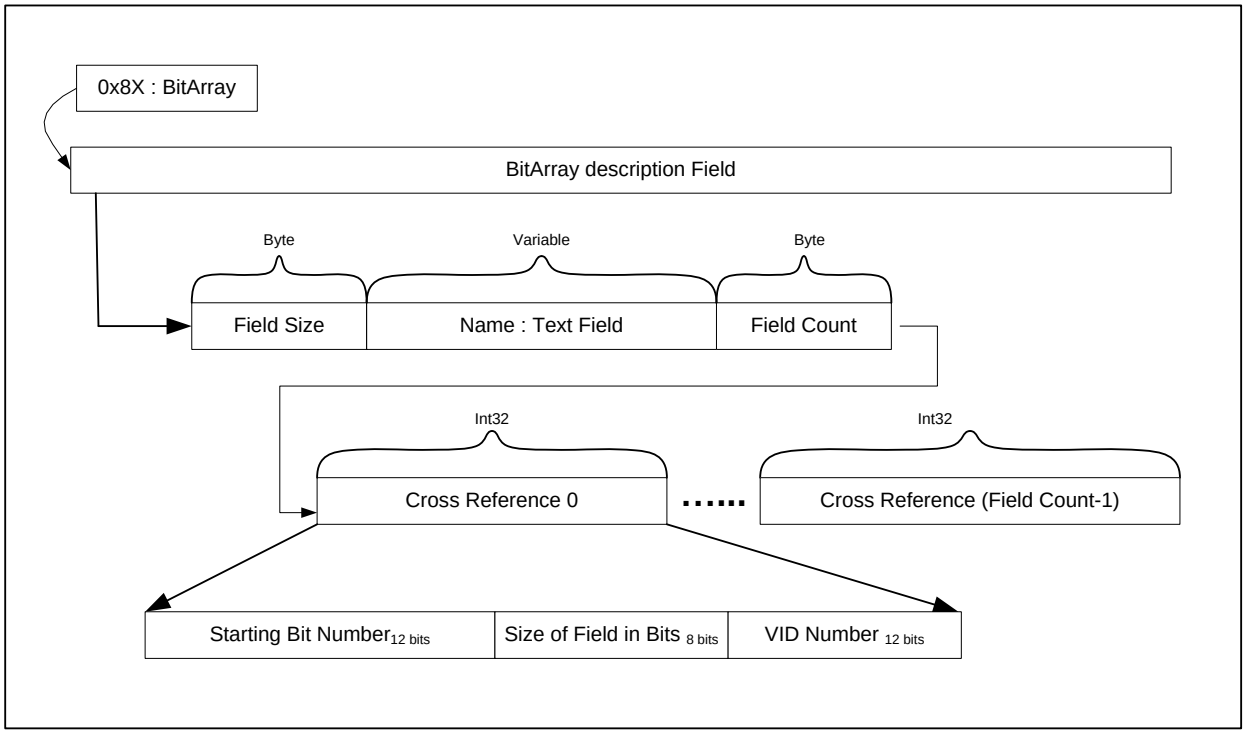

Figure 4-20 BitArray DDO

4.7.2.2 Data Description Object with Value

A Data Description Object with Value (DDO+V) contains the same information in the DDO with the addition of the current value. The current value or values are appended to the DDO. This is shown diagrammatically in the following figures. Note that BitArray description is not sent in the DDO+V field. Because the description itself could be large due to the number of cross referenced variables, when a

Get_Request is issued on a BitArray variable, the response is the BitArray value only, without the description. This is shown in the overall diagram that immediately follows.

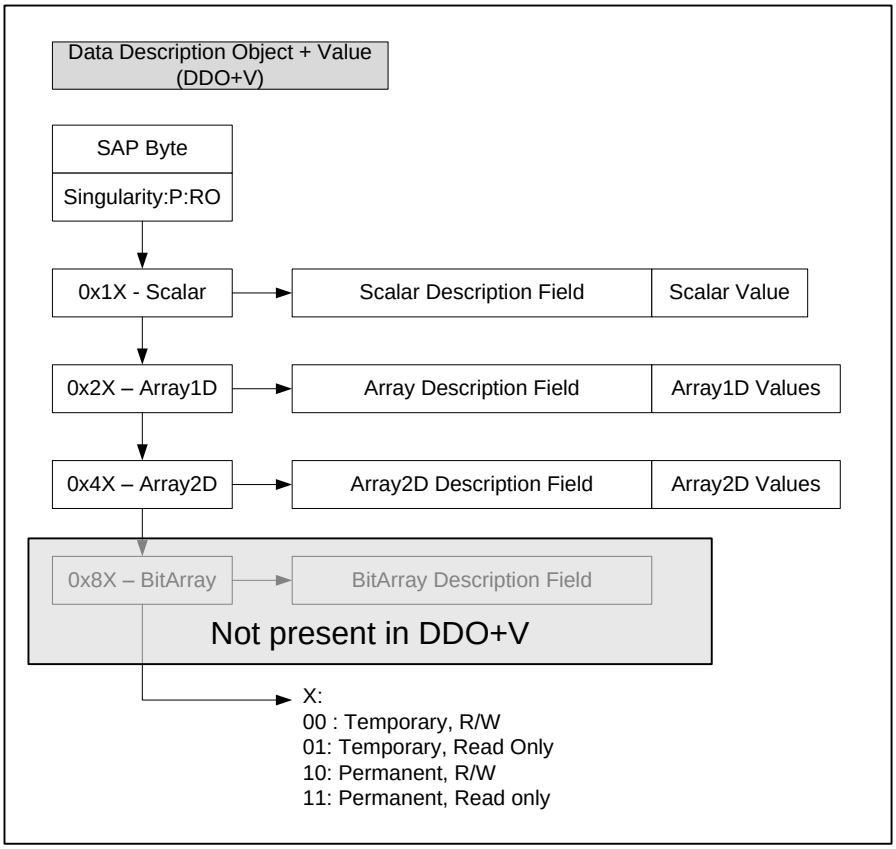

Figure 4-21 DDO+V High Level Format

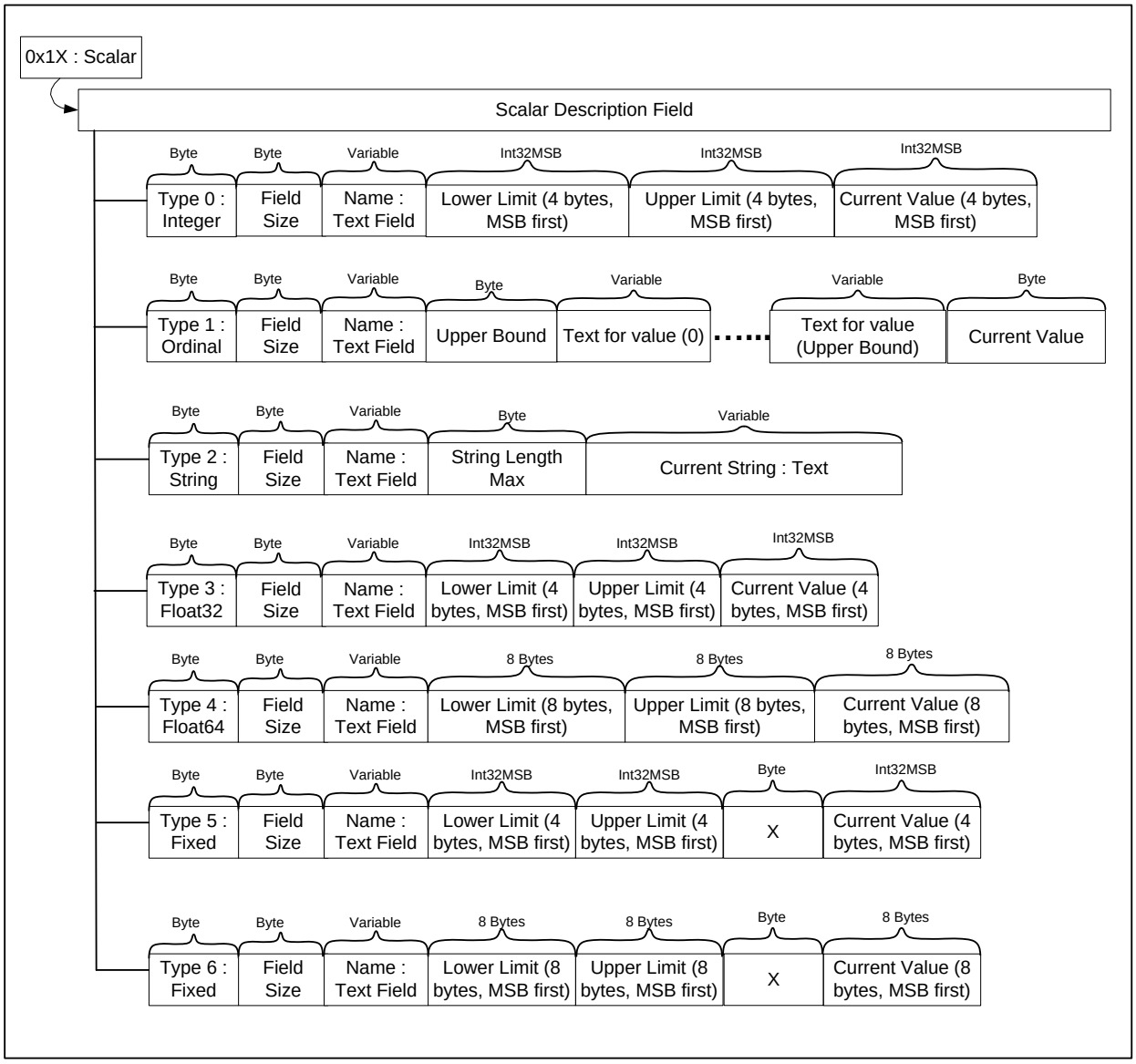

Figure 4-22 DDO+V Scalar Description

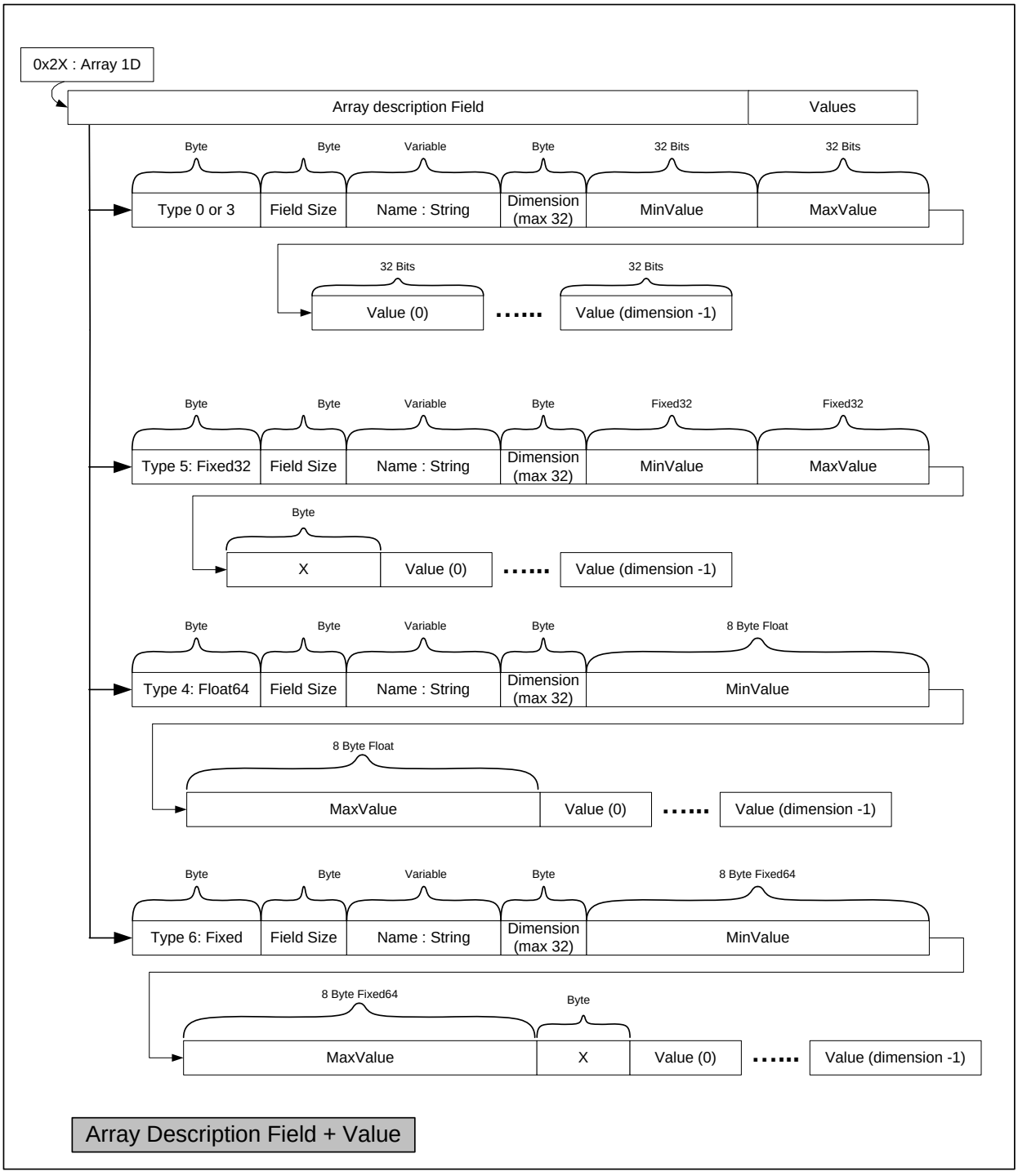

Figure 4-23 Array 1D DDO+V

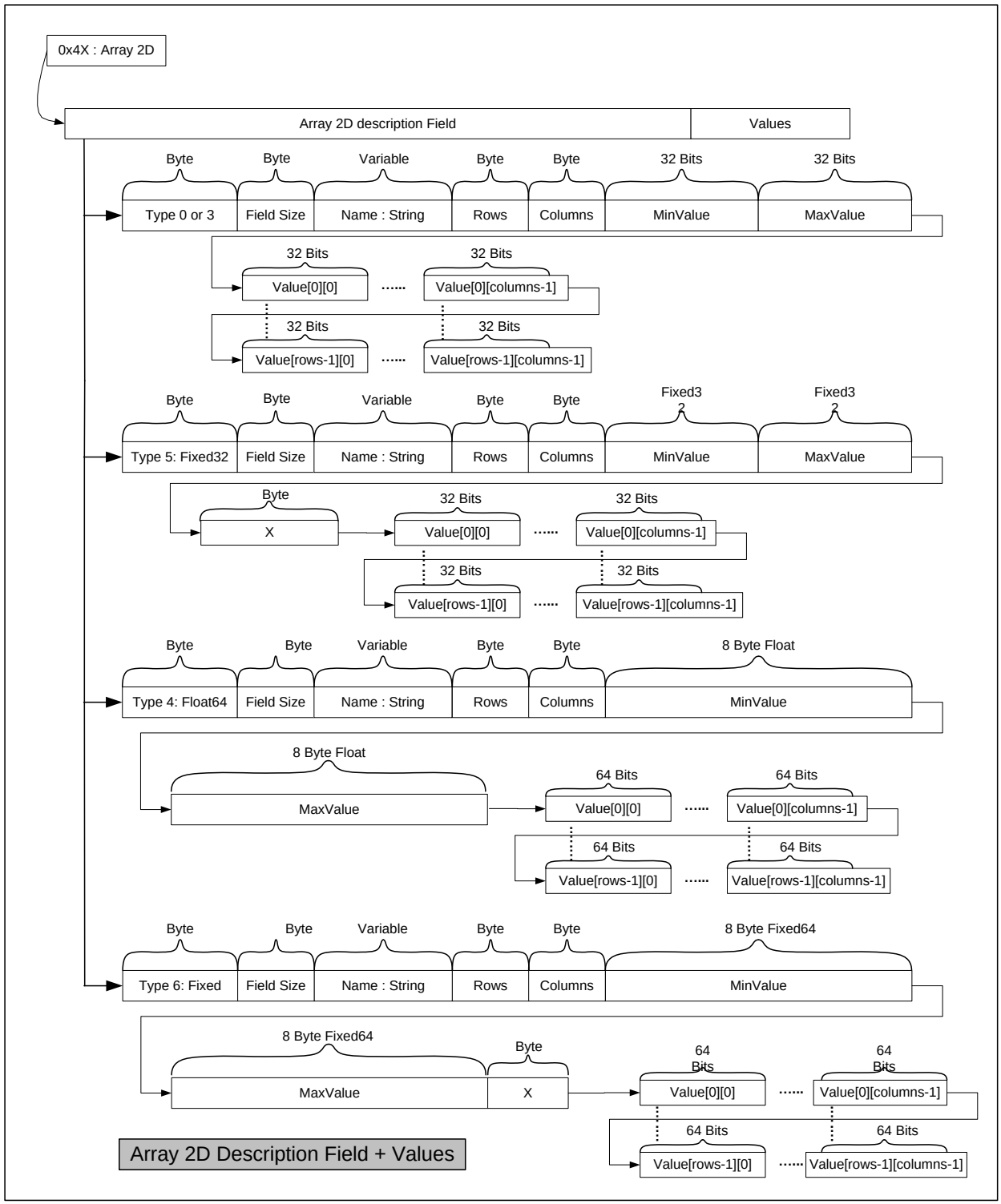

Figure 4-24 Array 2D DDO+V

4.7.2.3 Value Object

The Value Object (VO) contains only the value of a variable without any meta-data. The message is used to Set a value by the client or report a value from the server. The value object includes one byte of SAP. The remainder of the VO varies according to the type specified in the SAP byte. This is shown diagrammatically below:

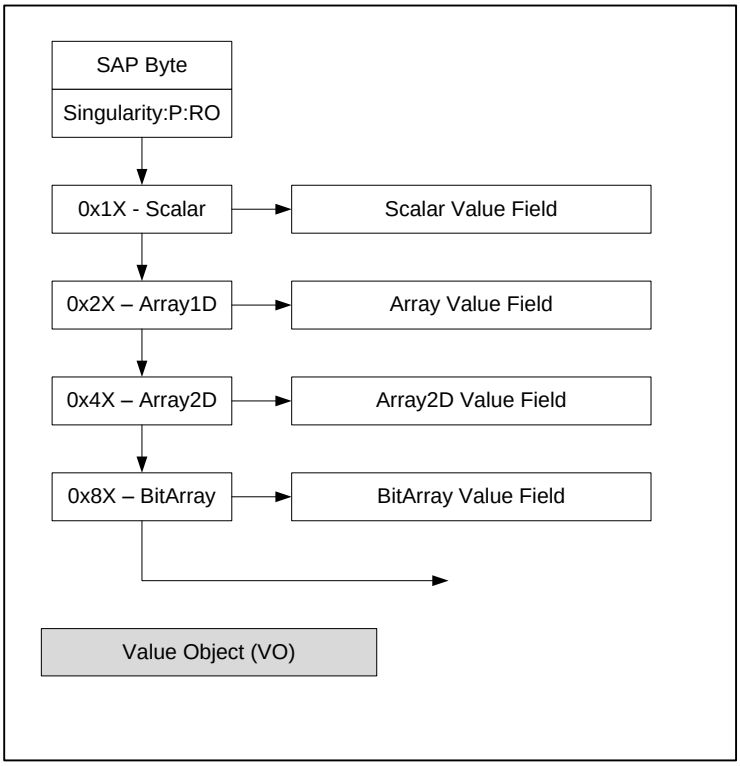

Figure 4-25 Value Object (VO)

4.7.2.3.1 Scalar Value Field

The scalar value field is identified by a 0x1X in the SAP identifier byte that precedes it. The scalar value field first identifies the basic data type being transmitted, and then contains a check byte called field size to ensure that the receiving devices have the correct number of bytes for the data, then the data itself. The field size indicates the actual number of bytes of data that follow the field size itself.

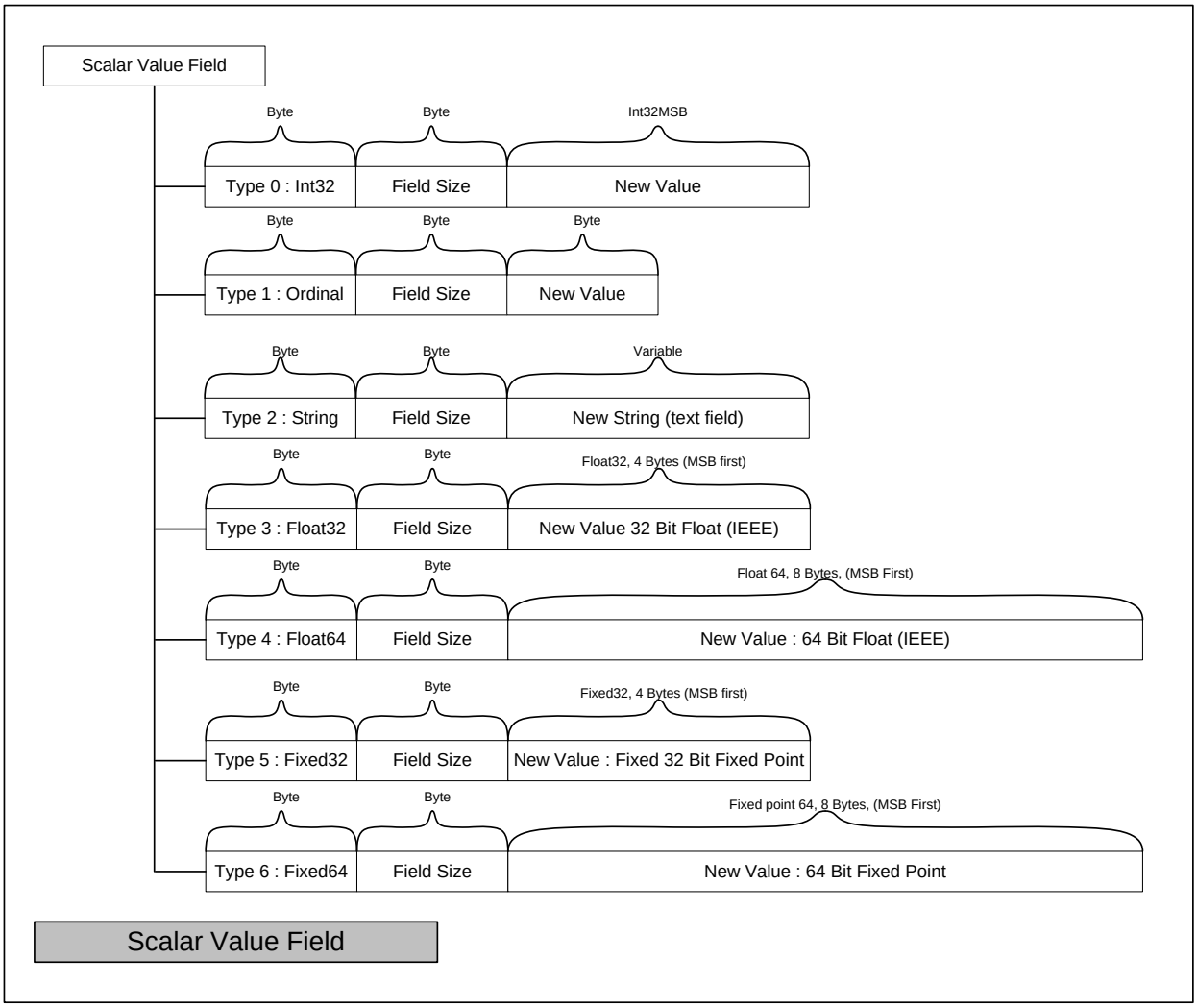

Figure 4-26 Scalar Value Field

4.7.2.3.2 Array 1D Value Field

An array value field is used to set some or all elements of an array variable. The format is shown in the diagram that follows below. The start and stop index are zero based values. For example an array of dimension 3 has possible indices of 0, 1 and 2. The start and stop indices allow setting of subsets of an array. The type tag provides for multiple basic data types as array elements. The field size indicates the remainder of the message size. Two bytes follow indicating the starting and stopping index of the data to follow. A single value may be written into an array at any location. In this case the start and stop index would be the same value. The values to be written follow the stop index. The device will ignore any attempt to set values beyond the range of the variable and will generate an error message if the bounds are violated, or the number of values in the message do not match the amount indicated by the start and stop indices.

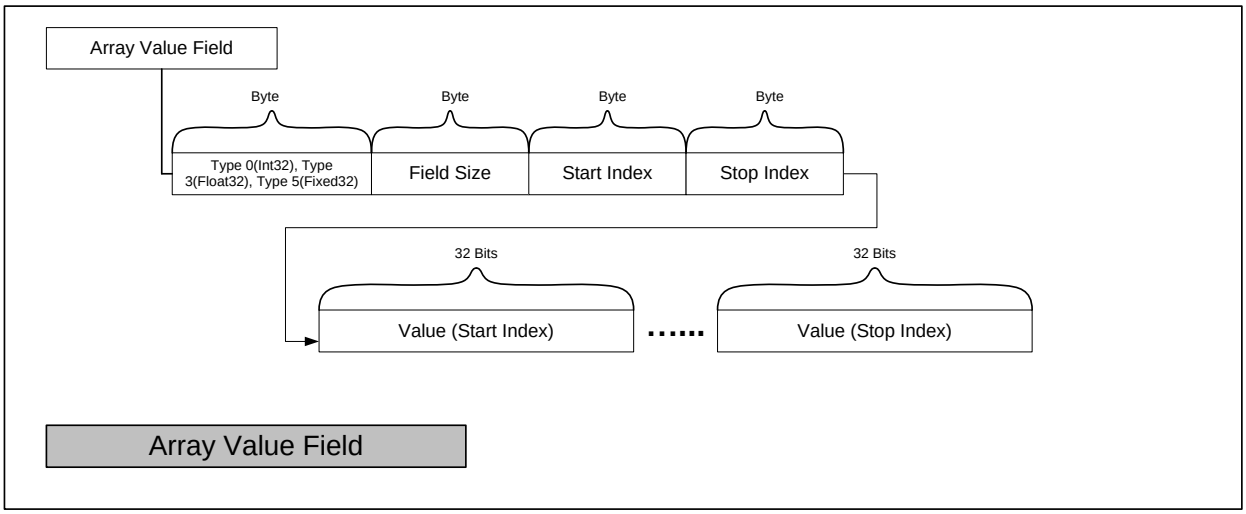

Figure 4-27 Array (1D) Value Field (32 Bit)

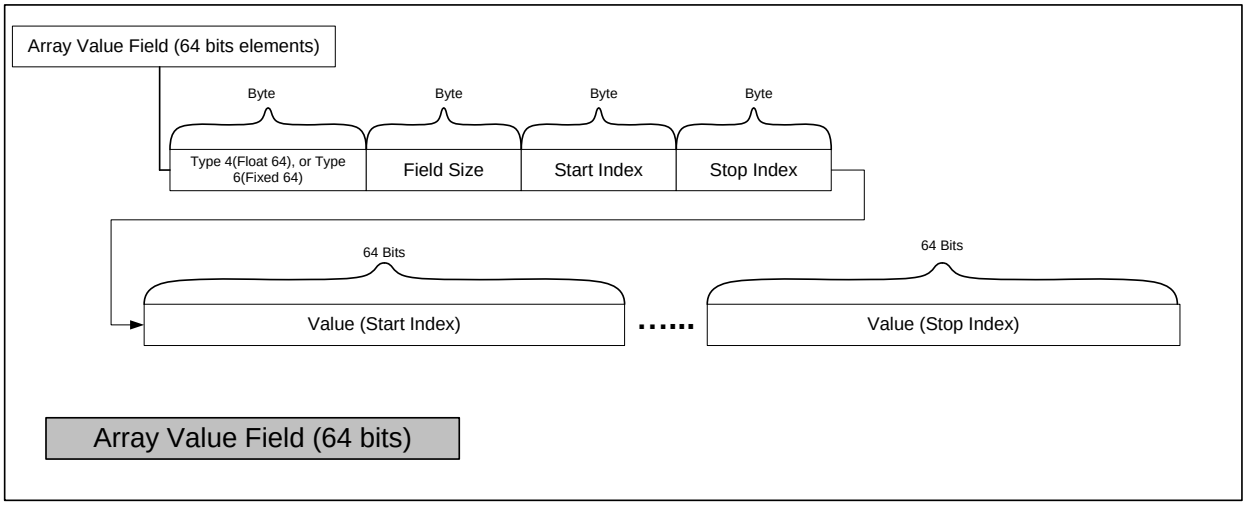

Figure 4-28 Array (1D) Value Field (64 Bit)

4.7.2.3.3 Array 2D Value Field

Two dimensional array values are very similar to one dimensional array value fields. Setting of any portion of a 2D array is possible, however the client must be careful to observe the row-major ordering implied in the data transmission. The format includes a starting row and column and ending row and column. If a starting row/column pair of 0, 0 is selected and an ending row/column pair of (rows-1), (columns-1) is selected the full array is included. If the row/column pairs are not the starting and ending rows of the array respectively, the provided data is expected to be organized sequentially in the message as shown in the following example.

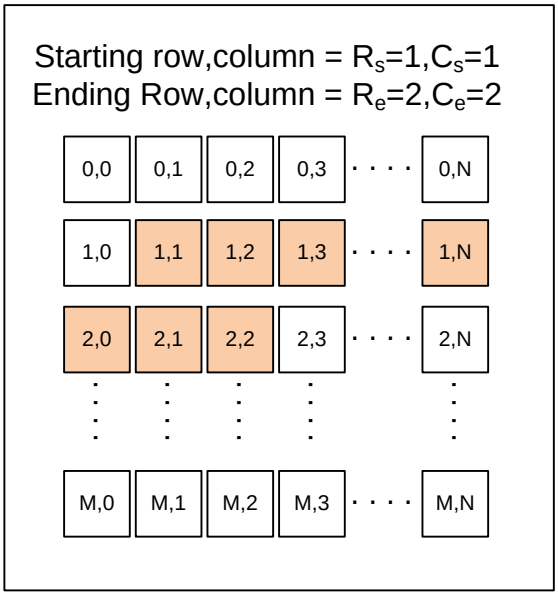

Figure 4-29 2D Array Set Value Subset Example

This example shows that if the specified starting ending rows and columns are specified as they are, the items will be packed into the message as shown starting with element 1, 1, continuing to element 1, N, and finally concluding at element 2, 2.

The general form of the 2D Array Value field is shown below:

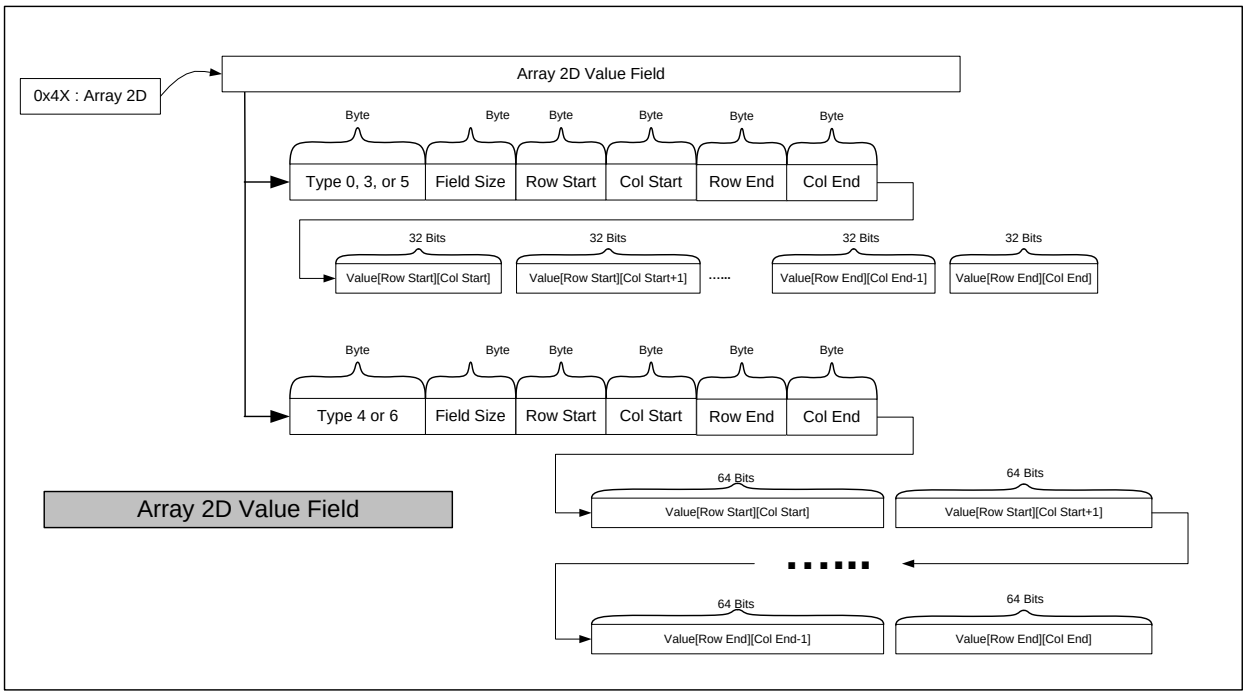

Figure 4-30 Array (2d) Value Field (32/64 bit)

4.7.2.3.4 BitArray Value Field

BitArrays are transmitted as shown in the following diagram. The value N indicates the number of 32 bit words that will be used to transmit the BitArray. The BitArray data may not completely fill the number of transmitted words. The sender shall round the actual number of bits used to transmit the array to the next 32 bit word sized boundary. The combination of N and field size provides some measure of "sanity" checking on the data for validity. Note that the interpretation of the BitArray data implicitly assumes that the client or server has previously retrieved the Data Description Object for the BitArray via a Get message and has also retrieved the individual VID Data Description Objects for the assembled VIDs in the BitArray. The expectation is that the client side equipment will have retrieved the display information once, and will then retrieve the actual values many times over the session.

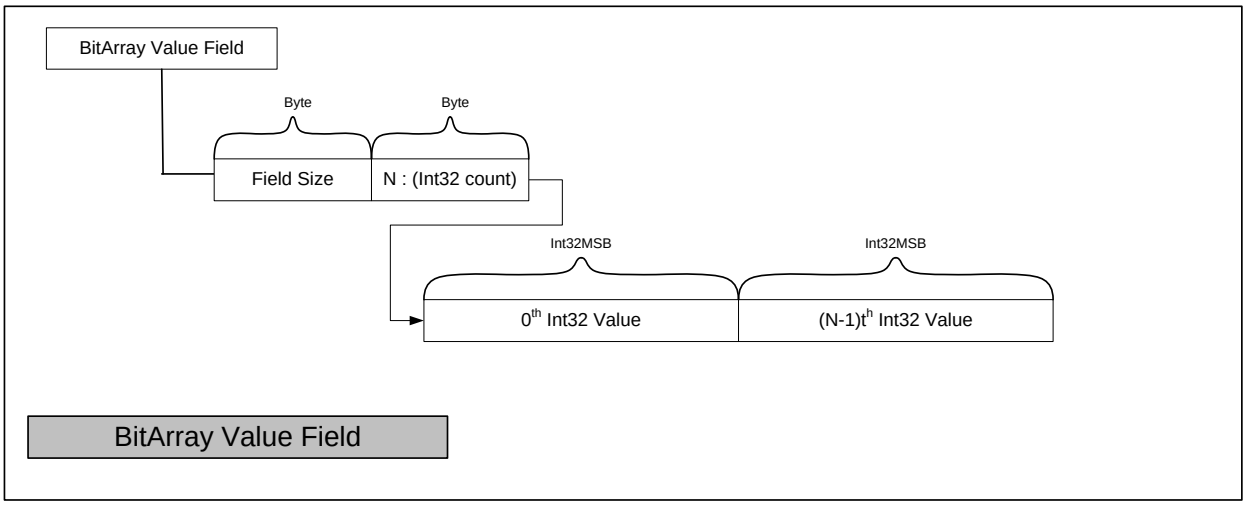

Figure 4-31 BitArray Value Field

4.7.2.4 BitArray Constructor Object

A BitArray Constructor Object (BACO) is used to modify the variables that are packed into a BitArray variable. The BitArray Constructor Object is transmitted in a Construct RFS message. Its format is similar to the DDO for the BitArray with some fields eliminated. There is no SAP byte and the name field is omitted. This message is used by the client to (re)define which VIDs are included in the BitArray data when it is transmitted to the client. The client may send the Construct message, followed by the Get or Show message to confirm that the BitArray definition was constructed as desired.

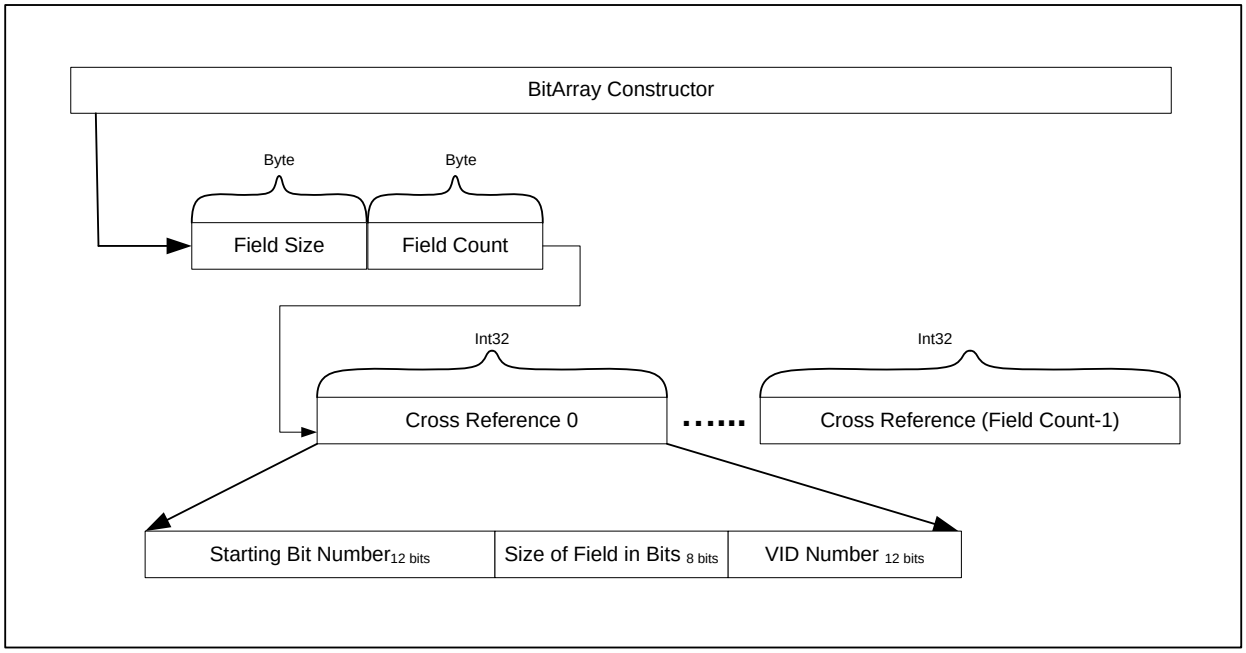

Figure 4-32 BitArray Constructor Object

4.7.2.5 Text Object

A text object contains simple, human readable text in the String format. It is used to display log type messages to the client's human user.

Appendix A Standard VID Values

The following standard VIDS shall be present in every device.

VID 0: VID Count. This variable will always be a scalar Int32 indicating the total number of VIDs in a device. This count shall include the standard VIDs. This VID is read only.

VID 1: Device Name. This variable will always be a scalar string field containing the name of the device. This VID is read only.

(RFS3) VID 2: MajorVersion. This variable shall contain Int32 indicating the major version of the device. This will typically be the major product identifier. This VID is read only.

(RFS3) VID 3: MinorVersion. This variable shall contain an Int32 value that is interpreted as the minor version number. This VID is read only.

Deprecated: This vid is no longer required to be vid #2 [VID 2 : Quiet. This is an Int32 scalar variable. Setting this value to 0 suppresses all traps and/or debug output. Setting this variable to any other value enables all previously enabled traps and/or debug output.]

Appendix B RFS Version Capability Matrix

Functionality/Version Matrix

Version	Feature/Command
0	Basic frame structure, Basic data types of String, Int32, and Ordinal. Message types of Get, Get_Response, Set, Get_Next
1	BitArrays containing integers and ordinals, Get_Previous, Show, Format, Get_Value, Value_Is commands.
2	Arrays, Float32, Fixed32, Float32, Float64. Added arrays to BitArrays. Added BitArray Construct message.
3	Read only bit and Persistence bit on SAP byte. 2D Arrays. Two more required vids.

Appendix C Sample RFS Database XML file

```
<?xml version="1.0"?>
<embeddedDB name="config">
	<item name="VIDCount">
		<longname>Variable Count</longname>
		<type>Int32</type>
		<flags>DATA_READONLY</flags>
		<defaultValue>0</defaultValue>
		<minValue>0</minValue>
		<getfunction>N</getfunction>
		<setfunction>N</setfunction>
		<maxValue>1000</maxValue>
		<units>None</units>
	</item>
	<item name="Name">
		<longname>Device Name</longname>
		<type>String</type>
		<flags>DATA_READONLY</flags>
		<dimension>80</dimension>
		<defaultValue>An Embedded Device</defaultValue>
		<units>None</units>
	</item>
	<item name="MajorVersion">
		<longname>Major version number for this software</longname>
		<type>Int32</type>
		<flags>DATA_READONLY</flags>
		<defaultValue>0</defaultValue>
		<minValue>0</minValue>
		<getfunction>N</getfunction>
		<setfunction>N</setfunction>
		<maxValue>1000</maxValue>
		<units>None</units>
	</item>
	<item name="MinorVersion">
		<longname>Major version number for this software</longname>
		<type>Int32</type>
		<flags>DATA_READONLY</flags>
		<defaultValue>0</defaultValue>
		<minValue>0</minValue>
		<getfunction>N</getfunction>
		<setfunction>N</setfunction>
		<maxValue>1000</maxValue>
		<units>None</units>
	</item>
	<item name="serialnumber">
		<longname>Serial number for this unit</longname>
		<type>String</type>
		<flags>DATA_PERMANENT|DATA_PROTECTED</flags>
		<dimension>20</dimension>
		<defaultValue>undefined</defaultValue>
		<units>None</units>
	</item>
	  <item name="orientation">
	    <longname>Physical orientation of device</longname>
	    <type>Ordinal</type>
```

```
      <flags>DATA_PERMANENT</flags>
      <defaultValue>0</defaultValue>
      <maxValue>4</maxValue>
      <labels>
        <label>Horizontal</label>
        <label>Vertical</label>
        <label>Left Edge</label>
        <label>Right Edge</label>
        <label>Inverted</label>
      </labels>
      <setfunction>Y</setfunction>
      <units>Furlongs</units>
    </item>
<item name="baud">
      <longname>User Port Baud rate</longname>
      <type>Ordinal</type>
      <flags>DATA_PERMANENT</flags>
      <defaultValue>8</defaultValue>
      <maxValue>8</maxValue>
      <labels>
            <label>300</label>
            <label>1200</label>
            <label>2400</label>
            <label>4800</label>
            <label>9600</label>
            <label>19200</label>
            <label>38400</label>
            <label>57600</label>
            <label>115200</label>
      </labels>
      <units>Bits Per Second</units>
</item>
<item name="pitch">
      <longname>Pitch angle in degrees</longname>
      <type>Float32</type>
      <flags>DATA_READONLY</flags>
      <defaultValue>0.0</defaultValue>
      <minValue>-180.0</minValue>
      <maxValue>180.0</maxValue>
      <units>Degrees</units>
</item>
<item name="roll">
      <longname>Roll angle in degrees</longname>
      <type>Float32</type>
      <flags>DATA_READONLY</flags>
      <defaultValue>0.0</defaultValue>
      <minValue>-180.0</minValue>
      <maxValue>180.0</maxValue>
      <units>Degrees</units>
</item>
<item name="yaw">
      <longname>Yaw angle in degrees</longname>
      <type>Float32</type>
      <flags>DATA_READONLY</flags>
      <defaultValue>0.0</defaultValue>
      <minValue>0.0</minValue>
```

```
      <maxValue>360.0</maxValue>
      <units>Degrees</units>
</item>
<!-- A Bit field definition -->
<item name="Direction">
      <longname>Composite object with Roll,Pitch,Yaw</longname>
      <type>BitField</type>
      <flags>DATA_PERMANENT</flags>
      <dimension>16</dimension>
      <!-- This is the maximum number of fields possible -->
      <fields>
            <!-- This determines the default number of fields -->
            <field>
                  <vidName>pitch</vidName>
                  <bits>32</bits>
            </field>
            <field>
                  <vidName>roll</vidName>
                  <bits>32</bits>
            </field>
            <field>
                  <vidName>yaw</vidName>
                  <bits>32</bits>
            </field>
      </fields>
      <getfunction>N</getfunction>
      <setfunction>N</setfunction>
      <units>None</units>
</item>
<item name="lattitude">
      <longname>Latitude as 9.23</longname>
      <type>Fixed32</type>
      <flags>DATA_PERMANENT</flags>
      <defaultValue>0.0</defaultValue>
      <minValue>-90.0</minValue>
      <maxValue>90.0</maxValue>
          <wholedigits>9</wholedigits>
      <units>Degrees</units>
</item>
<item name="longitude">
      <longname>Longitude 9.23</longname>
      <type>Fixed32</type>
      <flags>DATA_PERMANENT</flags>
      <defaultValue>0.0</defaultValue>
      <minValue>-180.0</minValue>
      <maxValue>180.0</maxValue>
          <wholedigits>9</wholedigits>
      <units>Degrees</units>
</item>
<item name="alt">
      <longname>Altitude</longname>
      <type>Float32</type>
      <flags>DATA_PERMANENT</flags>
      <defaultValue>0.0</defaultValue>
      <minValue>-10000.0</minValue>
      <maxValue>1000000.0</maxValue>
```

```
            <units>Degrees</units>
        </item>
    <item name="reserved">
      <longname>Reserved object</longname>
      <type>Int32</type>
      <flags>DATA_READONLY</flags>
      <defaultValue>0</defaultValue>
      <minValue>0</minValue>
      <getfunction>N</getfunction>
      <setfunction>N</setfunction>
      <maxValue>1</maxValue>
      <units>None</units>
    </item>
</embeddedDB>
```

Appendix D Implementation Notes

This section describes a typical implementation process and gives some debug and development suggestions. The assumption is that the reader intends to develop both the embedded device and the client device. The typical application is a microcontroller based embedded device with a UART connected to a PC using one of the COM ports. The following diagram shows a suggested architecture. The UART layer can be somewhat combined with the link layer as described in the suggestions that follow. It is recommended that an ISO layering approach be taken facilitate debug and reuse.

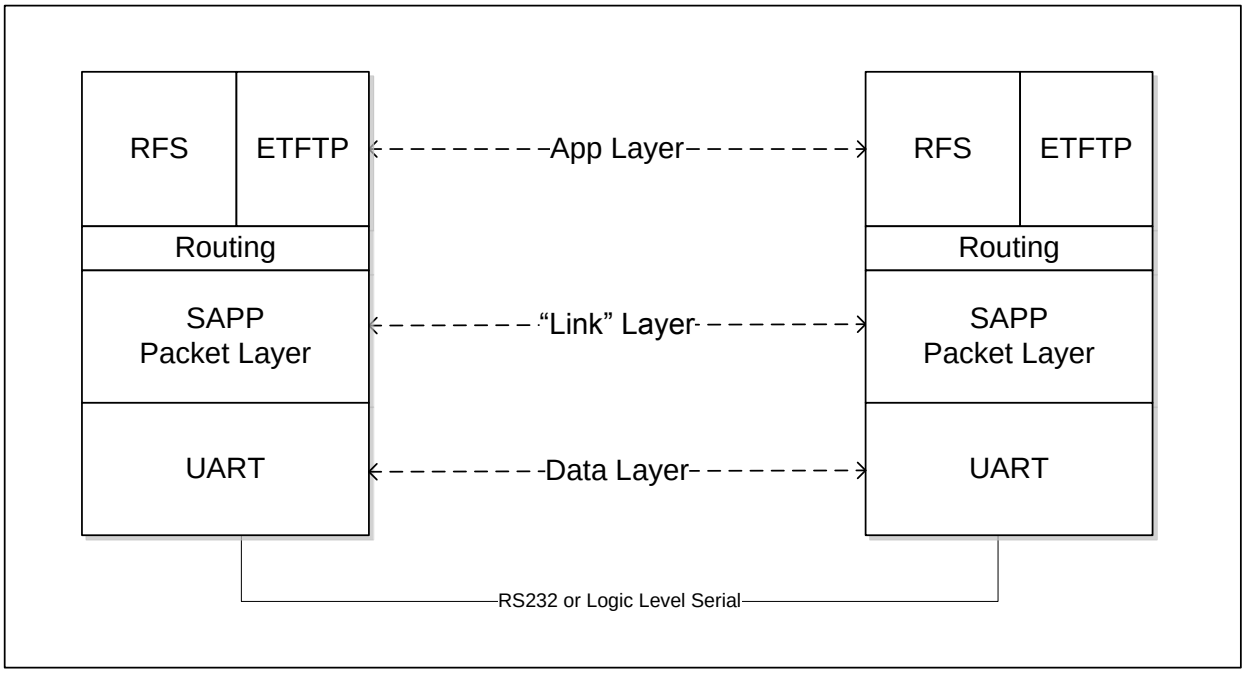

Figure D.4-33 Suggested Protocol Stack Implementation

Some suggestions and tips are:

- Write the SAPP driver to operate independently of the RFS/ETFTP protocol stack. Get this to work first at the basic packet level.
- Decide if the SAPP driver will be tightly integrated with the UART driver or not. A tightly integrated driver will perform the de-escaping on receive and the escape insertion on transmit in the interrupt service routine. The ISR can remove and insert the SOH/ETX bytes as well. This solution uses less memory as the message arrives in the buffer already de-escaped. Choose this option if you are memory limited. Note that this approach spends more time in the ISR and increases the latency, if that is an issue.
- Computing the CRC can be done one byte at a time, or over the whole message. The decision depends on how much time you can afford to spend in the ISR. A runningCRC function is included in the sample code. The partialCRC that is passed in needs to be initialized with the INITIAL_CRC_SEED before the first call, then the running CRC value gets passed back in for each incremental CRC calculation.

- Transmitting the ACK/NAK can be done in the ISR if you choose to also do the CRC calculation there. Perfuming the CRC in this way results in a simpler solution in the packet driver compared to doing things in a thread or in the main polling loop.
- It's a good idea to make sure the packet driver checks for packets that get too big before receiving the ETX. A timeout for packets that never complete is also a good idea.
- If the ISR is not performing the de-escaping, the receive ISR is very simple. It just receives bytes until an ETX is received. Since the protocol only allows the ETX and the end of a packet, this is an easy test.
- Making the RFS process standalone makes it easy to debug. Test fixtures can be written to test the RFS portions without being bogged down in serial port issues.
- If you are using an RTOS and/or using an IrDA or other protocol stack, you will already have a memory allocation and de-allocation strategy. For very small systems with just a serial port, once the VID and command are extracted from the RFS message, the buffer can be re-used to send the response.
- The creation of a small library of functions or macros to perform the following actions is very helpful in taking the protocol apart and putting outbound messages together: extractInt32MSB(), extractFloat32MSB(), insertInt32MSB(), insertFloat32MSB(), etc. Remember that the protocol does not bother with native word boundaries, so casting pointers and overlaying structs is likely to fail in weird ways. Keep in mind that some messages have variable length strings before Int32 values so the Int32 alignment can be at any byte position.
- The field size bytes are helpful in skipping to the next field in a message.
- There are multiple ways to connect the RFS messages to the control logic for your device or client application. A good idea is to first extract the over the air form into native data types (using the functions/macros described above) then using the native form to execute some action. A very modular, flexible solution is created if the back-end data/control engine accepts "commands" that are equivalent to the RFS commands, but with native values. Then the RFS stack basically just has to error check and extract to the local form. A bonus is that when porting the stack to a new device the only thing that has to change is the native back-end logic for new variables and/or actions.

Appendix E Standard Asynchronous Packet Protocol

Revision History

DATE	DESCRIPTION OF CHANGE	INITIALS
7/03/07	Original	CMN
5/2/08	Reformatted	CMN
9/30/09	Modified to include error field to control ACK/NAK behavior	CMN
8/25/11	Included with RFS document. Minor typo corrections and formatting.	CMNJr

E.1 Scope

This document describes a method for sending binary data in packets using a standard asynchronous serial port (UART). The purpose of this document is to provide the developer with the proper information and examples from which software can be developed to implement a packet transmitter and receiver. This document does not describe the contents of the binary payload carried within; rather it describes a simple framework that is capable of carrying any arbitrary binary payload of up to 253 bytes. This is only a framing/link layer. Provision is made to carry payloads larger than 253.

E.2 Background

This document describes a standardized method for transmitting data over an asynchronous serial link between two devices. The purpose of this protocol is to create a standard way to communicate a block of arbitrary binary data from one system to another. This protocol is not designed to carry high speed or synchronous data, but is intended to carry command, control, configuration and status information between devices. This document does not address the physical, or data layer that addresses the signaling conventions and actual transmission details. This protocol assumes an underlying layer that will transmit a single byte or receive a single byte. In some cases this protocol layer will be tightly integrated with the software layer below for convenience or efficiency. However, this protocol does not assume nor enforce an underlying transmission model. This protocol may be carried over a standard serial port, a RS485 link, a virtual serial link such as an IRDA IrCOMM channel, or some other real or virtual channel. This document does not specify an Applications Programming Interface (API) however it is expected to result in a fairly standard software component that may be reused on different devices.

E.3 Communication Protocol

E.3.1 Protocol Overview

This protocol is designed to carry relatively low speed, non time-critical information from one system to another. This protocol may be carried on a variety of serial links that may be full or half duplex. The protocol allows for a synchronization character to be transmitted between frames to aid in clock recovery or other synchronizations issues.

This protocol provides error detection, but not error correction. This protocol provides for acknowledgement of the reception of each packet. A packet may be retried, but it is left up to the application above the protocol layer to retry the packet. The protocol does not perform retries intrinsically.

This protocol uses framing characters to indicate the boundaries of a packet of data.

This protocol is designed to carry binary data. Since binary data will contain some of the framing characters this protocol is designed to modify naturally occurring framing characters so that they are not confused with the real packet framing. This is accomplished by “escaping” each naturally occurring framing character. The escaping process involves sending a reserved character and the modifying the naturally occurring framing character in a predictable way. The receiver detects the reserved character (the “escape”), removes it and restores the following character to its original value. Naturally, because of this process, the transmitted data will often take more bytes to transmit than are contained in the original message. This effect is acceptable as the protocol is not designed to carry time sensitive information.

Each block of transmitted data shall be called a packet. The block of transmitted data shall be called the payload.

E.3.2 Packet Transmission Order

Each sender may transmit a packet at any time. Thus two packets may be in progress simultaneously. However a device may not begin transmission of another packet until the acknowledgement is received from the receiver of the packet. This provides low power devices with limited memory to implement the protocol and pace the reception of packets.

E.3.3 Control Characters

The following characters are deemed control characters for the protocol. In other words these characters indicate special meaning during transmission, other than their use as packet data. The control characters are as follows:

Table E.1 – Protocol Control Characters

Character Name	Hex Value	Description
DLE	0x10	Data Link Escape. The "escape" character. This character precedes an "escaped" control character that occurs in the data, including a DLE
SOH	0x01	Start of Header. This character begins a packet frame. This character only occurs once in the packet frame.
ETX	0x03	End of Text. This character closes a packet frame. This character only appears once in a packet, as the closing character.
SYN	0x16	Synchronous Idle Character. This character may be transmitted between packets for the purposes of maintaining the clock circuitry on the receiving end. This character never appears in a packet.
ACK	0x06	Acknowledgement. This character is transmitted by itself to acknowledge reception of the current packet. A packet is successfully received when the framing is intact and the CRC checksums correctly on the received packet.
NAK	0x15	Negative Acknowledgement. The single character is sent by itself to indicate a reception of characters that was not properly framed or one in which the CRC did not checksum correctly.

E.3.4 Packet Frame

Each packet shall consist of an opening SOH and end with a closing ETX. The packet body falls between the framing characters. Between the opening SOH and closing ETX the only allowable control character shall be a DLE. Naturally occurring control characters will be replaced with the two character sequence DLE, (0x80 | character). In other words, any time one of DLE, SOH, SYN, ACK, NAK, ETX occurs in the message between the SOH and ETX, the one character is replaced by two characters, first a DLE character, then the offending character that is modified by bitwise OR'ing it with 0x80. This quote mechanism guarantees that the SOH and ETX only appear at the beginning and end of a packet. The receiving end shall detect the DLE character, remove it from the stream and modify the next character by anding off the high bit.

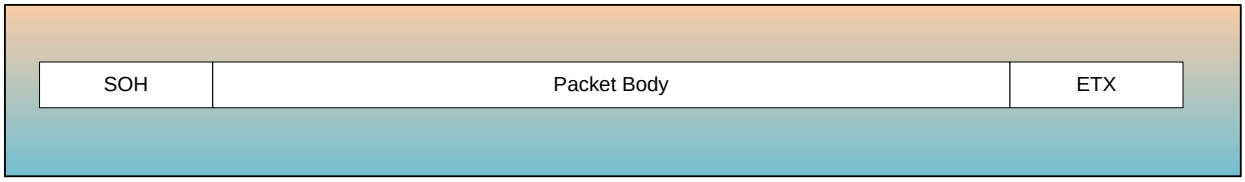

Figure E.4-34 – Packet Frame

The packet body is as follows:

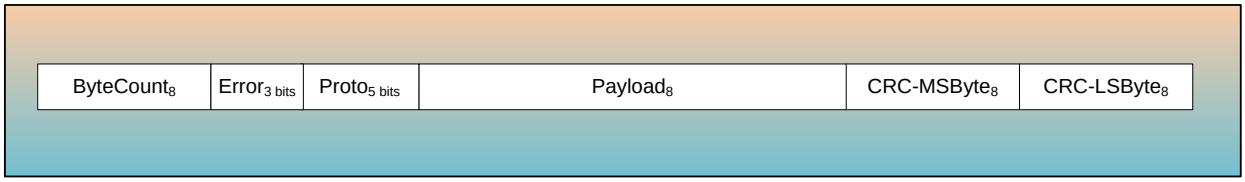

Figure E.4-35 – Packet Body

The byte-count shall be from 3 to 255 or a special value of 0. **If the byte-count value is zero the payload size is determined by the transmission layer**. This allows for packets to be larger than 253 bytes of payload. The use of packets bigger than 256 is system dependent. The minimum payload size is one byte. The payload field consists of the Error/Protocol byte plus the payload values. The byte-count shall describe the total length of the payload field plus the two CRC bytes. Thus the minimum byte count value shall be 3 (if the payload was 0 length). Since 255 is the maximum value in a byte sized value, the maximum payload that can be carried is 253 (255 – 2 CRC bytes). This ignores any growth in the payload due to escaping. The Error/Protocol field (EP) describes the error handling requested for this packet and the protocol of the payload field. Note that the Bytecount, EP field and CRC bytes are subject to the DLE mechanism should they naturally be control characters.

The transmission of a packet from sender to receiver is shown in the sequence diagram below:

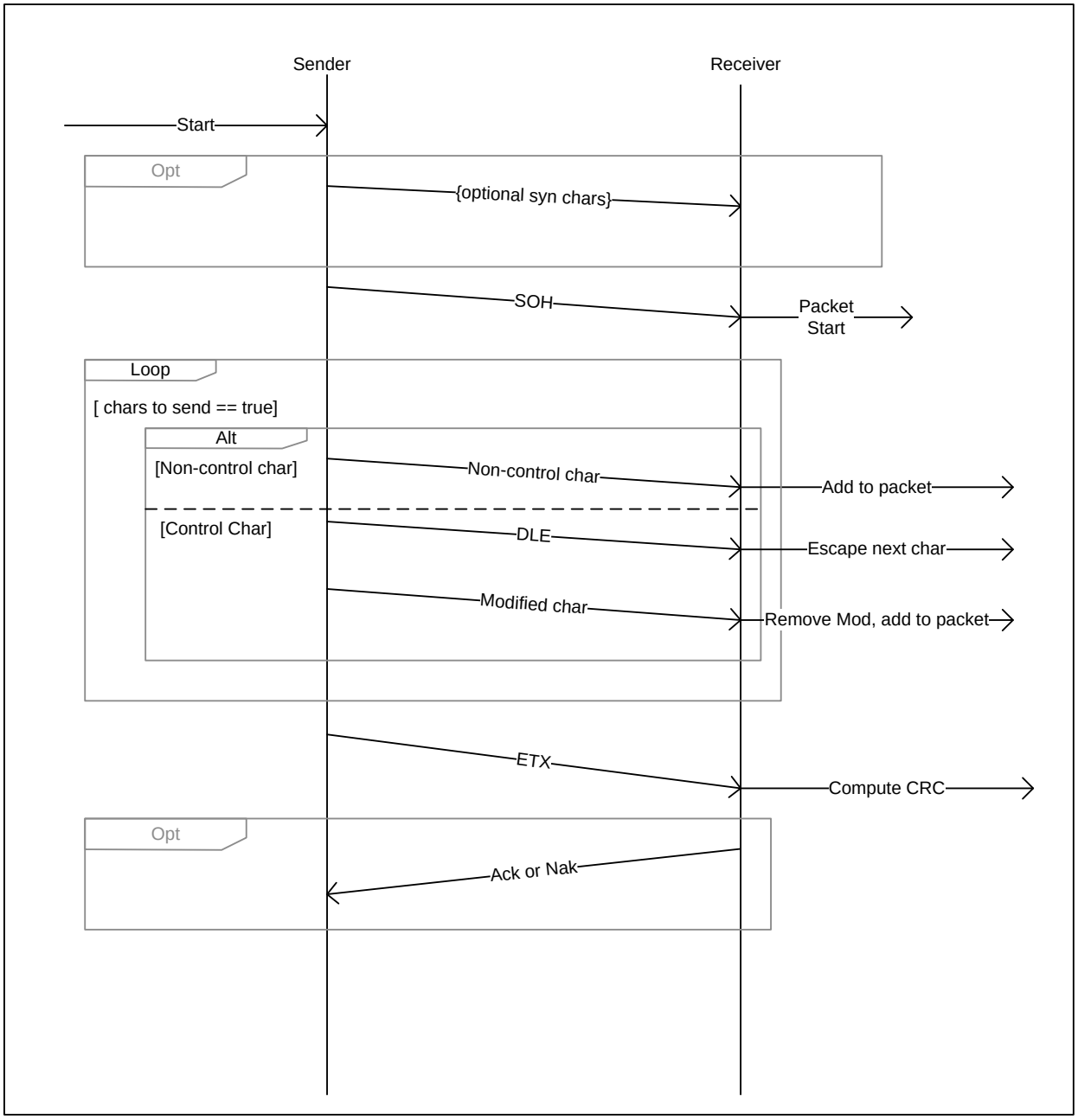

Figure E.4-36 SAPP Sender Receiver Sequence Diagram

The Error/Proto fields are used to indicate which protocol is being carried in the payload field and the error handling behavior. These fields are encoded as shown:

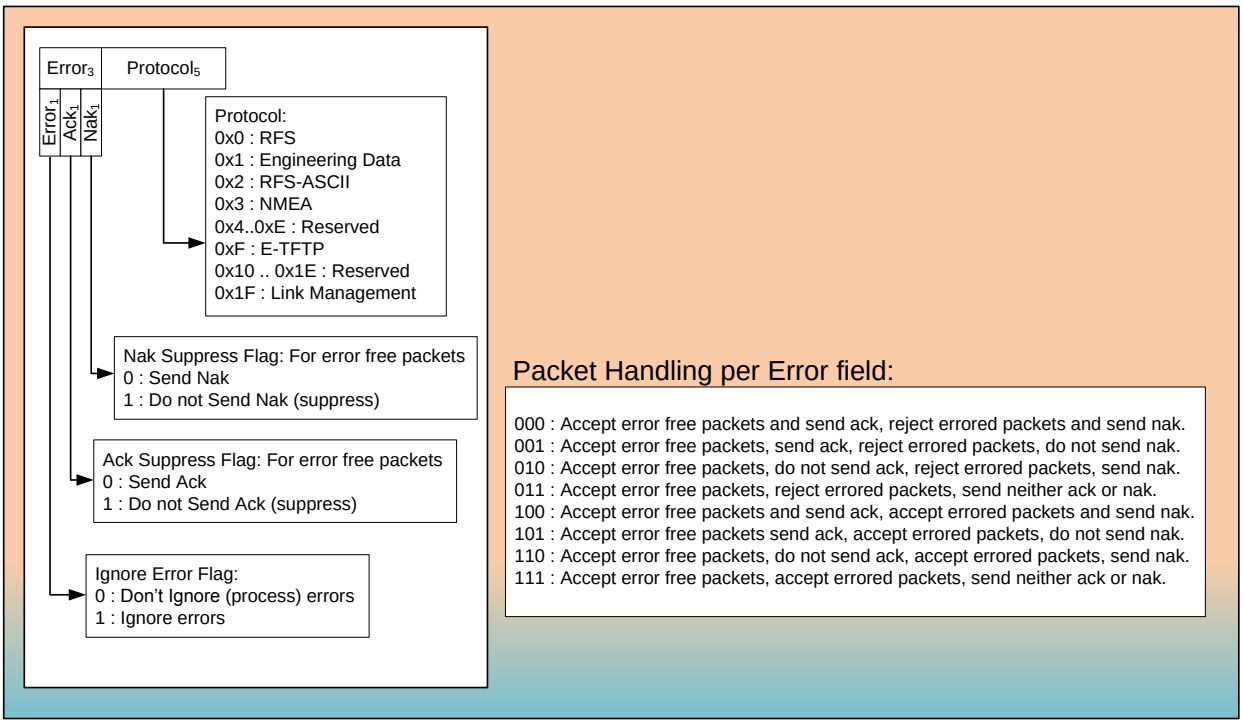

Figure E.4-37 – Error/Protocol Field Encoding

E.3.5 Inter-packet Transmissions

Three characters may be transmitted between packets. They are the SYN, ACK and NAK. ACK and NAK shall be transmitted between packets as described in the error handling section that follows. SYN characters may be transmitted at any time between packets. The lack of a SYN character preceding a packet does not constitute and error condition. SYN characters between packets are discarded by the receiver. A SYN character within the body of a packet constitutes an error.

E.3.6 Error Handling

Error handling is specified on a per packet basis as indicated by the error field Error/Proto byte. Packets are considered to be received correctly if they meet the following conditions:

- The SOH and ETX framing characters are received in the proper order.
- The CRC calculates to 0 over the payload field (excluding Byte-Count field) and CRC field, post DLE removal (and any ACK or NAK characters have been removed).
- The Byte-Count (when non-zero) value correctly identifies the location of the CRC bytes.
- No control characters are received in violation of the requirement to escape them.

Any packet that is received not meeting the correct packet rules shall be considered received in error. The receiver handles the packet according to the encoded value in the error field. The action for each error field encoding is shown in the table below:

Table E.2-Packet Error handling

Error Code (binary)	Packet action
000	Accept error free packets and send ACK, reject errored packets and send NAK. In other words, accept and acknowledge error free packets, reject and negatively acknowledge packets with errors.
001	Accept error free packets send ACK, reject errored packets, and do not send NAK. In other words, accept error free packets and acknowledge them, reject errored packets but do not negatively acknowledge them.
010	Accept error free packets, do not send ACK, reject errored packets, and send NAK. In other words, accept error free packets but do not acknowledge them, reject and negatively acknowledge packets with errors.
011	Accept error free packets, reject errored packets, and send neither ACK nor NAK.
100	Accept error free packets and send ACK, accept errored packets and send NAK. In other words accept all packets, but provide an acknowledge or negative acknowledge as feedback.
101	Accept error free packets send ACK, accept errored packets, and do not send NAK. I other words, accept all packets but only acknowledge the ones without errors.
110	Accept error free packets, do not send ACK, accept errored packets, and send NAK. In other words, accept all packets, but only negatively acknowledge the ones with errors.
111	Accept error free packets, accept errored packets, and send neither ACK nor NAK. In other words quietly accept all packets.

Note that the reception of an ACK or NAK during a packet reception is not an error and allows the channel to be used more efficiently. Systems should accept the ACK or NAK of a previous packet sent while simultaneously accepting a packet from the opposing source.

E.4 Code Samples

E.4.1 CRC16 COMPUTATION

```
typedef unsigned char Byte;
typedef unsigned short int Word16;
typedef unsigned int Word32;
```

```
typedef short int Int16;
typedef int Int43;
/* Sample crc16 computation routines. */
/* A little table to bit reverse bytes easily */
const Byte flip[] =
{
    0x00,0x80,0x40,0xc0,0x20,0xa0,0x60,0xe0,
    0x10,0x90,0x50,0xd0,0x30,0xb0,0x70,0xf0,
    0x08,0x88,0x48,0xc8,0x28,0xa8,0x68,0xe8,
    0x18,0x98,0x58,0xd8,0x38,0xb8,0x78,0xf8,
    0x04,0x84,0x44,0xc4,0x24,0xa4,0x64,0xe4,
    0x14,0x94,0x54,0xd4,0x34,0xb4,0x74,0xf4,
    0x0c,0x8c,0x4c,0xcc,0x2c,0xac,0x6c,0xec,
    0x1c,0x9c,0x5c,0xdc,0x3c,0xbc,0x7c,0xfc,
    0x02,0x82,0x42,0xc2,0x22,0xa2,0x62,0xe2,
    0x12,0x92,0x52,0xd2,0x32,0xb2,0x72,0xf2,
    0x0a,0x8a,0x4a,0xca,0x2a,0xaa,0x6a,0xea,
    0x1a,0x9a,0x5a,0xda,0x3a,0xba,0x7a,0xfa,
    0x06,0x86,0x46,0xc6,0x26,0xa6,0x66,0xe6,
    0x16,0x96,0x56,0xd6,0x36,0xb6,0x76,0xf6,
    0x0e,0x8e,0x4e,0xce,0x2e,0xae,0x6e,0xee,
    0x1e,0x9e,0x5e,0xde,0x3e,0xbe,0x7e,0xfe,
    0x01,0x81,0x41,0xc1,0x21,0xa1,0x61,0xe1,
    0x11,0x91,0x51,0xd1,0x31,0xb1,0x71,0xf1,
    0x09,0x89,0x49,0xc9,0x29,0xa9,0x69,0xe9,
    0x19,0x99,0x59,0xd9,0x39,0xb9,0x79,0xf9,
    0x05,0x85,0x45,0xc5,0x25,0xa5,0x65,0xe5,
    0x15,0x95,0x55,0xd5,0x35,0xb5,0x75,0xf5,
    0x0d,0x8d,0x4d,0xcd,0x2d,0xad,0x6d,0xed,
    0x1d,0x9d,0x5d,0xdd,0x3d,0xbd,0x7d,0xfd,
    0x03,0x83,0x43,0xc3,0x23,0xa3,0x63,0xe3,
    0x13,0x93,0x53,0xd3,0x33,0xb3,0x73,0xf3,
    0x0b,0x8b,0x4b,0xcb,0x2b,0xab,0x6b,0xeb,
    0x1b,0x9b,0x5b,0xdb,0x3b,0xbb,0x7b,0xfb,
    0x07,0x87,0x47,0xc7,0x27,0xa7,0x67,0xe7,
    0x17,0x97,0x57,0xd7,0x37,0xb7,0x77,0xf7,
    0x0f,0x8f,0x4f,0xcf,0x2f,0xaf,0x6f,0xef,
    0x1f,0x9f,0x5f,0xdf,0x3f,0xbf,0x7f,0xff
};


const Byte remainderHigh[256] =
  {
      0x00,0x10,0x20,0x30,0x40,0x50,0x60,0x70,
      0x81,0x91,0xa1,0xb1,0xc1,0xd1,0xe1,0xf1,
      0x12,0x02,0x32,0x22,0x52,0x42,0x72,0x62,
      0x93,0x83,0xb3,0xa3,0xd3,0xc3,0xf3,0xe3,
      0x24,0x34,0x04,0x14,0x64,0x74,0x44,0x54,
      0xa5,0xb5,0x85,0x95,0xe5,0xf5,0xc5,0xd5,
      0x36,0x26,0x16,0x06,0x76,0x66,0x56,0x46,
      0xb7,0xa7,0x97,0x87,0xf7,0xe7,0xd7,0xc7,
      0x48,0x58,0x68,0x78,0x08,0x18,0x28,0x38,
      0xc9,0xd9,0xe9,0xf9,0x89,0x99,0xa9,0xb9,
      0x5a,0x4a,0x7a,0x6a,0x1a,0x0a,0x3a,0x2a,
      0xdb,0xcb,0xfb,0xeb,0x9b,0x8b,0xbb,0xab,
      0x6c,0x7c,0x4c,0x5c,0x2c,0x3c,0x0c,0x1c,
```

```
      0xed,0xfd,0xcd,0xdd,0xad,0xbd,0x8d,0x9d,
      0x7e,0x6e,0x5e,0x4e,0x3e,0x2e,0x1e,0x0e,
      0xff,0xef,0xdf,0xcf,0xbf,0xaf,0x9f,0x8f,
      0x91,0x81,0xb1,0xa1,0xd1,0xc1,0xf1,0xe1,
      0x10,0x00,0x30,0x20,0x50,0x40,0x70,0x60,
      0x83,0x93,0xa3,0xb3,0xc3,0xd3,0xe3,0xf3,
      0x02,0x12,0x22,0x32,0x42,0x52,0x62,0x72,
      0xb5,0xa5,0x95,0x85,0xf5,0xe5,0xd5,0xc5,
      0x34,0x24,0x14,0x04,0x74,0x64,0x54,0x44,
      0xa7,0xb7,0x87,0x97,0xe7,0xf7,0xc7,0xd7,
      0x26,0x36,0x06,0x16,0x66,0x76,0x46,0x56,
      0xd9,0xc9,0xf9,0xe9,0x99,0x89,0xb9,0xa9,
      0x58,0x48,0x78,0x68,0x18,0x08,0x38,0x28,
      0xcb,0xdb,0xeb,0xfb,0x8b,0x9b,0xab,0xbb,
      0x4a,0x5a,0x6a,0x7a,0x0a,0x1a,0x2a,0x3a,
      0xfd,0xed,0xdd,0xcd,0xbd,0xad,0x9d,0x8d,
      0x7c,0x6c,0x5c,0x4c,0x3c,0x2c,0x1c,0x0c,
      0xef,0xff,0xcf,0xdf,0xaf,0xbf,0x8f,0x9f,
      0x6e,0x7e,0x4e,0x5e,0x2e,0x3e,0x0e,0x1e,
  };


const Byte remainderLow[256] =
  {
      0x00,0x21,0x42,0x63,0x84,0xa5,0xc6,0xe7,
      0x08,0x29,0x4a,0x6b,0x8c,0xad,0xce,0xef,
      0x31,0x10,0x73,0x52,0xb5,0x94,0xf7,0xd6,
      0x39,0x18,0x7b,0x5a,0xbd,0x9c,0xff,0xde,
      0x62,0x43,0x20,0x01,0xe6,0xc7,0xa4,0x85,
      0x6a,0x4b,0x28,0x09,0xee,0xcf,0xac,0x8d,
      0x53,0x72,0x11,0x30,0xd7,0xf6,0x95,0xb4,
      0x5b,0x7a,0x19,0x38,0xdf,0xfe,0x9d,0xbc,
      0xc4,0xe5,0x86,0xa7,0x40,0x61,0x02,0x23,
      0xcc,0xed,0x8e,0xaf,0x48,0x69,0x0a,0x2b,
      0xf5,0xd4,0xb7,0x96,0x71,0x50,0x33,0x12,
      0xfd,0xdc,0xbf,0x9e,0x79,0x58,0x3b,0x1a,
      0xa6,0x87,0xe4,0xc5,0x22,0x03,0x60,0x41,
      0xae,0x8f,0xec,0xcd,0x2a,0x0b,0x68,0x49,
      0x97,0xb6,0xd5,0xf4,0x13,0x32,0x51,0x70,
      0x9f,0xbe,0xdd,0xfc,0x1b,0x3a,0x59,0x78,
      0x88,0xa9,0xca,0xeb,0x0c,0x2d,0x4e,0x6f,
      0x80,0xa1,0xc2,0xe3,0x04,0x25,0x46,0x67,
      0xb9,0x98,0xfb,0xda,0x3d,0x1c,0x7f,0x5e,
      0xb1,0x90,0xf3,0xd2,0x35,0x14,0x77,0x56,
      0xea,0xcb,0xa8,0x89,0x6e,0x4f,0x2c,0x0d,
      0xe2,0xc3,0xa0,0x81,0x66,0x47,0x24,0x05,
      0xdb,0xfa,0x99,0xb8,0x5f,0x7e,0x1d,0x3c,
      0xd3,0xf2,0x91,0xb0,0x57,0x76,0x15,0x34,
      0x4c,0x6d,0x0e,0x2f,0xc8,0xe9,0x8a,0xab,
      0x44,0x65,0x06,0x27,0xc0,0xe1,0x82,0xa3,
      0x7d,0x5c,0x3f,0x1e,0xf9,0xd8,0xbb,0x9a,
      0x75,0x54,0x37,0x16,0xf1,0xd0,0xb3,0x92,
      0x2e,0x0f,0x6c,0x4d,0xaa,0x8b,0xe8,0xc9,
      0x26,0x07,0x64,0x45,0xa2,0x83,0xe0,0xc1,
      0x1f,0x3e,0x5d,0x7c,0x9b,0xba,0xd9,0xf8,
      0x17,0x36,0x55,0x74,0x93,0xb2,0xd1,0xf0
```

```
  };


Word16 computeCRC(Byte *buffer,Byte length,Word16 startingValue,Byte doLSB)
{
    Byte   crcH,crcL;
    Byte index;
    crcH = startingValue >> 8;
    crcL = startingValue & 0xff;
    if(doLSB)
    {
        while(length--)
        {
            // index = ((flip[*buffer++]) ^ (crc >> 8));
            index = ((flip[*buffer++]) ^ (crcH));
            //   crc = (crc<<8) ^ mt[index];
            crcH = (crcL) ^ remainderHigh[index];
            crcL = remainderLow[index];
        }
    }
    else
    {
        while(length--)
        {
            //        index = ((*buffer++) ^ (crc >> 8));
            index = (*buffer++) ^ crcH;
            // crc = (crc<<8) ^ mt[index];
            crcH = (crcL) ^ remainderHigh[index];
            crcL = remainderLow[index];
        }
    }
    return((((Word16)crcH)<<8) | (Word16)crcL);
}

#define NOMINAL_CRC_SEED  0xFFFF
// use the MSB version so we don't have to bit flip the CRC value
// when sending, the other end will use the same source code
// to check the crc, so it does not have to match CCITT standards exactly.
/*-------------------------------------------------------------------------*/
Word16 calcCRC(Byte *buffer, Byte length)
/*-------------------------------------------------------------------------*/
{
  return(computeCRC(buffer,length, NOMINAL_CRC_SEED,0));
}

Word16 runningCRC(Byte *buffer,Int32 length, Word16 partialCRC)
{
    if((length > 1000) || (length <= 0))
    {
        return(0xffff);
    }
    return(computeCRC(buffer,length,partialCRC,0));
}//Word16 runningCRC(Byte *buffer,Int32 length, Word16 partialCRC)

void addCRC16(Byte *buffer,int length)
{
```

```
    Word16 crcValue;
    crcValue = calcCRC(buffer,length);
    buffer[length] = (crcValue >> 8);
    buffer[length+1] = (crcValue & 0xff);
}//addcrc16
```

E.4.2 DLE PROTOCOL Sample Software

```
//*****************************************************************************
//*****************************************************************************
// dle protocol
//*****************************************************************************
//*****************************************************************************

// SOH starts a packet
// ETX ends a packet
// ACK acknowledges a packet
// NAK negatively acknowledges a packet, i.e. non SYN received
// with mal formed packet
// SYN used for clock sync
// All chars above prohibited from the body of the packet between SOH and ETX
// If the char does occur the sequence DLE,(char OR 0x80) is sent instead.
// This includes the DLE char itself, which gets sent as DLE, DLE|0x80
// On receive all chars up to SOH are discarded. All chars from SOH to ETX
// are received. Each time DLE occurs in the data the DLE is discarded and
// the next character is restored as (char AND 0x7f).
// If a packet is received well formed (may or may not include a correct CRC
// as an option). The ACK is sent. If a partial packet is received the NAK is
// sent. Retries on packets are up to the application layer.
// Fixed messages can be canned for this application.


#define NUL 0x00   /*  null */
#define SOH 0x01   /*  start of header */
#define STX 0X02   /*  start of text */
#define ETX 0x03   /*  end of text */
#define EOT 0x04   /*  end of transmission */
#define ENQ 0x05   /*  enquiry */
#define ACK 0x06   /*  acknowledge */
#define BEL 0x07   /*  bell or alarm */
#define BS  0x08   /*  backspace */
#define HT  0x09   /*  horizontal tab */
#define LF  0x0A   /*  line feed */
#define VT  0x0B   /*  vertical tab */
#define FF  0x0C   /*  form feed */
#define CR  0x0D   /*  carriage return */
#define SO  0x0E   /*  shift out */
#define SI  0x0F   /*  shift in */
#define DLE 0x10   /*  data link escape */
#define DC1 0x11   /*  device control 1   CTRL_Q */
#define DC2 0x12   /*  device control 2 */
#define DC3 0x13   /*  device control 3   CTRL_S */
#define DC4 0x14   /*  device control 4 */
#define NAK 0x15   /*  negative acknowledge */
#define SYN 0x16   /*  synchronous idle */
```

```
#define MASK_UP  0x80
#define MASK_OFF 0x7f

Byte isControl(Byte value)
{
    // short circuit most characters
    if(value > SYN)
    {
        return(0);
    }
    switch(value)
    {
        case SYN:
        case SOH:
        case DLE:
        case ETX:
        case ACK:
        case NAK:
            return(1);
        default : return(0);
    }//switch
}//Byte isControl(Byte value)



// Convert a non SOH buffer into a SOH buffer
// return the length to transmit
int addSOHFrame(Byte *output, Byte *input, int length)
{
    Byte *origin = output;
    // put on a couple (optional) sync characters
    *output++ = SYN;
    *output++ = SYN;
    // now opening SOH
    *output++ = SOH;
    while(length--)
    {
        if(isControl(*input))
        {
                *output++ = DLE;
                *output++ = *input++ | MASK_UP;
        }//if
        else
        {
                *output++ = *input++;
        }//else
    }//while
    *output++ = ETX;
    return((int)(output - origin));
}//void addSOHFrame(Byte *output, Byte *input, int length)

int removeSOHFrame(Byte *output, Byte *input, int length)
{
    Byte *origin = output;
```

```
    // skip any non SOH (SYN?) at the front
    while(length--,*input++ != SOH)
    {
        if(length <= 0)
        {
            return(0);
        }//if
    }//while
    // now reduce all DLE'd pairs
    while((length-- > 0) && (*input != ETX))
    {
        if(*input == DLE)
        {
            length--;
            input++;
            *output++ = *input++ & MASK_OFF;
        }
        else if(*input == ACK || *input = NAK)
        {
             //process ack/nak
        }
        else
        {
            *output++ = *input++;
        }
    }//while
    // if packet is not well formed
    if(*input != ETX)
    {
        return(0);
    }
    return((int)(output - origin));
}//void removeSOHFrame(Byte *output, Byte *input, int length)
#if 0
// sample receive interrupt handler
Byte packetBuffer[100];
int packetLength=0;
int packetIndex = 0;
int droppedPacketCounter;
void interrupt receiveHandler(void)
{
    Byte value;
    value = readHardware();

    // if packetlength is still 0 packet hasn't been moved yet
    if(packetLength)
    {
        // previous packet hasn't been finished with yet
        droppedPacketCounter++;
        return;
    }// if
    if(packetIndex)
    {
        packetBuffer[packetIndex++] = value;
        // packet is already in progress
        if(value == ETX)
```

```
        {
            // mark the packet as ready to process
            packetLength = packetIndex;
            // get ready for next packet
            packetIndex = 0;
        }//if
    }//if(packetIndex)
    else
    {
        // no packet in progress
        // throw away everything not SOH
        if(value == SOH)
        {
            packetBuffer[packetIndex++] = value;
        }
    }//else if(packetIndex)
}//void interrupt receiveHandler(void)
// send a message NAK
void sendNAK(void)
{
}
// send a message ACK
void sendACK(void)
{
}
// sample main polling loop function
Byte currentPacket[100];
int currentLength;
Byte checkForPacket(void)
{
    if(!packetLength)
    {
        return(0);
    }//if
    // a packet exists
    currentLength = removeSOHFrame(currentPacket,packetBuffer,packetLength);
    if((!currentLength) || (calcCRC(currentPacket,currentLength) != 0))
    {
        sendNAK();
        return(0);
    }//if((!currentLength) || (calcCRC(currentPacket,currentLength) != 0))
    else
    {
        sendACK();
        return(1);
    }//else ((!currentLength) || (calcCRC(currentPacket,currentLength) != 0))
}//Byte checkForPacket(void)


#endif
```

Appendix F Embedded Trivial File Transfer Protocol

REVISION HISTORY:

DATE	DESCRIPTION OF CHANGE	INITIALS
12/15/09	Initial release (Rev 0)	CMNJr
3/30/10	Added additional details, MAPP diagram. Described header record.	CMNJr
5/27/10	Modifications due to CDR information	CMNJr.
7/4/10	Added desired delivery time.	CMNJr.
8/25/11	Included in RFS document, minor re-writes.	CMNJr.

F.1 Document Overview

The purpose of this document is to describe a simple file transfer mechanism for embedded devices. This document does not address the content of files being transmitted. The content of the transmitted files is application dependent. This protocol only provides for a named file transfer of bulk data.

This document does address the physical/data layer between endpoints. The protocol described here assumes an underlying layer that allows the transmission of binary 8 bit values that are collected into a single packet. The framing, and segmentation and/or reassembly of a packet of information are assumed to be performed by a lower layer if needed.

For the different systems, this protocol is carried in one of several framing layers, either the Sparton Standard Asynchronous Packet Protocol (SAPP), the Sparton Multi-Drop Asynchronous Packet Protocol (MAPP), or the MAPP frame structure carried over the other communications interfaces.

F.2 Referenced Documents

1. RFC1350, THE TFTP PROTOCOL (Revision 2), http://www.ietf.org/rfc/rfc1350.txt

F.3 Protocol Description

F.3.1 Definitions

For the purposes of this document the following definitions will be used.

1) Protocol : A protocol is the description of data transmission and its framing and transmission rules combined with a set of rules that describe when and what kinds of data causes a given effect on the system or end point.

2) Packet: One collection of data transmitted as a single unit according to framing rules. A packet does not constitute an entire message.

3) Packet Based Protocol: A protocol that at its most fined grained layer communicates with packets. All communication is accomplished with regular sized aggregate data quantities.

4) Stream Based Protocol: A protocol that at its most fine grained layer communicates with single information data units, bits or bytes, for example, and there is no regular packet boundary or framing.

5) Message: A complete application level communication that may span more than one packet. A complete message may require acknowledgement or other handshaking to be completely transmitted.

6) Temporal Protocol : A protocol that depends on messages being sent in a specific order. Messages sent out of order in a temporal protocol can be meaningless. For example a multiple packet file transfer protocol may require that the packets be sent in order for the transfer to be successful. By contrast a single control packet that turns on or off a certain feature does not require any previous or following message to be interpreted properly.

7) State-full Protocol : A protocol that may be different depending on the state of the system or one of its endpoints. For example a rule that requires that each packet be or not be acknowledged at the link layer may be dependent on a state variable that has persistent state based on a system or endpoint setting.

8) Stateless Protocol : A protocol where any packet or message can come in any order. No packet depends on a previous or future packet to be a valid packet.

9) Command or Control: Messages communicated between endpoints that are designed to configure one of the endpoints or characteristics of the communications systems between endpoints.

10) Status : Messages carried between endpoints designed to convey the status of one of the endpoints or the status of the communications between the endpoints.

11) Data : Messages that carry user information from one endpoint to another.

12) Addressing : Information carried in a packet or message that allows one endpoint to select another endpoint as the destination of the information.

13) Routing : Information either carried in a packet or message, transmitted in a previous message, or retained as state-full information used to describe to an endpoint how to select the address of the subsequent endpoint of control, status or data messages. Routing information is used by an intermediate endpoint to determine the address of the next endpoint in the forwarding of a message.

14) In-Band Control : Use of the nominal form of a message to control or manage a lower layer in the communications system. For example the message acknowledgement scheme used in the Acoustic Interface in this document uses a normal message to transmit acknowledgement. The Virtual Electronic Function Select packet level protocol uses an out-of-band technique (a single special character (ACK) that is exempt from framing rules to acknowledge packets).

15) Out-of-Band Control : Use of special signaling methods not available to the normal message mechanism to control or manage a lower layer in the communications system. For example, the ACOMMS protocol generates a non-data signal pattern as a preamble to indicate the start of a message.

16) Errored Packet: A packet that has been received or partially received that does not meet the framing rules in some way.

17) Invalid Packet : A packet that meets the framing rules for the interface but contains an unknown command or incorrect parameter value.

18) Data Layer ACK/NAK : An acknowledgement (or negative acknowledgement) sent to indicate a received packet without(with) errors.

19) Application Layer ACK/NAK : An acknowledgment (or negative acknowledgement) sent by the application layer to indicate that a valid (invalid) packet or message was received.

F.3.2 Conventions

Unless otherwise specifically stated the following conventions apply to all interfaces in this document:

1) Bit ordering : When data is serially transmitted through a communications channel the least significant bit is transmitted first, this matches standard UART hardware convention.

2) Bit numbering : Bit 0 shall refer to the least significant bit in an 8 bit word (byte), bit 7 shall refer to the most significant bit in an 8 bit word. Bit 15 shall refer to the most significant bit of a 16 bit word. Bit 31 shall refer to the most significant bit of a 32 bit word.

3) Byte : The term byte shall refer to an 8 bit quantity.

4) Byte ordering : As transmitted through a communications channel all quantities that require more than one byte of information shall be transmitted most significant byte first ("big endian").

5) Binary Data : Any 8 bit quantity that can take on any value from 0x00 to 0xFF and does not strictly represent an encoded character set.

6) ASCII or Text Data : Any 8 bit quantity that represents a printable ASCII value 0x20 to 0x7F, plus nominal carriage control characters (0x0D and 0x0A).

7) String data : For the purposes of this document, String Data is defined to be any sequence of ASCII characters terminated by a null (0x00).

8) Ordinal Value : A value that starts at 0 and can be any integer value up to a terminal value. This value represents a selection of some type. Typically carried in a single byte, but may be extended to a larger size. Ordinals are listed in numerical order and may optionally have an "=" sign followed by an integer indicating the actual value if unclear in context.

9) Word32 : Refers to a 32 bit unsigned binary value. If transmitted, sent most significant byte first.

10) Int32 : Refers to a 32 bit signed binary value. If transmitted, sent most significant byte first.

11) Word16 : A 16 bit unsigned quantity. If transmitted, sent most significant byte first.

12) Int16 : A 16 bit signed quantity. If transmitted, sent most significant byte first.

F.3.3 Addressing and Routing

This protocol does not describe addressing and routing as these are dealt with at the physical/link layer.

F.4 Protocol Details

The purpose of this protocol is to send files from and to the embedded device. A file is any arbitrary stream of bytes. All files in the embedded device system are streams of binary bytes. The interpretation of the files, whether it be text or binary is made by the application. The file transfer

mechanism does not examine or alter the file contents in any way during transmission (no “cooking”, as it were).

The Embedded device Trivial File Transfer Protocol (E-TFTP) is closely modeled after the IETF-RFC1350 Trivial File Transfer Protocol as listed in the references. The user should become familiar with this protocol for an understanding of the basic transfer paradigm. The E-TFTP is modified slightly from RFC1350 to accommodate additional requirements in the embedded device system. The slight modification is to allow different packet sizes other than the default 512 as specified in the RFC. In the description below the number of data payload bytes transferred shall be referred to as N. The value N shall be determined by the application/system. Implementers should make N a user selection or configuration item on user interface client software.

F.4.1 Basic Algorithm

The basic algorithm of file transfer to the embedded device is as follows:

1) The embedded device, as the server accepts a write (WRQ) request from the client. The write request contains the name of the file being written and must conform to the expected file names expected by the embedded device.
2) The embedded device acknowledges the WRQ with an E-TFTP ACK packet with a block number of 0.
3) The client sends the first packet of file data containing N bytes of the file, with a block number of 1. (If the file is shorter than N bytes, only one data packet is sent.)
4) The embedded device sends the TFTP ACK packet with the block number that was sent in the data packet.
5) Steps 3 and 4 are repeated until a packet is sent with < N data bytes. (Note if the file length is an integer multiple of N bytes then one additional packet will be sent after the last N byte packet. The one additional packet will have 0 data bytes and will indicate the end of the file.) The embedded device returns an ACK for the final packet. Some devices may send the final ACK to indicate that some internal checksum calculation across the entire file was correct.
6) The file transfer ends.

The basic algorithm of file transfer from the embedded device is as follows:

1) The embedded device, as the server accepts a read (RRQ) request from the client. The read request contains the name of the file being read and must be a file that is known to be present in the embedded device by reading the database.
2) The embedded device acknowledges the RRQ with an E-TFTP ACK packet with a block number of 0.
3) The embedded device sends data packets 1 through N.
4) The client sends the TFTP ACK packet with the block number that was sent in the data packet.
5) Steps 3 and 4 are repeated until a packet is sent with < N data bytes. (Note if the file length is an integer multiple of N bytes then one additional packet will be sent after the last N byte packet. The one additional packet will have 0 data bytes and will indicate the end of the file.)
6) The file transfer ends.

F.4.2 Packet Structure

The packet structure follows the IETF RFC 1350 (Excluding any IP header information).

F.4.3 Op-codes

The Op-codes follow the IETF RFC 1350

Appendix G Abbreviations

The following abbreviations appear in this document

RFS	Remote Function Select
RFS0	Remote Function Select - Revision 0. Features that were part of Revision 0.
RFS1	Remote Function Select - Revision1. Features that were implemented in RFS Revision 1.
RFS2	Remote Function Select – Revision 2. Features that were implemented in RFS Revision 2.
RFS3	Remove Function Select – Revision 3. Features that were implemented in RFS Revision 3.

sparton
NAVIGATION AND EXPLORATION
5612 Johnson Lake Rd • Deleon Springs, FL • 32130
800.824.0682 • 386.985.4631
www.sparton.com/ne • www.thedigitalcompass.com