Gazebo Protocol Specification

Version 0.8.10

Introduction and Statement of Purpose

In many electronics applications there is a need for devices to communicate with each other. Often this is accomplished via a proprietary application layer running on a semi-standard link layer. Often much manual setup is required, e.g. entering device addresses, writing custom drivers, etc. Another problem is that devices are often incompatible with each other and cannot share a physical layer, or require certain physical data lines to be exclusive to one device.

Gazebo aims to correct these problems by defining a protocol that runs on common, existing physical layers, does not require the purchase of any developer license or unique codes, allows device to share a common bus, allows devices to be discovered with manual addressing, and allows devices to be self-documenting.

Gazebo does not require custom hardware nor does it require excessive CPU resources. It runs over common RS485, USB,or TTL interfaces and the software is freely available. Gazebo was designed not to require devices to have any user interface(DIP switch,LCD,Buttons, etc) at all, an important concern for small sensors and the like.

Gazebo has native support for things like units of measure, data types (Including compound data types) remote function calls, nonvolatile configurations, and defines ways to ask questions about devices. Without human intervention the master can enumerate the devices on the bus, discover their capabilities, and issue commands.

Gazebo is general purpose and is not specific to any industry. It was designed to allow devices to expose very generic functionality in a uniform way. Gazebo is not limited to any predefined list of device types.

The core concept in Gazebo is the Parameter. A parameter is a variable that a device exposes to the network. Parameters may support read and write operations, and every parameter has associated metadata that may be requested from the device.

Parameters are not limited to simple variables, as both writes and reads may have side effects, and read requests may carry arguments, making them equivalent to function calls. Gazebo defines a mechanism for asking if a read or write will have side effects.

The concept of Objects is supported in a lightweight manner through Grouping. Parameters may be grouped together, and every group is an instance of a group class. In effect this allows anyone to effectively extend the standard by publishing an API as a group class and giving that class a unique name.
A side effect of this is that we can do away with ."device classes" and the like, because instead of looking for devices of a specific class, we can instead look for devices that implement a specific class.

Gazebo specifies native support for Error responses. While Gazebo was not designed for mission-critical systems, reliability was a large concern in the design of the protocol. Small timing errors in

gazebo data will not affect operation, nor is it possible for a string of bytes from noise to lock up the protocol decoder state. CRCs are used on all packets except for one-byte acknowledgments and Boolean responses, and data is to some degree self-synchronizing due to the use of packet start codes and length fields. In addition to CRCs, runtime validation is performed and attempting to write the wrong number of bytes to a parameter, perform an operation on a nonexistent parameter, or perform an invalid operation will result in an error. In addition, application code may perform additional validation on the data and return a standard error packet if anything is amiss.

For these reasons, it is believed that Gazebo is a viable protocol for use in the fields of lighting control, process automation for non-critical equipment, entertainment, device configuration, scientific monitoring, and general embedded communication.

Table of Contents

About this Document

Definition of the Protocol

The protocol is to be considered as defined by this document and not by any reference implementation.

Assumed Knowledge

It is assumed that the reader is familiar with basic electronics and computer science, and with common standards. This document will attempt not to reference anything that could not be researched with a simple Internet search.

Build Process

This document is edited and stored as an OpenOffice.org ODT file and exported using Writer's export functionality. When in doubt, the ODT version should be considered authoritative.

Conventions Used

All numbers are to be interpreted as decimal unless preceded by '0x' in which case they are hex. Section numbering is not used in light of the modern trend towards e-book reading and searchability. Sections numbers are only used to express numbered lists where the number is important. References to other sections in this document are italicized.

Version Numbers

Major

The major revision number indicates a stable release this version is based on. A major release is made when the current specification is stable. Every Major release will attempt to be backwards compatible with every previous Major release. You should design devices to comply with the most recent major release unless a) the device has user-upgradable firmware, or b) the device does not have the hardware resources to support the current version. In that case a previous major version should be used.

Note that every version before V1.0 should be considered an experimental version, and backward compatibility may not exist with pre-1.0 versions.

Minor

The minor revision number indicates a change that modifies a feature. Features added since the last Major revision should be considered unstable.

Trivial

The final part of the minor revision number indicates changes to the text that do not alter the actual protocol but instead clarify or improve wording.

Must and Should

As used in this document, "must" indicates an absolute requirement without which devices are not considered to be compliant. "should" indicates a requirement that may be ignored if engineering practicalities do not allow it to be implemented, but otherwise should not be ignored and should not be considered unimportant.

Basic Concepts

Master-Slave

The protocol is master-slave, with all transactions being initiated by the master. Here a "Transaction" refers to one complete operation, such as a write or read, and consists of one packet sent by the master and optionally one packet sent in response. This eliminates complex arbitration protocols.

Addressing

Slaves all have a fixed 128 bit UUID. The master may conduct a binary search algorithm to find all devices, and then assign temporary 16 bit addresses to nodes to eliminate the overhead of such a large address space.

This binary search period is the only time when there is a possibility of nodes colliding, and collisions are acceptable because the presence of any data at all is what the master looks for, even if it contains framing errors. This does however depend on the master being able to detect bytes even if framing errors are present. However this appears to be the standard behavior.

UUIDs are not required to conform to any standard and may simply be random numbers. Care should be taken to ensure that UUIDs are generated with sufficient entropy.

Parameters

Each slave device has a set of *parameters* that may be readable and/or writable by the master. Each parameter is associated with a *type* which is similar to standard programming concepts of type.

Parameters also have an *interpretation,* which gives meaning to the value of the parameter, either by specifying a unit of measure o a data format. Interpretations are named in a specified manner with preference given to SI names as outlined in Naming of Interpretations.

Parameters may also have a list of Arguments passed when reading to allow exposing a function. In that case the arguments would all have a name, a type, and an interpretation, allowing for some degree of self documentation even for function calls.

Devices are *not required* to accept write requests and may *optionally* return an error message if a slave attempts to write an invalid datum to a parameter even if it is formatted correctly e.g. writing an out of range value to a numeric variable.

Each parameter has a name, and a device cannot have two parameters with the same name.

Parameters are numbered from 0 to the index of the highest parameter. Parameters must be numbered sequentially.

A devices parameter set should be considered *bound* to it's UUID. If any firmware update changes it's interface, than it should get a new UUID, and is considered a new device.

Grouping

Parameters may be *grouped* together, and every group is an instance of a group class. Groups allow lightweight object-like behavior without impacting slave performance.

Groups are entirely implemented in stored metadata on the slave, and the slave firmware need not have any further understanding of groups. Likewise the master may ignore the group metadata.

Every group instance has a name and a slave may not have two groups with the same name.

Grouped parameters have an associated *Group Role,* such as 'Shuffle' in a group representing a virtual deck of cards.

Anyone may create an *Interface* by publishing an *Interface Specification* defining a compliant group. Such groups should have Class names of the form "<Base64UUID>_<Whatever>". Well known organizations may use their organization name instead of the UUID.

Group Class names may not begin with the asterisk as that is reserved. Groups intended for one-off use(not intended as a public standard) are not required to have UUIDs or organization names.

Allowing unique user-defined group classes allows, for example, a maker of challenge-response tokens to publish a standard for the "kr97bc5Lm7BBnjkiHjr_Token". Now anyone can make a challenge-response token that is compatible with all the existing devices. Someone could make an upgraded version, say, that also serves as a digital signature device. They could implement the digital signature device as a separate group, and have backwards compatibility.

Profiles

Slave devices do not have "device profiles" as a device profile can easily be replaced with a group class. Group classes may be more flexible in that they allow a device to implement multiple inheritance simple by implementing two distinct group classes without resorting to ideas of multiple endpoints.

Multicasting

broadcasting and multicast addressing is an integral part of the protocol, and at any time the master may command a device to join a group or quit a group. Group addresses are taken from the same address space as any other address, and slaves must take the same action when given a multicast command as they would when given a direct request. All slaves must be able to listen to at least 4 groups addresses.

Error Reporting

A slave may respond to any request with an Slave Error Packet.
Slave Error Packets contain a one byte error code, which is to be interpreted as per the table in *Error Codes.*
Slave error codes may also contain an arbitrary null-terminated UTF-8 string of additional information.

Slaves should implement as many error checks as is practical.
Slaves that do not have the resources to return specific errors can return error 0(generic error) in place of any other error message, but this is to be avoided where possible.

Information Broadcasting

Information Broadcasting is a way to send information to a group of slaves, all slaves, or one slave without knowing anything about the slaves an without depending on the numbering of their parameters.

Every information Broadcast contains a Key, which is an 8 character description of the broadcast, such as Year for the number o the current year. The rest of the information broadcast contains up to 240 characters of data.

The purpose of this is to have a standard way of sending data to many different slaves at once, even if they have different numbering for their parameters. Any slave may ignore an information broadcast, and since they are standard packets, you can send them to one slave, a group of slaves, or all slaves because of Gazebo's flexible addressing.

Scalability

Gazebo provides the main protocol, which is already very easy to implement, and the Small Gazebo subset which does not even require devices be able to communicate back. Small Gazebo only uses the reserved address range 0 256-512, and the broadcast address.
In small gazebo, all commands except for writing data to a parameter and reading from a parameter are optional.

Sanity Checking

A central theme in Gazebo is sanity checking. Error messages are provided not only for protocol level errors like writing to an invalid parameter, but also general Application Level errors, such as writing an invalid sequence to a byte array that must conform to a format. Device developers are encouraged to sanity check all incoming data. Not only does this increase reliability but it makes development easier, because programmers can see exactly what commands are invalid and why.

Endianness

All data with the exception of the CRC is always sent little-Endian, because it is the more common transmission.
The CRC is always sent big-Endian.

Physical Layer

The protocol defines several possible physical layers. Each physical layer defines connectors, voltage levels, and power may be carried alongside in all of them. All three layers are intercompatible with very simple adapters.

Gazebo-USB

A point-to-point version of Gazebo tunneled over a virtual COM port connection using the standard USB-CDC class. One advantage of Gazebo-USB is that from the software's point of view it is identical to talking to a RS485 Gazebo device through a serial adapter. The device should present itself as 8-N-1 Serial, and baud rate commands should be ignored as USB serial has no baud rate in the traditional sense.

Gazebo-RS485

8-N-1 Serial data carried over RS485 voltage levels. For compatibility, Devices should accept any input power voltage between 7 and 30 volts but are not required to accept any particular voltage level. Connections, wire type, and power are left to the individual designer.

Gazebo-TTL

8-N-1 Idle high non-inverted data carried over a shared TTL wire pulled high to 5V via no less than 1k via any connector. Should power be carried alongside the voltage should be 5V.

Packet Format

The protocol defines two types of packet, *Standard* and *Short*.
A Standard Packet complies with the below format while a short packet is simply an arbitrary byte sequence that may include ACK, and ASCII 0 to F.

Packets are described in terms of bytes transmitted over an arbitrary medium.

Standard Packets are to be of the form

(In order from first transmitted at the top to last transmitted at the bottom)

[8 bits fixed preamble that must be 0x55]
[8 bit Packet-Type]
[16 bit Address]
[8 bit Length]
[0-240 bytes Data Payload]
[16 bit CRC16]

For a total overhead of 70 bits per packet accounting for start and stop bits.

The exception to this rule is Short Packets such as Slave Presence Response, ACK packets, and other packets which are a single byte or a series of bytes not including the 0x55 byte. A slave device never responds to a packet that does not start with 0x55. Single byte packets are allowed due to the required start code.

Meanings of packet fields:

Preamble

This is simple a fixed 0x55 used to indicate a packet start.

Packet-Type

An index into a list of standard packet-types that is listed in the tables.
A node receiving a packet with a type that it does not understand should ignore it.

Address

A 16 bit destination address. The first 4096 addresses are reserved however any other address may be assigned to a device or used as a group address. Address 1 is reserved for the master and address 0 is reserved for ALL slaves.

Length

The length field is to include the Packet-Type, Address, Length, and CRC but not the preamble.

Therefore the length byte of an empty message will be 6, allowing for 249 bytes of data.

The overhead of a maximal length packet is only 2%, however the overhead of a Parameter read request for short data such as a single float32 is around 4.5 times the length of the data due to the request packet and the overhead involved with the response.

Data Payload

Meaning is specific to the packet-type.

CRC

A cyclic redundancy check used to detect errors. A slave should completely ignore packets with known errors in them. It is the responsibility of the master to retry if necessary. The CRC is the has of the entire rest of the packet EXCLUDING the preamble. The polynomial is to be 0x1C86C, not reversed, with 0 as the initial value and no final XOR. The CRC is to be transmitted high byte first, unlike the rest of Gazebo's data, to comply with the traditional CRC ordering.

Short Packets

Short Packets are used to avoid the overhead of a full packet. A short packet consists of one or more bytes that do not include 0x55.

The major uses of short packets are Slave Presence Responses, which consist of the 0x00 byte, And ACK packets, which consist of the ASCII ACK byte.

Short packets can also represent binary or hexadecimal simply by transferring the actual ASCII character such as '1' or '0' or 'F'

Timing and Delimiting rules

Acknowledgment and Responses

Some packets-type specify that the slave must give a response.

This response must be given within 5 byte times or 10% of the time it took to send the message, however slaves **should** respond within one byte time for maximum throughput . The same rules apply for data responses to parameter requests and the like.

The exception to this rule is Nonvolatile Memory. When the master sends a command to write to NV memory it should be prepared to wait up to 100ms+25ms per byte that needs to be written. Devices of course should still carry out this operation as fast as possible.

Delimiting

The beginning of a packet is to be detected by an [0x55](ASCII 'U') appearing when the receiver is idle. 0x55 is chosen because it is common to do auto baud-detection, and also because no byte at a baud rate slower than you receive on can erroneously appear as an 0x55.

Receivers are to go idle after a full packet has been received, OR after the greater of 10ms or 1 byte-time at the current speed have passed without data. This feature means that multiple baud rates can coexist on one bus without the use of auto-baud detection.

The master has only to wait more than 1 byte times at the new rate or 10ms before sending data after changing baud rates. No inter-frame spacing is required between frames of the same baud rate, because slaves detect end of packet based on length.

The master **should** check to make sure no noise has been on the bus in the last 1 byte-time or 10ms before transmitting for maximum reliability and throughput.

When any error is detected, the master **should** wait at least 2 byte-times or 25ms before transmitting again.

No packet may include more that 1 byte-time in between bytes, nor may more than 15% of the total content of any packet be blank space between bytes.

Initiation

All communication is to be initiated by the master.
Packets going **TO** the master must have **1** as the LSB of the packet type.
Packets going **FROM** the master must be **0** as the LSB.
This lets a slave decide to handle a packet or not within two bytes.

Slave Properties

Every slave is to have a fixed 16 byte random identifier that is only used until the master assigns the slave a 2 byte Network ID. The random identifier may be programmed at the factory or generated and saved at first power-up. The 2 byte network ID is to be initialized to 0 after a reset. The master can then use the UUIDS to run a binary search and assign devices specific temporary network IDs.

Addressing

Addresses are simple 16 bit numbers with no semantics whatsoever except for the reserved address range 0-4096. Address 0 is the broadcast address, address 1 is the master, all other addresses can be freely assigned.

Any slave must respond on its own address, the broadcast address, and any of its four group addresses exactly uniformly. A slave should respond to a broadcast packet exactly as it would a packet addressed only to it. Slaves must default to address 0 at power-up.

Addresses 256 through 512 are reserved for Small Gazebo, the restricted subset of Gazebo.

Supported Baud Rates

All devices by default must boot up to a rate of 4800 baud. All devices must support at least 9800 baud. Higher baud rates may be selected by broadcast or on a per-device basis. Baud rates are changed via a command that represents the rate as a uint8 interpreted as an index into the following table:

The baud rate table for all devices is:

Rate Code	Rate in Baud
0	100
1	300
2	1200
3	2400
4	4800
5	9600
6	38400
7	57600
8	115,200
9	250,000
10	500,000
11	1,000,000

The choice of baud rates was designed to allow easy operation from modern crystals which are often (2^n)*(1000^m) hertz. The inclusion of "traditional" baud rates is only for compatibility with legacy hardware and serial ports. Baud rates below 115.2k use only traditional values because the error introduced from the combination of a modern crystal and a low "traditional" rate is likely to be low due to the high divider value used relative to any rounding error.

The rates also have the property that no two rates are separated by less than a factor of two.

Valid Packet Types

In this section the packet types are listed by type code and sorted into two categories: Packets sent only by the master and packets sent only by the slave. Packets sent by the master have even packet type codes and packets sent by the slave have odd packet type codes.

Headings are preceded by their Packet Type codes, specified in decimal.

Master May Send These

0: Slave Presence Detect Request

(Required for all slaves to handle)

Payload

[16 bytes UUID] [2 Bytes network address] [16 Bytes UUID bitmask] [2 Bytes Network Address bit mask]

Description

Slaves are required to respond with a slave presence declaration packet if the bit pattern matches theirs. A zero in any bit mask bit designates a "Don't Care' bit allowing for binary search trees over all nodes or only those nodes that have not been assigned network IDs yet, avoiding use of recursion or excessive ram in the enumeration routine.

By requesting only nodes that have not been assigned a network address(by setting 0 as the required address) You always get a new slave each time you run the search tree and can loop through slaves easily without using recursion of other complex algorithms(at least complex by embedded standards)

2:Slave Address Set Command

(Required for all slaves to handle)

Payload

[16 bytes UUID] [2 Bytes New network address]

Description

Instructs the slave with a UUID that matches the UUID in the packet to set its temporary network address to the 2 byte address sent. The slave must respond with a slave ACK packet.

4:Slave Information Request

(Required for all slaves to handle)

Payload

[1 byte page number]

Description

Slaves must return a string of data up to 248 bytes long with starting position is S=(248*Page Number) of the Slave Description String.
The slave description string must be null terminated and the last page should not contain data after the null.
The format is UTF-8. For example:
Page 1 would by bytes 0-247
Page 2 would be bytes 248-496
etc. until the last page which may be less than 248 bytes long and must have a null terminator as the last byte.

6:Parameter Information Request

(Required for all slaves to handle)

Payload

[1 byte parameter number]

Description

Slave must respond with a Data Response containing a null-terminated string satisfying a format or with the appropriate error if the parameter number is out of range.

8:Parameter Value Read

(Required for all **relevant** slaves to handle)

Payload

[1 byte parameter number] [optional 1-239 byte arguments]

Description

Slave must respond with the current value of the requested parameter or with the appropriate error if the Parameter Number is out of range, or the wrong amount of argument data was passed.
The return value must be encapsulated as a slave data response, or as a short packet.
If the parameter is a function, the appropriate arguments must be passed. All arguments should be concatenated in the order they appear in the arguments list in the Parameter Descriptor.

10: Parameter Write

(Required for all **relevant** slaves to handle)

Payload

[1 Byte parameter number] [1-239 bytes Data]

Description

Command to write Data to the selected Parameter.
Slave must respond with a Slave Acknowledge message, or with the appropriate error if the Parameter Number is out of range, the Data to be written is the wrong length or otherwise incorrect, or if any other error occurs..

12: Parameter Write Without Acknowledge

(Required for all **relevant** slaves to handle)

Payload

[1 Byte parameter number] [1-239 bytes Data]

Description

Command to write Data to the selected Parameter without requesting an acknowledgment.
Slaves must NOT respond with acknowledgment. In case of errors slaves must still remain silent.

14: Baud Rate Set Command

(Required for all slaves to handle)

Payload

[1 byte index into the baud rate table in *Supported Baud Rates*]

Description

Slaves must set their transmit and receive baud rates to this value. Slaves must respond with an ACK packet or with the appropriate error is the baud rate is out of range or any other error occurs. The ACK packet should be sent at the current baud rate, not the new one.

16: Information Broadcast Message:

(Slaves may ignore this message)

Payload

[8 bytes broadcast identifier] [0-232 Bytes Payload Data]

Description

A standard way for the master to broadcast short pieces of information, such as the current time, data about the master, cryptographic nonces, or other information that a large number of nodes would be interested in.

Information that is global to the network such as the current time should only be sent using an Information Broadcast Message. Slaves must never respond to an information broadcast message.

Even though this is called a "Broadcast", slaves must ignore Information Broadcast Messages not addressed to them, a group they are a member of, or the global broadcast address. This is to ensure that it is possible to broadcast information to only a subset of slaves.

Non-standard information broadcast keys(those not defined in this standard in *Standard Information Broadcast Keys*) must not use all capital names.

18: Group Join Command

(Required for all slaves to handle)

Payload

[2 bytes new group address]

Description

Command a slave to ALSO listen on a group address
Slaves must respond with a Slave Acknowledge message.
If the slave is already a member of that group, return the same acknowledge message.
If there is no more space to store group addresses, return an error.

20: Group Quit Message

(Required for all slaves to handle)

Payload

[2 bytes group to quit message]

Description

Commands a device to quit the specified group. If the specified group is 0, quit all groups.
Slave must respond with a Slave Acknowledge message.
If the slave is not a member of that group, return the acknowledge anyway.

22: Save Parameters to Nonvolatile Memory

(Only slaves with nonvolatile parameters must implement this)

Payload

[7 bytes comprising the word CONFIRM in uppercase ASCII]

Description

This will cause the slave to save all parameters that are marked with the 'n' flag to permanent storage.
The current value will become the new default at power up. Slaves must respond with acknowledgment or the appropriate error.

This allows for a standardized way to implement configurable devices such as electronic speed controllers and motion detector sensitivity thresholds. Master software should require confirmation such as "Are you sure you want to make the current settings the new default?" before continuing.

As mentioned in *Timing and Delimiting rules*, when the master sends a command to write to NV memory it should be prepared to wait up to 100ms+25ms per byte that needs to be written.

111-127: Device Specific Commands

This range of packet types has been reserved for device specific messages.
Messages going to the master should still be odd and messages from the master should still be even.

Slave may send these

1: Slave Presence Declaration

(short packet)

Payload

The entire packet is just 1 0x00 byte set in response to a presence detect.

3: Slave Acknowledge Message

(short packet)

Payload

[the entire packet is 1 ASCII ACK]

Description

Sent in response to any successful command that requires acknowledgment

5: Slave Error Message

Payload

[1 byte index into error table] [0-239 bytes UTF-8 Description]

Description

A generic error response that may contain an extended plaintext description.

7: Slave Data Response

Payload

[0-240 Bytes of data to be interpreted by the master based on context]

Description

Sent in response to any request for data by the master at all, Except in cases of error.

111-127: Device Specific Commands

This range of packet types has been reserved for device specific messages.
Messages going to the master should still be odd and messages from the master should still be even.

Error Codes

The valid error codes are:

0: Error

Generic Error Occurred. A device may send error 0 instead of any other error if there is not enough flash to properly implement specific errors. If used to indicate a Application Level error, there should be extra information attatched.

1: Nonexistent Parameter

A read or write was attempted to something not there.
i.e. the parameter index was higher than that of the last parameter.

2: Bad Input

Invalid data was sent.
e.g. Attempting to write an invalid value to a byte array that must conform to a specific interpretation.

3: Insufficient Power

The requested operation would exceed available electrical power.
e.g. Attempting to trigger a solenoid coil when only the limited power from the network bus is available.

4: Not Available in this Mode

The slave is not currently configured for the requested operations.
e.g. Attempting to read an ADC value when the ADC is disabled.

5: Unsupported Action

The parameter selected does not support this action.
e.g. attempting to read from a write-only parameter or vice versa.

6: Critical Internal Error

The slave has encountered an error, such as a watchdog interrupt, and is currently offline.

7: Too much data sent in write request

The write request tried to write more data than fits in the selected variable.

8: Too short data sent in write request

The write request tried to write less data than would fit in the selected variable.

9:Too Many Groups

There was a request to join a group but there is no more space to store group addresses. Quit at least one group and retry.

10: Expected Less Data in Read Request

You made a read request with a longer string of arguments than expected. Can also indicate that 0 bytes of arguments were expected.

11: Expected More Data in Read Request

You made a read request to a parameter that takes arguments but did not pass enough data.

Slave Description String

This is the string that describes a slave and may be up to (248*256)-1 bytes long.
It must be of the form:
(Without enclosing Square Brackets or line breaks)

[Gazebo Major Version, Product Name, Device Firmware Version, flags, Number of parameters,maximum baud rate,description,key-value object]

Any field may be empty except number of parameters and maximum baud rate.
Numbers must be decimal numbers without thousands separators.

The meanings of the fields are as follows.

Gazebo Major Version

The major part of the version. If the devices was built to comply with a non integer(experimental version), the full number should be prefixed by an x.

Product Name

The product name identifies the type of device. The individual device name allows that specific device to be given a name.

Device Firmware Version

An arbitrary short string which is specific to the device.

Flags

A lit of single character flags. This field reserved for future use as no flags are currently defined.

Number of parameters

What it says on the tin, in decimal of course.

Maximum Baud Rate

A decimal or hexadecimal prefaced with 0x number in units of baud. ignored for USB only devices.

Description

An arbitrary mUTF-8 description.

Key-Value object

The key-value object is a simple JSON object that may contain the following:

Listeners:

An associative array(Object in JSON terminology) of Listener Description Strings(Having an identical format to a Parameter Description String) indexed by the Broadcast Key being listened to.

This lets the master know what broadcast keys it will listen to, and the Listener Description String allow the master to know what format the data is expected in, and what its meaning is.

SListeners

A list of strings, each one being the name of a standard information broadcast key(as listed in *Standard Information Broadcast Identifiers*) that the device implements.

Parameter Description String

The parameter description string is a CSV string enclosed by square brackets of the following form:

(Without Enclosing Square Brackets)
[Parameter Name, Return Type,Interpretation,[[Name;Type;Interpretation];...],flags,group role,group name,group class,description,key-value object]

Whitespace will be interpreted literally.

Parameter Name

The name of the parameter. May only contain a-z A-Z and the underscore. Two parameters may not share a name. Names are case sensitive but names differing only in case should not be used to avoid confusion.

Type

Equivalent to concepts of type in programming languages, this field fully specifies the range of lengths the parameter may take on. The type is expressed as a string, optionally followed by either [len] or [lenmin:lenmax] to denote an array. Multidimensional arrays are allowed and considered arrays of arrays.

Interpretation

Interpretation is used to give meaning to the data, and could be as simple as "milliKelvin". Interpretation should only be used for the units and not for specifics such as "front room temp" which is better suited to the name.

Interpretation should be thought of as a "subtype". Good examples are "Score", "Volts", "meters/second", or "micrometers". Bad examples are "FirstSwitch", "LivingRoomLightLevel" or anything else device or application specific. See Naming of Interpretations for more details.

When naming interpretations for custom formats, you can prefix them with a base64 UUID as described before, or you can simply place the variable inside a group representing your interface.

Should the parameter be a tuple, each element of the tuple should be given its own interpretation and interpretations should be separated by semicolons. example, a (float32;float32) tuple with interpretation (Kelvin;Relative Humidity)

[[Name,Type,Interpretation]...]

This entire field may be blank. If it is not, than it is a list of parameters that must be passed when "calling" i.e. reading from the parameter. Bracketed 3-tuples of name,type, and interpretation. Only the last parameter may be a variable array.

Flags

This field is a set of one character flags describing the parameter's properties. The list of valid flags is described in *Valid Flags*.

Group Role

This describes the *Role* that an individual parameter plays in a group. An example would be `Derivative` for a variable to control the derivative gain in a group representing a PID control loop. May be blank for ungrouped parameters. A group may not have multiple variables playing the same role.

Group Name

This is the name of the group, if any, that the variable is a part of. A group is analogous to an object instance. A device may not have two groups with the same name. May be blank for ungrouped parameters.

Group Class

The class of group that the parameter is in. All parameters with the same group name must have the same group class. Anyone can define a group class, so to avoid conflict between unrelated manufactures the company or individual name may be prepended to the group name, or alternately a UUID in base64 may be prepended. May be blank for ungrouped parameters.
Group classes beginning with an asterisk "*" are reserved for "official" standard group classes.

Description

A plain text description, in quotes, with all quotes escaped by a backslash and newlines converted to \n. May be UTF-8 and may be formatted as reStructuredText.

Key-Value Object

A JSON file with additional attributes may be placed here. May be blank. All additional attribute names beginning with an asterisk "*" are reserved for "official" attributes. Official Additional Attributes may be found in `Additional Attributes`.

Valid Flags

Flags may appear in any order and are case sensitive.

0-9= Group Class Specific

A group class can define special meaning for numerals appearing in its members' parameters' metadata. You will find these meanings in the Interface Documentation provided by the party that specified the interface.

r=readable

The parameter may be read by Master

w=writable

The master may write here

i=reads are idempotent

Any number of consecutive reads will return the same data unless something external changes it. However the act of reading itself may not cause changes. For parameters that take argumets to thier read operations, idempotency only applies where two reads have the same arguments.

I=writes are idempotent

N+1 writes of the same data have the same effect as N writes. Does not imply that write A followed by write B where A and B have different data is equivalent to write B alone.

b=this is an interface to a item-wise FIFO buffer

The type of the parameter must be an variable array, with size 0 to N. Reading from this an array containing the oldest N items from a queue. If the item is variable sized, N will always be 1.

Arrays of arrays are allowed, with each inner array in the outermost array being considered one item. Data items written here are entered into the queue. The same rules apply.

Older items will always be first in the returned array. If the queue is empty a message with no payload will be returned.

Buffered types should not be both readable and writable.

B=message-wise FIFO buffer

This is an interface to a FIFO buffer, reads will always return the oldest 1 element. The type may be a variable array but operations should still be considered as returning 1 "message" of variable length, unlike an item-wise buffer, which returns the top N items of fixed size. The same rules apply for writing. If the queue is empty an empty packet will be returned.

Buffered types should not be both readable and writable.
Buffered types usually should ot take any arguments.

s=side effects to reading here

Indicates that reading the parameter will have an effect such as popping the last item in a queue.

f=function return

Indicates that the parameter should be considered a return value for parameters passed into other parameters of the group. A read will trigger execution of the function

n=nonvolatile

Anything written here will be saved to nonvolatile memory when the Save Parameters to Nonvolatile Memory command is issued. It should be considered acceptable for the device to take more time than normal to read or write these.

A slave must not commit this parameter to nonvolatile memory under any circumstances unless the Save Parameters to Nonvolatile Memory command is issued, to allow frequent updates without fear of undue wear on the storage devices.

Slaves should support at least ten thousand erase cycles.

m=mirror

This register has functionality overlapping one or more other registers. For example, an ADC that has data available as a float or as an int.

c=command

A write to this should be considered an explicit command rather than data storage or setting I/O, and the Interpretation should give a name for the command type.

!= Important

The voltage control of a lab power supply would be one example. This flag indicates that extreme care must be used when writing and possibly even when reading. Bootloaders, LED current limiters, things that could cause damage to the node or damage to connected equipment all fall under this category.

Master side software may choose to require extra confirmation before changing these parameters. A standard message might be:

```
"This parameter has been marked as Important. An incorrect value in this parameter may cause hardware damage. Are you sure you want to continue?"
```

Data Types

The standard data data types are:

Basic Types

As previously mentioned, all data is little-endian.

float32

IEEE Single Precision floating point value.

uint8

Unsigned 8 bit integer.

uint16

Signed 16 bit integer.

uint32

Unsigned 32 bit integer.

int8

Signed 8 bit integer.

int16

Signed 16 bit integer.

int32

Unsigned 32 bit integer.

enum{FirstPossibleValue|SecondPossibleValue|etc}

An enumeration similar to C enumerations. Between the brackets can be any number for 0 to 255 of pipe separated identifiers which may not contain spaces or commas or brackets. This should be considered exactly the same as a uint8 with additional semantics attached. Keep in mind parameter description strings are limited to 248 bytes total.

UTF-8

Alias of `uint8`, used to indicate that this should be interpreted as a string. Null terminators should not be sent.

Composite Types

Arrays

An array type of any Base Type is valid, following the convention type[count] or type[min:max] for variable lengths.

An example would be a uint8[10] array. Every write must write every index and every read will return the entire array.

A write to a float[0:50] array may contain 0 to 50 floats, and a read to a float[0:50] array will return 0 to 50 results depending on how many items are stored there currently.

Multidimensional arrays are considered arrays of arrays, and only the last element may be variable. the syntax for a multidimensional array is basetype[a][b], representing b arrays of size a. any number of dimensions is allowed. uint8[10][0:10] would be valid whereas uint8[1:5][10] would not.

The actual transmission format for a multidimensional array is to send each element of the first array, then each element of the second array, and so on. We essentially treat the array as a basetype and send an array of it, which may further be considered yet another type of array and repeated. For an example of array serialization:[[1,2] , [3,4]]would serialize to [1,2,3,4].

Tuples:

A tuple may contain any basic type or array. Tuple elements are separated by semicolons and may not be other tuples. Tuples are serialized by concatenation in the same way as read arguments. Just like arguments, only the last element in a tuple may be of variable size. Tuples do not have any special enclosing syntax and are simple semicolon separated lists.
An example of a valid tuple: "uint8;uint8;float32;UTF-8[0:32]"

Each field of the tuple should be given its own name via the *tuple attribute in *Additional Attributes*. Gazebo does not support 1-tuples.

Special Types

These may return 1-Byte Short Packets.

bool

When read from, returns a 1-byte packet consisting of ASCII 0 or 1.
Writes as a uint8 with literal 1 or 0 not ASCII 1 or 0 values.

hex

When read from, returns a hex digit represented as 1 ASCII byte
Writes as a uint8 from 0 to 15.

Additional Attributes

These may be applied to expose more information about the parameter.

*qlen

Applied to a queue indicates length in bytes

*maxpoll

A string that indicates the maximum polling rate that will result in meaningful data as opposed to repeats of old data. The string must be a number followed by by Hz.

*max

A number indicating the maximum value that a numeric parameter may be set to

*min

A number indicating the minimum value a numeric parameter may be set to.

*fields

A string containing a comma separated list of names, one for each field in the tuple.
example "*fields"="temperature;humidity"

Naming of Interpretations

To increase compatibility and decrease conflict, the following rules should be used when naming Interpretations

Reference should be made to the SI brochure. Units with official SI symbols should be abbreviated unless the official SI symbol is a character that ASCII does not encode in which case they should be spelled out in full.

Non-SI units should never be abbreviated except as listed below. SI capitalization rules should be followed.

In addition to the SI units, the liter may be used with abbreviation L and the degree may be used with abbreviation deg, Any other unit used should be spelled in full.

SI prefixes that are not powers of 1000(H,h,D,d) should be avoided. SI prefixes should *never* be used to refer to the powers of two(e.g. Kilo used to mean 1024), Instead the binary prefixes(Ki,Mi,Gi,Ti,etc) should be used.

Various non-metric customary units such as ounces, pounds, and feet should be avoided when possible.

Compound units should always be created with the forward slash for division and caret to denote exponentiation e.g. `m/s^2` or `kg/m^3`.

Interpretation may also be used to define custom types. The name "unit" was purposefully avoided to allow complex named data structures. An example of this would be "WaveForm" for a DDS function generator, or "8/16kAudioStream" for a buffered 8 bit audio stream.

To avoid conflict, A UUID or a well-known company name should be prepended to custom units.

Names beginning with any punctuation mark are reserved for "official" and standardized Interpretations.

The unit "counts" should be used to count events.

Examples

"kg" Kilograms
"cm^3/s" Cubic centimeters per second
"km/H"
"EmptyDelimitedOGGFrame"
"HexColor"
"RGB"
"RGB-A?"(RGB if 3 bytes, RGBA if four)

Reliability Concerns

Gazebo was never designed to be used in life safety, military, aerospace, medical, or similar applications.
While Gazebo was designed to be highly reliable, implementers must use the protocol entirely at their own risk.

Multiple Baud Rates

The use of multiple baud rates on the bus could cause the device running at the lower baud rate to see an 0x55 when none was sent. Should random line noise or higher bad rate transmissions cause a device to see a valid packet type, preamble, address, data length, and data, undefined operation may result. This is unlikely to occur due to multiple layers of integrity checking, however only a single baud rate should be used if reliability is important.

This will not cause problems when commanding devices individually to change to a higher baud rate if you do not send any commands at the new higher baud rate until all devices have changed to it, because it is not possible for any other data sent at a lower baud rate to appear as an 0x55(which contains the maximum number of transitions) at a higher rate if the higher rate is at least twice the lower rate.

This will also not cause any problems commanding a device to change to the next lower baud rate as long as you wait for at least 3*one byte time at the slowest rate, or 10ms, whichever is higher, as the baud rate set command is 8 bytes long, and 8 bytes will appear as at most 4 bytes at any lower baud rate that is less than half the higher rate. When the acknowledgememt arrives, it will appear as at most 1 additional byte. Therefore the slave at the lower baud rate will see at most 5 bytes, which is not enough for a valid long packet, and therefore will be seen as random line noise and ignored.

Acknowledgment

The use of a one-byte acknowledgment sequence creates the problem that random line noise could coincidentally contain the ACK character and coincidentally appear within the timeout just after the master sends a Write Request, causing the master to think the command was successful even if it was not. Even though the chances of this occurring are low, very important write operations like bootloaders should use some form of readback validation rather than depending on the built in acknowledgment features.

Modbus Compatibility

The use of modbus devices on the same bus is possible if 0x00, 0x55, ASCII ACK, and ASCII zero through uppercase F are not used as modbus addresses. However, modbus transmissions may create similar situations as described in *Reliability Concerns:Multiple Baud Rates.*

Labeling

Ports that are intended to act as masters can be labeled either with `MASTER` or `TO SLAVES`.
Ports intended as slaves can be labeled with `SLAVE` or `TO MASTER`
Switches should be labeled `MASTER|SLAVE` or `ACT AS MASTER|ACT AS SLAVE`
Bidirectional ports can be unlabeled or labeled `MASTER/SLAVE SWITCHABLE`
Any port may be unlabeled where space or cost does not permit labeling.
Devices should be marked with their maximum power consumption.

Naming Conventions

- Devices with physical I/O pins or connection should organize their pins into ports of 8.
- Every port should be assigned a letter as the common MCU convention.
- Nonexistent pins within a port are allowed to allow 1 to 1 mapping of MCU pins to logical ports.

Device Documentation Conventions

Specifying Required Bandwidth

To specify the amount of bandwidth required by the feature, one should specify the number of actual bits transferred after accounting for start and stop bits, and network overhead bytes. One should also always specify at what polling rate the data is valid for. For example, a light switch might say:

On/Off switch function:
Bandwidth: 400 Baud@5Hz polling

Identifying Devices

When there is a need to present an identifier for a device, Base64 with the padding characters removed should be used rather than the traditional UUID notation.

Design Patterns and Solutions

These are suggestions, and not requirements for compliant devices.

Partial Arrays and Data Segmentation

Use a Position parameter and a Data parameter. When you read or write the Data, you actually get the 0:248 bytes after the pointer position. For write-only arrays you can also use a byte array where the first byte is the index. Or you can use a read with arguments interface.

User Interfaces

Avoid directly reading the state of momentary switches. Instead, you can define a rolling count of button presses, or you can define a buffer of bytes for a keyboard. When you read it you get all keys pressed since you last read from it. For buttons that require accurate timing you can use a buffer of timestamps recording the milliseconds part of the current time at each press. Synchronize all devices using Information Broadcast Messages. For switches, knobs, toggles, and sliders, you should just directly read their state.

Measurements that require time

For measurements that require time and/or energy to take, such as an ultrasonic distance reading, you can either take the measurement continuously and return the latest when polled, or have a separate begin measurement parameter.

Energy Saving

Things that require energy should always be able to be turned off in software.
Attempting to read from a disabled sensor should always return error 4 Not Available in This Mode.

General Purpose I/O modules

Each "Special Function" should have it's own group. Each special function group should have a group role called ENABLE. When ENABLE is 0, we can use the pin as a normal digital I/O. Otherwise the special function takes control. Each pin should be able to be accessed as an individual parameter. If the device is intended to be used for user interfaces, several debounced rising/falling edge counters should be implemented.

Group and Multicast addressing

Normally group addressing would require devices to be exactly the same or at least share the exact same parameter array layout. However, there is a way around that. Information Broadcast Messages contain an 8 character identifier and as such make no reference to the layout of parameters on the slave. Whenever a slave is to receive data that is likely to be shared among many slaves, such as the current time or global power save settings, the slave should accept that data as an information broadcast request, and not as a parameter write.

Button Presses, Motion Sensors, and other Short Events

Since the slaves cannot report without being asked, simply polling the actual sensor state would leave a large possibility of missed events. Instead, maintain a rolling event count, and simply check if it has changed since the last time you polled it. Alternately, maintain a queue and place an entry in for every event. This requires more memory but allows the events to have attached data.

Procedure Calls

While function calls are best implemented as reads with arguments, it is suggested that function calls that do not return a value
be implemented as write operations to give the master the option of calling the procedure without waiting for acknowledgement.

Glossary

Gazebo **Parameter: A variable, function, or property exposed to the master by a slave device**

mUTF-8: Modified UTF-8. This variant allows compatibility with null terminated strings.

8-N-1: Asynchronous serial with 8 data bits, 1 start bit, and 1 stop bit.