|  |  |
| --- | --- |
|  | JETDRIVE Network Specification  Revision 2.1  Dynojet Research, 2020  *Rev 2.0, 3/11/2020 – Initial Proposal*  *Rev 2.1, 4/21202 – Clarification of terms and resolution of pitfalls*  *Rev 2.1.1, 5/1/2020 – Changed pong echo offset from -9 to -5* |
| **Abstract**  **How It Works**  **Providers**  **Clients**  **Transport**  **KLHDV** | Tuning an engine involves measuring air/fuel mixtures, exhaust gas temperature, power output, modifying engine controller calibrations, etc. There is no single tool or platform that performs all of these functions, and thus, users are left to piece together a toolchain from an array of vendors. Currently, no standard exists to unify these tools into a cohesive tuning pipeline and users must combine their data by hand using spreadsheet software, best guesses, and cell-by-cell table adjustments. We aim to create an industry standard that allows tooling from any participating vendor to work together to provide better tunes, faster.  JETDRIVE defines a network topology and communication standard for sharing raw and calculated sensor samples between differing software and hardware platforms. The topology is much akin to a CAN network – all participating nodes receive all messages, but only use the data relevant to the particular node. The topology *differs* from a CAN network in that it uses Ethernet, which can be wired or wireless, and in addition, can take advantage of an inexpensive, ubiquitous and full-duplex communication layer present in most tuning centers.  The JETDRIVE network does not have a central node or server, but rather acts as a sensor data mesh. The network consists of both providers and clients (nodes may be one or the other, or both at the same time). Providers stream out channel data; clients read in channel data.  Providers are defined as nodes that stream channel data to the network. Providers don’t have to respond to any messages except the Ping message, which facilitates time synchronization. Providers should transmit a ChannelInfo message every 30 seconds or whenever the set of channels being transmitted changes, allowing new and existing nodes to correctly decode and list channel values and names.  Clients are defined as nodes that receive channel data from the network. Clients do not need to transmit any messages, but transmission of a Ping message to any unsynchronized Provider is *recommended* for clock synchronization. In addition, if the client will transmit any messages, it should respond to the Ping command for address collision prevention.    JETDRIVE networks shall use a UDP multicast group (group address and port to be determined). All messages shall be KLHDV (Key-Length-Host-Destination-Value) encoded, and messages that span multiple packets (those larger than a network’s MTU) shall be separately encoded in their own KLHDV wrapper. (Sidenote: We considered other data wrappers, such as Google’s protocol buffers, but it’s difficult to deterministically predict the size of a message, which does not lend itself well to the UDP transport protocol).  KLHDV, or Key-Length-Host-Destination-Value messages, are a lightweight wrapper for arbitrary data. For JETDRIVE, the Key is simply the message type or command – ChannelValues, ChannelInfo, Ping, etc. The Length field is the number of bytes in the Value field (not including the host/destination fields). The host field is the ID of the message’s sender, and the destination is the host ID of the message’s destination. The value field is an array of bytes, to be decoded depending on message type (key).  The special host value, ALL\_HOSTS, (0xFFFF) shall be used ONLY in the destination field. Nodes should ignore all messages where the host ID (source) is ALL\_HOSTS.  The following table illustrates the KLHDV format:   |  |  |  | | --- | --- | --- | | **Name** | **Size (bytes)** | **Description** | | Key | 1 | Message type | | Length | 2 | Length of value, in bytes | | Host | 2 | Source, sender of message. | | Seq Number | 1 | Message Sequence Number | | Destination | 2 | Destination host for message. | | Value | *Length* | Message data – each message must be decoded differently depending on Key (message type). |     The sequence number exists to detect out of order or missing packets. The sequence number may start at any value (0-255), but all following messages must be sequentially ordered. The sequence number should reset to 0 upon the 257th message transmitted. |

**ChannelInfo (0x1) -** The ChannelInfo message is used to transmit the list of channels that a provider is or will be streaming. ChannelInfo messages contain the ID (used to associate the channel in the values message), name, unit type, etc. ChannelInfo messages are composed of multiple ChannelInfo data structures – the number of messages should be determined by dividing the length of the message by the size of the structure, (minus the provider name field length). This message should be automatically transmitted every 30 seconds or whenever the internal set of channels being transmitted is changed.

num\_infos = (length-50)/34

The first field of the ChannelInfo message is a fixed-length UTF8 encoded null-terminated 50 byte string which occurs ONCE, before the repeating ChannelInfo structures, which is the name of the provider.

If the number of channels exceeds the MTU, the message may be repeated as many times as needed with additional channels. The Provider Name field must be present in each subsequent message. MTU will be a user setting – most modern hardware can handle a 1500 byte MTU, (especially the Windows/Mac computers that are likely to be used in 99% of cases). If network traffic isn’t being received, the user’s first troubleshooting step should be to reduce the MTU setting on all nodes. Path MTU Discovery is not practical with more than two nodes and is outside the scope of this specification. (Every single node would have to PDMTU to every other node and then transmit that information to all nodes).

After this field, the ChannelInfo structure repeats as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Offset** | **Len** | **Description** |
| channel\_id | 0 | 2 | A unique identifier for the channel, used to associate this information to the value posted in the Values message. |
| vendor | 2 | 1 | Vendor-specific |
| channel\_name | 3 | 30 | UTF8 encoded null-terminated fixed length channel name string. |
| unit | 33 | 1 | See the “units” table. |

**Units Table**

|  |  |
| --- | --- |
| Time (seconds) | 0 |
| Distance (meters) | 1 |
| Speed (kilometers per hour) | 2 |
| Force/Weight (newtons) | 3 |
| Power (kilowatts) | 4 |
| Torque (newton-meters) | 5 |
| Temperature (celsius) | 6 |
| Pressure (kilopascals) | 7 |
| Engine Speed (RPM) | 8 |
| Gear Ratio (rpm/kph) | 9 |
| Acceleration (kph/s) | 10 |
| Air/Fuel Ratio (AFR) | 11 |
| Flow Rate (kg/hr) | 12 |
| Lambda | 13 |
| Volts | 14 |
| Amps | 15 |
| Percentage | 16 |
| NO UNIT | 255 |

**ChannelValues (0x02)**

The ChannelValues message is a list of repeating ChannelValue structures. To determine the number of values in a message, divide message length by the size of a ChannelValue structure. Like the ChannelInfo message, if the number of channels transmitted exceeds the MTU, additional ChannelValues packets may be transmitted with the remaining channels.

num\_vals = length/10

Each ChannelValue structure repeats as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Offset** | **Length** | **Description** |
| channel\_id | 0 | 2 | A unique identifier that corresponds to the id used in a ChannelInfo structure. |
| ts | 2 | 4 | Provider relative timestamp, in milliseconds. (Typically this value should be set to 0 when the provider initializes). |
| value | 6 | 4 | 4-byte single-precision floating point channel value. |

**ClearChannelValues (0x03)**

The ClearChannelValues message contains no data and will have a length of 0. This message is only valid for the ALL\_HOSTS destination address, and instructs all clients to clear any previously stored channel information from the source host. This message should be fired whenever a provider’s channel list changes, and should precede transmission of an updated ChannelInfo message. When possible, this message should be transmitted before a provider goes offline so as to notify clients that the provider is no longer available.

**Ping (0x04)**

The Ping message is used to discover providers AND clients, as well as to synchronize provider-client clocks together. Both specific host addressing and ALL\_HOSTS addressing is valid here. All providers must reply to this message to facilitate host id collision prevention. Any clients that transmit must reply to the ping message as well and for the same reason.

The ping message contains the following fields:

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Offset** | **Length** | **Description** |
| Version | 0 | 1 | The number “1”. If this value is 1-9, communication is allowed. If this value is 10 or greater, a breaking change has been made and communication should cease, ideally with an error message to the user. |
| host\_ts | 1 | 4 | Current local clock value |
| Data | 5 | Length of message - 5 | Arbitrary data to be echoed back. Field is optional (0 length). This field may be used to test maximum payload transmission size if desired. The JETDRIVE specification provides no way to directly take action on this, but a warning can be shown to the user to request a reduction in MTU on all nodes. |

**Pong (0x05)**

The pong message should only ever be sent when a Ping message is received which is addressed to the current provider or ALL\_HOSTS (clients need not reply unless they plan to transmitt). The Pong message MUST be addressed to the host that sent the Ping message. Clients should ignore any Pong message not specifically addressed to them. The ALL\_HOSTS address is not valid for the Pong destination address.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Offset** | **Length** | **Description** |
| Version | 0 | 1 | Remote Version. See Ping command. |
| host\_ts | 1 | 4 | Clock value sent via the Ping message that caused this Pong message to be generated. |
| remote\_ts | 5 | 4 | Current local clock value. |
| Echo Data | 9 | Message length – 5 | Any data transmitted in the Ping command’s data field. |

In order to perform clock synchronization on the client (when the Pong message is received as a response), compare the current local clock value to the clock value in the host\_ts field – divide this difference in half to determine latency from the provider.

To determine clock offset, store the difference between the local clock and the remote\_ts field. Add this value to all incoming sample timestamps from this provider to determine local clock time for the sample, then subtract the latency value to account for network latency.

latency = (current\_time() – host\_ts)/2

offset = current\_time() – remote\_ts

(You’ll need a signed long (64-bit) storage variable for offset, as both ts fields are uint32 types).

Then, when processing sample values from a ChannelValues message:

local\_ts = sample.ts + offset – latency

**RequestChannelInfo (0x06)**

Requests that the specified provider (or ALL\_HOSTS) transmit the ChannelInfo message.

This command has no data and has a length of 0.

**Host ID Determination**

Each node must have a unique host ID. The host ID should NOT be fixed, and should be randomly generated every time a node is initialized. This allows multiple instances of the same node type to exist on the same network. Any clients that transmit on the network must also have a unique host ID. We recommend randomly generating any 16 bit value OTHER than 0x0 (reserved) and 0xFFFF (the ALL\_HOSTS address). Transmit a Ping message to ALL\_HOSTS using source host ID 0 – if no nodes reply using the same host ID as the one generated, the host ID is claimed and valid. If any collisions are detected, generate a new host ID and try again.

**A note on socket binding:**

It is CRITICAL that, when binding your sockets, the socket option, “so\_reuseaddr”, is set to 1/true. If this option is not set, other JETDRIVE nodes running on the same machine *will not* be able to join the multicast group and listen for JETDRIVE traffic.

**Extended Command Set**

No provider is required to respond to these commands – they are optional and are defined for standardizing control of either a dynamometer or logging software.

**RequestLogging (0x10)**

Requests that a node (or all nodes) begin or end logging.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Offset** | **Length** | **Description** |
| log\_enable | 0 | 1 | 0 = end logging  1 = begin logging  2 = cancel logging (do not save log)  3 = get logging state |

**RequestLoggingResponse (0x20)**

Response to a RequestLogging message. May also be transmitted unsolicited when the logging state changes to keep clients updated.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Offset** | **Length** | **Description** |
| log\_enable\_state | 0 | 1 | 0 = logging stopped  1 = logging started |
| log\_start\_ts | 3 | 4 | The local timestamp at which logging was started or stopped last. |

**RequestLoadControl (0x11)**

Allows a client to set the load control target for a dynamometer.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Offset** | **Length** | **Description** |
| lc\_enable | 0 | 1 | 0 = disable  1 = enable  2 = get current setting (other fields ignored) |
| lc\_type | 1 | 1 | RPM = 0  Speed = 1  Accel = 2  Brake% = 3  Torque = 4 |
| lc\_value | 2 | 4 | Load control target value. Four byte single-precision floating point value. Refer to units table for the unit to use. (Brake% should be a value between 0 and 100) |

**RequestLoadControlResponse (0x21)**

The response to a RequestLoadControl message. May also be fired without solicitation whenever load control state is changed in order to keep clients updated on current load control value.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Offset** | **Length** | **Description** |
| lc\_enabled | 0 | 1 | 0 = disabled  1 = enabled  2 = not equipped (dynamometer does not have load control) |
| lc\_type | 1 | 1 | RPM = 0  Speed = 1  Accel = 2  Brake% = 3  Torque = 4 |
| lc\_value | 2 | 4 | Load control target value. Four byte single-precision floating point value. Refer to units table for the unit to use. (Brake% should be a value between 0 and 100) |

**RequestBrake (0x12)**

Enables, disables, or gets the status of the drum brake.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Offset** | **Length** | **Description** |
| brake\_enable | 0 | 1 | 0 = disabled  1 = enabled  2 = get status |

**RequestBrakeResponse (0x22)**

Response to the RequestBrake message. This message may be sent unsolicited to update clients when the brake is engaged or disengaged.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Offset** | **Length** | **Description** |
| brake\_enable | 0 | 1 | 0 = disabled  1 = enabled  2 = not equipped (no brake installed on dynamometer). |