# **Olsonet Communications Corporation**

# **OSS Interface Specification: Alphanet 1.5, 2.0, ATOL.**

June 29, 2015

*Proprietary and confidential*

# Preamble

Here are the names of the relevant network components as used in this document:

|  |  |
| --- | --- |
| **NIB** | (Network Interface Box). By this acronym we shall refer to the equivalent of AP321 from Version 1.0, i.e., the device consisting of the Renesas - TI tandem. |
| **Peg** | The TI (Olsonet) component of a NIB. This name reflects our tradition of referring to a forwarding node of our wireless network. |
| **Tap** | The Renesas component of a NIB. The name reflects the fact that the Renesas module represents a tap to the wired (RS485) section of the network. |
| **Tag** | A (potentially) mobile, battery powered device running Olsonet firmware and communicating wirelessly with Pegs (the equivalent of AP320/AP319 from Version 1.0). |

This document deals with the communication between the Peg and the Tap of the same NIB, which is carried out over the UART interface interconnecting the two components. The physical parameters of the UART interface are: **115200 bps, 8 bits, no parity, 1 stop bit**.

By the OSS we mean the external system responsible for running and controlling the network. The Tap components of the NIBs communicate with the OSS via wired connections (over RS485 or over other means) which are transparent to the Pegs. They can pass OSS commands to their tandem Pegs over the UART and receive replies (to be forwarded to the OSS) the same way.

Any Peg of any NIB can (in principle) communicate with its tandem Tap. Generally, not all Taps are expected to provide this kind of functionality. In particular, it can be confined to the NIBs acting as sinks for the events delivered by the wireless network.

# Data format

Data exchanged between Pegs and their associated Taps are encapsulated into packets consisting of sequences of bytes. The format of those packets is similar to the format assumed in Version 1.0. Specifically, a packet begins with the character STX (ASCII code 0x02) and terminates with ETX (ASCII code 0x03). Immediately preceding ETX is a single checksum byte, dubbed bcc, calculated as the *negated arithmetic sum modulo 256* of all bytes in the packet,[[1]](#footnote-1) including the starting STX byte and the terminating ETX byte + zero bcc, but excluding any escaping DLLs (see below). This part of the packet format (see ) is called the *OSS frame*. The part between the starting STX byte and the checksum byte (exclusively) is called the *OSS [frame] content*.

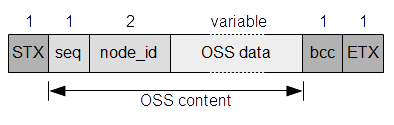


Figure . OSS frame format.

One more special character, DLE (ASCII code 0x10) is used for escaping special characters within the frame. If any of the bytes between STX and ETX (i.e., any byte of the frame's OSS content or the checksum byte) looks like STX, ETX, or DLE, then it is preceded with DLE (ASCII code 0x10). For example, this sequence of bytes:

**0x55 0x57 0x02 0x13 0x12 0x10**

will be encoded as:

**0x02 0x55 0x57 0x10 0x02 0x13 0x12 0x28 0x03**

The escape character has been highlighted. The escapes are transparent with respect to the OSS content, i.e., they belong to the OSS frame, but are removed before the OSS content is presented to its interpreter.

The checkusm byte is defined in such a way that the arithmetic sum (modulo 256) of all bytes in a correctly received packet, *excluding* the DLE escapes, yields a zero byte. Here is the (conceptual) algorithm for calculating the checksum in an outgoing packet (note that the packet includes no escape bytes, but it does include STX and ETX):

**unsigned char \*packet, csum;**

**int packet\_length;**

**...**

**packet [packet\_length - 2] = 0;**

**for (csum = 0, int i = 0; i < packet\_length; i++)**

**csum -= packet [i];**

**packet [packet\_length - 2] = csum;**

**...**

The maximum length of the OSS content (escape DLE's excluded) is 82 bytes. The node\_id field is a two-byte (little-endian) Peg identifier.[[2]](#footnote-2) For a request going from the Tap to the associated Peg, node\_id must match the Peg node Identifier or be zero (otherwise, the Peg will completely ignore the request). For a request going from the Peg to the Tap, node\_id is always the Id of the sending Peg.[[3]](#footnote-3)

## Sequence numbers

The role of seq (content sequence numbers) is to tag the requests passed in the OSS frames with distinguishing identifiers for the purpose of:

1. matching acknowledgements to requests, as explained below
2. preventing multiple interpretation of the same (repeatedly issued) request

The least significant 7 bits of seq are interpreted as a sequence number associated with the OSS data. Normally, the sequence number is a value from 2 to 127, with the intention that consecutively issued requests are tagged with increasing values of seq wrapping around at 128 to 2 (i.e., the next value to 127 is 2, not 0). These numbers are maintained independently for each side of the Tap-Peg tandem. Having received and processed a request with a given sequence number the peer (Peg or Tap) will ignore subsequent requests with the same number until it receives one with a different sequence number.[[4]](#footnote-4)

This facilitates repetitive requests with the same OSS data sent blindly with no need for acknowledgements, which provides for an alternative (and often convenient) way of increasing the reliability of tandem communication without waiting for acknowledgements and retrying on timeouts. Note that instead of re-sending a request, say, three times, and waiting a few milliseconds between the retries for an acknowledgement, the same (or better) kind of reliability will be achieved by sending the same request blindly three times without waiting for anything. The sequence number makes sure that only (at most) one copy of the request will be interpreted by the recipient. Another potential use for this feature is in a situation where one side wants to send a high-rate stream of messages to the other side.[[5]](#footnote-5) Typically, in such a situation, one is willing to put up with occasional losses, while the need to absorb acknowledgements for every single message slows things down considerably. Thus the option to switch the acknowledgements off becomes useful.

## Acknowledgements

If the most significant bit of seq (dubbed FG\_ACKR) is set,[[6]](#footnote-6) it indicates that the sender expects an immediate acknowledgement. Having correctly received and accepted such a request, the other party will immediately respond with a frame looking like this:

STX seq node\_id 0x00 status bcc ETX

which is interpreted as a positive or negative acknowledgement (depending on *status*). The fact that a frame is an acknowledgment is recognized by the zero opcode, i.e., the first byte of its payload. Only an acknowledgment frame can have zero in this place. The least significant 7 bits of seq match the 7-bit sequence number of the request being acknowledged. The most significant bit of seq is ignored (reserved for future extensions).[[7]](#footnote-7) The status field is a single byte. Zero means "success" and any nonzero value indicates a failure. Note that only immediate failures can be signaled this way. For example, if the request calls for some long-term operations involving exchanges across the network, a positive acknowledgment (zero status) only means that the request has been recognized and formally accepted for processing. Similarly, a negative acknowledgement means that the peer has immediately concluded that the request has failed, e.g., the Peg has requested something that lies beyond the Tap's capabilities. Note that we do not (negatively) acknowledge, e.g., incorrectly received packets (ones with illegal format or with the wrong checksum).[[8]](#footnote-8) Such packets will be resent on a timeout, if the sender cares about their reliable delivery. An acknowledgement frame is sent, if and only if the following two conditions hold simultaneously:

1. the original request (the one being acknowledged) was issued with FG\_ACKR set
2. the OSS frame has been correctly received, i.e., its parity byte and formal structure were valid

In particular, a correctly received frame with a nonzero opcode[[9]](#footnote-9) and FG\_ACKR set in its seq field is always acknowledged, even if it is a duplicate, i.e., its sequence number is the same as that of the last-processed request. On the other hand, a frame with FG\_ACKR cleared is never acknowledged, even if the request immediately fails.

The list of error codes (nonnegative status values) will be defined as we proceed (perhaps we should associate such codes with specific requests).

## Resets

When any party (the Tap or the Peg) resets, it will set its last-received request sequence number to zero and its next-to-send sequence number to 1. As these sequence numbers can never arrive in a "normal" incoming request, this will:

1. force the reset peer to accept the first incoming request from the other party as non-duplicate
2. force the other party to accept the first outgoing request from the reset peer as non-duplicate

By setting its last-received sequence number to zero the resetting peer becomes prepared to accept any request coming from the other party. Note that no legitimate request can ever have a zero sequence number. Thus, the role of zero in this context is to represent NULL or NONE. By setting the sequence number in its first outgoing request to 1, the resetting peer effectively notifies the other party that it has reset (this is the only circumstance where the sequence number can be 1) and also makes sure that whatever the state of the other party, the request will be looked at. To make sure that its status is noticed, the peer can send the first request after its reset (whatever it is) a few times in a row.

# Types of OSS data

An OSS frame whose OSS data part starts with a zero byte (zero opcode) carries a (low-level) acknowledgement. The data part of an ACK must include at least one more byte, i.e., the status. We do not say what happens when there are more bytes. Perhaps we should assume for now that any bytes in excess of the status byte are ignored. Otherwise (i.e., nonzero opcode), the data field contains some message to the other party and fits one of the following three formats:

## Command

This kind of message is sent from the Tap to the Peg. Its format is shown in . The op\_code field identifies the command type, op\_ref is a *reference value* associated with the command, whose purpose is to identify responses to different (outstanding) commands of the same type, node\_id is the target node Id, and payload is the op\_code-specific content (parameters) of the command.

Note that the node\_id field of a command need not be the same as the node\_id field of the encapsulating packet, i.e., commands can be addressed to remote Pegs.[[10]](#footnote-10) Typically a command describes some *action* to be carried out by the target Peg.

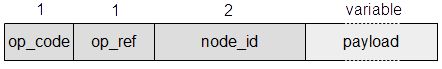


Figure . Command format.

## Response

A message of this type originates in a Peg in *response* to a *command* and is always related to a specific command. The response format is shown in .

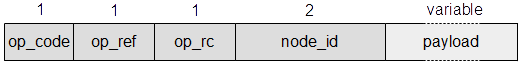


Figure . Response format.

The first two fields match the respective attributes of the corresponding (soliciting) command. The additional field, op\_rc, is the response code (one byte) indicating the command's processing status. The header is followed by the command-specific payload constituting the actual response.

Some commands are queries triggering prescribed, specific responses from target nodes. Such a command is (application-level) acknowledged by its response. For a non-query command, no automatic (application-level) acknowledgement is generated, unless the most significant bit of the reference byte is set. In such a case, the target Peg will issue a response consisting of the matching op\_code/op\_ref pair, op\_rc, and an empty payload.

## Report

Reports (see ) are unsolicited messages sent from the Peg to the Tap. In contrast to responses, they are not directly associated with any prior commands.

The rep\_type field identifies the report type. The contents of payload are report-type-specific. Note that 0xFF cannot be used as a command op\_code (the role of this byte is to act as a reserved "opcode" telling reports from command responses).

A typical example of a report is an event message or a (periodic) sensor readout.

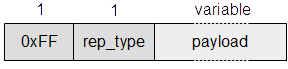


Figure . Report format.

# The list of messages

This list doesn't exhaust all message types and their parameters. At this stage we want to agree on the format of OSSI for the upcoming deployment (dubbed Alphanet 1.5) and hint at a few possibilities that we may want to implement in Alphanet 2.0 (1.5 will not change the functionality of 1.0 beyond the OSSI format).

Note that the OSSI defined here applies only to Pegs. Configuring Tags is intentionally left out, as it is not known yet how we should approach it in Alphanet 2.0 or in ATOL firmware.

## Commands

|  |  |  |
| --- | --- | --- |
| **Set node parameters** | |  |
| **Symbolic** | **OpCode** | **Type** |
| CMD\_SET | 0x02 | request |

The command sets various configuration parameters of the Peg. The payload consists of any sequence of pairs:

*ptype pvalue*

up to the maximum length of 74 bytes, where *ptype* is a single byte identifying the parameter, and *value* is the parameter value. The length of *pvalue* is determined by *ptype*. In all cases where *pvalue* is numerical and longer than one byte, it is interpreted little-endian. Here is the list of *ptypes* and the corresponding *pvalues*:

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbolic** | **Code** | **Size** | **Description** |
| PAR\_LH | 0x01 | 2 (UI) | Local Host (aka Node ID) |
| PAR\_MID | 0x03 | 2 (UI) | Master ID |
| PAR\_NID | 0x04 | 2 (UI) | Network ID |
| PAR\_TARP\_L | 0x05 | 1 (UB) | TARP level |
| PAR\_TARP\_S | 0x06 | 1 (UB) | TARP slack |
| PAR\_TARP\_R | 0x07 | 1 (UB) | TARP route recovery speed |
| PAR\_TARP\_F | 0x08 | 1 (UB) | TARP forwarding (on/off) |
| PAR\_TARP | 0x09 | 1 (UB) | TARP all parameters (byte) |
| PAR\_TAG\_MGR | 0x0B | 1 (UB) | Accept alarms, manage tags (not only on the Master) |
| PAR\_AUDIT | 0x0C | 2 (UI) | timeout (in seconds) |
| PAR\_AUTOACK | 0x0D | 1 (UB) | Master’s autoack (no OSSI involvement). Likely will stay NOT IMPLEMENTED (always ON). |
| PAR\_BEAC | 0x0E | 2 (UI) | Beacon frequency (Master beacon) |
| PAR\_SNIFF | 0x1D | 1 (UB) | Sniffing mode: the values are 0 - off, 1 - on, 2 - on+promiscuous, i.e., ignore the Network ID |
| ... more? ... |  |  |  |

|  |  |  |
| --- | --- | --- |
| **Get node parameters** | |  |
| **Symbolic** | **OpCode** | **Type** |
| CMD\_GET | 0x01 | query |

The command retrieves configuration parameters and various attributes of the node. The payload consists of a sequence of bytes:

*ptype ... ptype*

specifying the parameters and attributes to be retrieved. Their values will arrive in the response payload up to the total length of 74 bytes as a sequence of pairs *ptype pvalue* (see CMD\_SET). The list of *ptypes* for CMD\_GET includes some non-settable attributes (not available to CMD\_SET):

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbolic** | **Code** | **Size** | **Description** |
| PAR\_LH | 0x01 | 2 (UI) | Local Host aka Node ID |
| ATTR\_ESN | 0x02 | 4 (UL) | Equipment Serial Number |
| PAR\_MID | 0x03 | 2 (UI) | Master Id |
| PAR\_NID | 0x04 | 2 (UI) | Network ID |
| PAR\_TARP | 0x09 | 1 (UB) | TARP byte (level, slack, rte, fwd) |
| ATTR\_TARP\_CNT | 0x0A | 6 | TARP counters 3xUI (rcv, snd, fwd) |
| PAR\_TAG\_MGR | 0x0B | 1 (UB) | Accept alarms, manage tags (not only on the Master) |
| PAR\_AUDIT | 0x0C | 2 (UI) | timeout (in seconds) |
| PAR\_AUTOACK | 0x0D | 1 (UB) | Master’s autoack (no OSSI involvement) Likely will stay NOT IMPLEMENTED. |
| PAR\_BEAC | 0x0E | 2 (UI) | Beacon frequency |
| ATTR\_VER | 0x0F | 2 (UI) | Firmware version (0xMMmm: Major minor) |
| ATTR\_UPTIME | 0x1A | 4 (UL) | since boot in seconds |
| ATTR\_MEM1 | 0x1B | 4 | 2 UI (mem free, min mem free so far) |
| ATTR\_MEM2 | 0x1C | 4 | 2 UI (max free chunk, free stack size) |
| PAR\_SNIFF | 0x1D | 1 (UB) | Sniffing mode |
| ... more ... |  |  |  |

|  |  |  |
| --- | --- | --- |
| **Set peg associations** | |  |
| **Symbolic** | **OpCode** | **Type** |
| CMD\_SET\_ASSOC | 0x13 | request |

The command is an exact implementation of the corresponding 1.0 function. Alphanet 1.5 will continue with it. How the command and the actual functionality will look like in Alphanet 2.0 and ATOL is still debatable. Same applies to the corresponding Get and Clear commands.

|  |  |  |
| --- | --- | --- |
| **Get peg associations** | |  |
| **Symbolic** | **OpCode** | **Type** |
| CMD\_GET\_ASSOC | 0x14 | request |

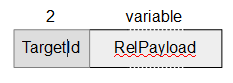
See CMD\_SET\_ASSOC.

|  |  |  |
| --- | --- | --- |
| **Clear peg associations** | |  |
| **Symbolic** | **OpCode** | **Type** |
| CMD\_CLR\_ASSOC | 0x15 | request |

See CMD\_SET\_ASSOC.

|  |  |  |
| --- | --- | --- |
| **Relay payload** | |  |
| **Symbolic** | **OpCode** | **Type** |
| CMD\_RELAY | 0x41, 0x42, 0x43 | request |

Request:



The command implements 1.0’s relays to be used in 1.5. Likely, it’ll be modified for 2.0 and ATOL. For example, we should discuss the actual needs, introduce a relay type / version in the payload and make sure that we accommodate existing devices and upcoming designs, especially in ATOL. Note that end to end relay acknowledgment, present in the 1.0 firmware but not used should be carefully designed on both sides of the OSSI. For now, there is only a placeholder for it below.

|  |  |  |
| --- | --- | --- |
| **Trace route** | |  |
| **Symbolic** | **OpCode** | **Type** |
| CMD\_TRACE | 0x51 | request |

The command requests a trace (forward, backward, bidirectional) to a target node. This is useful for setups and troubleshooting, could be used also for location data gathering (self-surveys). Note that this functionality is taxing all the involved nodes (especially multihopping traces). This command will NOT be implemented before Alphanet 2.0 and ATOL.

|  |  |  |
| --- | --- | --- |
| **Operator Directed Route** | |  |
| **Symbolic** | **OpCode** | **Type** |
| CMD\_ODR | 0x52 | request |

The command carries its payload along an explicitly requested route. As the route trace, it is designed for troubleshooting but proved to be useful for several applications. Note that ODR packets are NOT routed by the network. This command will NOT be implemented before Alphanet 2.0 and ATOL.

The two commands below and the corresponding reports are devised for *hybrid* networks where (sporadic) connectivity to classical networks may enable *warping* packets in parallel to the ad-hoc RF network. Alphanet 1.0 with RS-485 wired AP321 or cell phones with ad-hoc networked dongles are good examples. Another interesting one is a setup where some virtual nodes (think dongle-less handsets) and real ones are interconnected.

Tentatively, the opcode space 0xA? is reserved for this and similar functionality.

|  |  |  |
| --- | --- | --- |
| **Forward packet** | |  |
| **Symbolic** | **OpCode** | **Type** |
| CMD\_FORWARD | 0xA1 | request |

The command passes to the Peg a packet forwarded from another node. The packet, exactly as submitted with the corresponding REP\_FORWARD request (see below), comes as the command's payload.

The op\_ref argument is ignored. The node\_id argument can refer to a node different from the current Peg. In such a case, it means that the Peg is expected to forward the packet (over RF) to the indicated node.[[11]](#footnote-11)

|  |  |  |
| --- | --- | --- |
| **Inject packet** | |  |
| **Symbolic** | **OpCode** | **Type** |
| CMD\_INJECT | 0xA2 | request |

The command instructs the node to pass its payload as a complete packet to be transmitted (without any interpretation) on the RF interface. The op\_ref argument of the command is ignored. The node\_id argument must be the same as the corresponding argument of the encapsulating OSS frame.[[12]](#footnote-12)

**...**

## Responses

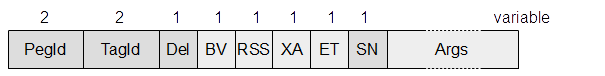
...

## Reports

|  |  |
| --- | --- |
| **Event report** | |
| **Symbolic** | **RepType** |
| REP\_EVENT | 0x00 |

The reports notifies the OSS about an event.

Payload layout:

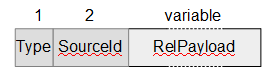


|  |  |
| --- | --- |
| **PegId** | The identifier of the Peg that has received the event from the originating Tag. |
| **TagId** | The identifier of the Tag generating the event. |
| **Del** | The accumulated delay of the event packet in seconds (255 if > 255). |
| **BV** | Battery voltage at the Tag. |
| **RSS** | Received signal strength at the receiving Peg. |
| **XA** | Transmission attributes. |
| **ET** | Event type. |
| **SN** | The event sequence number assigned by the Tag. |
| **Args** | Event arguments (depends on event type, extends until the end of payload. |

Note that the length of event arguments is flexible. It is even possible for events of the same type to have arguments of different length. For example, a GPS readout[[13]](#footnote-13) is typically an ASCII string encoding the geographic position information, whose length may vary depending on its momentary content.

|  |  |
| --- | --- |
| **RELAY at destination** | |
| **Symbolic** | **RepType** |
| REP\_RELAY | 0x01 |

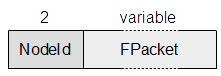
The report passes relayed info to the Tap. SourceId is the sender:



|  |  |
| --- | --- |
| **Forward request** | |
| **Symbolic** | **RepType** |
| REP\_FORWARD | 0xA1 |

The report passes a packet to the Tap for forwarding (over RS485) to another node.

Payload layout:



Note that formally NodeId need not refer to a Peg, i.e., we do not preclude forwarding to Tags as a matter of principle. The FPacket field is treated as the packet to be forwarded, i.e., the payload of CMD\_FORWARD to be delivered to the destination Peg.

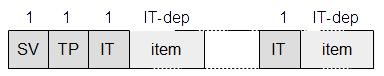
|  |  |
| --- | --- |
| **Sniffed packet** | |
| **Symbolic** | **RepType** |
| REP\_SNIFF | 0xE1 |

The report passes a sniffed RF packet to the Tap. The report type byte is followed by the complete sniffed packet as received by the RF module.

We intend to use the 0xD? opcode family for a wide range of efficient debugging facilities.

|  |  |
| --- | --- |
| **Log entry** | |
| **Symbolic** | **RepType** |
| REP\_LOG | 0xD1 |

The report passes to the Tap a message for logging. The payload format is:



where SV is the message severity code and TP is the message type (a numerical identifier of the message's content). The remaining portion of the payload consists of pairs <IT, item> where IT is the item type and item is the type-specific element of the message. The total length of the payload is limited to 77 bytes. A value of IT between 0 and 74 indicates that the corresponding item is a string of ASCII characters whose length is IT+1 bytes. The remaining legitimate IT values are:

|  |  |
| --- | --- |
| 0x80 | single byte to be shown in hex |
| 0x81 | 16-bit integer to be shown in hex |
| 0x82 | 32-bit integer to be shown in hex |
| 0xC0 | single byte to be shown as unsigned decimal |
| 0xC1 | 16-bit integer to be shown as unsigned decimal |
| 0xC2 | 32-bit integer to be shown as unsigned decimal |
| 0xD0 | single byte to be shown as signed decimal |
| 0xD1 | 16-bit integer to be shown as signed decimal |
| 0xD2 | 32-bit integer to be shown as signed decimal |
| 0xE0 | the remaining bytes in the payload to be shown in hex |
| 0xE1 | the remaining pairs of bytes in the payload to be shown as 16-bit integers in hex |
| 0xE2 | the remaining quadruples of bytes in the payload to be shown as 32-bit integers in hex |
| 0xF0 | the remaining bytes in the payload to be shown as unsigned decimals |
| 0xF1 | the remaining pairs of bytes in the payload to be shown as 16-bit unsigned decimals |
| 0xF2 | the remaining quadruples of bytes in the payload to be shown as 32-bit unsigned integers |
| 0xF8 | the remaining bytes in the payload to be shown as signed decimals |
| 0xF9 | the remaining pairs of bytes in the payload to be shown as 16-bit signed decimals |
| 0xFA | the remaining quadruples of bytes in the payload to be shown as 32-bit signed integers |

The intention behind this format is to provide for a relatively simple and flexible configuration of elements in a log message without having to 1) format the message at the node (which takes time and code), or 2) use rigid predefined templates shared by the node and the OSS (and, unavoidably, dependent on firmware version). This way the node does some rudimentary formatting with the OSS doing the rest.

Appendix A

This is a placeholder for documenting Alphanet 1.5 – a codename for the functionality as close to 1.0 as possible, with the new OSSI.

# How far from 2.0?

* Several commands / responses are not implemented.
* No remote execution, i.e. node id (within the command) must be the one attached to the Tap.
* Master Id is fixed at 1.
* Master change must be reworked for multiple masters (that includes the Eye Butler project).
* Relays should be reworked for a bit more flexibility.

# Testing, quick start, illustrations, divagations

VUEE configuration gor ‘light testing’: shared\_plug.tcl, apki.xml (Master, Peg2, Tag101 (AP319), Tag102 (AP320). (All IF notes for Alphanet 1.0 were produced from the same setup.)

Within shared\_plug, default debug / dump level is set to 2 (equivalent of ‘de 2’. Attach UART(hex) to Peg2 – all commands are entered there and the responses pasted from there. Occasionally, we’ll be using EMUL window as well.

Note that this is far from complete OSSI tests – we list the commands & reposnses & dwell on some side effects only to help with further OSSI and related tools development.

**CMD\_GET**

x82 2 0 1 80 2 0 1 2 3

-->: [82 02 00 01 80 02 00 01 02 03]

<-U: [82 02 00 00 00]

<-U: [04 02 00 01 80 00 02 00 01 02 00 02 02 00 ca ba 03 01 00]

FG\_ACKR is set (in 0x82) so the ACK is sent back before the due response.

x03 2 0 1 80 2 0 1 2 3

-->: [03 02 00 01 80 02 00 01 02 03]

<-U: [05 02 00 01 80 00 02 00 01 02 00 9e 02 02 00 ca ba 99 03]

FG\_ACKR clear, no ACK.

x02 2 0 1 1 2 0 1 2 3

-->: [02 02 00 01 01 02 00 01 02 03]

<-U: [05 02 00 01 80 00 02 00 01 02 00 02 02 00 ca ba 03 01 00]

0x80 in op\_ref doesn’t make a difference here.

x02 2 0 1 1 2 0 1 2 3

-->: [02 02 00 01 01 02 00 01 02 03]

Duplicate frame but FG\_ACKR clear.

x82 2 0 1 1 2 0 1 2 3

-->: [82 02 00 01 01 02 00 01 02 03]

<-U: [82 02 00 00 05]

Duplicate frame with FG\_ACKR set.

x03 2 0 1 80 2 0 1 77 3

-->: [03 02 00 01 80 02 00 01 77 03]

<-U: [07 02 00 01 80 02 02 00 01 02 00 03 01 00]

Bad parameter (0x77) skipped.

x02 2 0 1 1 2 0 1 77 3

-->: [02 02 00 01 01 02 00 01 77 03]

<-U: [08 02 00 01 01 02 02 00 01 02 00 03 01 00]

Bad parameter (0x80 in op\_ref doesn’t make a difference here).

x03 7 0 1 80 2 0 1 2 3

-->: [03 07 00 01 80 02 00 01 02 03]

x83 2 0 1 80 2 0 1 77 3

-->: [83 02 00 01 80 02 00 01 77 03]

<-U: [83 02 00 00 03]

Bad node id (without and with FG\_ACKR)

CMD\_SET

Only local host and TARP forwarding can be set in 1.5 (as it was in 1.0). LH can’t be changed from or to 1 (reserved for fixed master id). Be careful with (par, val) lists, sanity checks are difficult if not impossible without field marking or explicit lengths – we leave it to OSS (should we?) The parameters are set until the 1st error (returned only if op\_ref has MSBit set.

x02 2 0 2 80 2 0 1 1 0

-->: [02 02 00 02 80 02 00 01 01 00]

<-U: [09 02 00 02 80 01 02 00]

Not allowed: we’ve tried to change LH to 1.

x03 2 0 2 80 2 0 1 0 1

-->: [03 02 00 02 80 02 00 01 00 01]

<-U: [0a 00 01 02 80 00 00 01]

OK, we’ve changed LH 1 to 256. Note that the response came from the new id… Hm, I don’t like LH mutable in general. Better know what you’re doing.

x02 0 1 2 80 0 1 1 2 0

-->: [02 00 01 02 80 00 01 01 02 00]

<-U: [0b 02 00 02 80 00 02 00]

Back to 2.

… to be continued (all #defines are in commons.h, most of this crap in oss\_peg\_tcve.cc) …

1. After some deliberations, we have decided against the 16-bit checksum (previously suggested). A relatively easy solution would be to use Fletcher's checksums, but even that is rather expensive (computationally, e.g., it requires non-trivial division) and whichever way we look at it, appears as an overkill. After all, the UART connects components on the same PCB and should be in principle reliable. The simple sum makes much better use of the checksum byte than the parity bits while being practically equally easy to compute, so this modification is sensible, cheap, and natural. [↑](#footnote-ref-1)
2. All multi-byte numbers are assumed to be little-endian. [↑](#footnote-ref-2)
3. This (in principle) facilitates scenarios when the same OSS interface is broadcast-shared by multiple Pegs. The feature can be ignored in Alphanet 2.0. For example, the node\_id field can always be set by the Tap to zero. [↑](#footnote-ref-3)
4. Strictly speaking, the sequence numbers do not have to steadily increase, but only change (for consecutively issued requests) within the range 2 - 127. [↑](#footnote-ref-4)
5. This doesn't mean that we anticipate such scenarios in Alphanet. [↑](#footnote-ref-5)
6. Note that this bit doesn't count to the proper sequence number. [↑](#footnote-ref-6)
7. Feel free to assume that it is always zero, if it simplifies anything. [↑](#footnote-ref-7)
8. Note that to acknowledge a request one has to know with confidence that the request's sequence number has been recognized correctly. [↑](#footnote-ref-8)
9. I.e., a non-ACK frame (obviously, ACKs are never acknowledged). [↑](#footnote-ref-9)
10. This feature need not be taken advantage of in Alphanet 2.0, but it is included as an option. [↑](#footnote-ref-10)
11. At least this possibility is formally available. [↑](#footnote-ref-11)
12. Probably, to simplify things, it should be ignored. [↑](#footnote-ref-12)
13. This doesn't mean that we want to have GPS modules in Alphanet Tags. [↑](#footnote-ref-13)