# **Olsonet Communications Corporation**

P. Gburzynski, W. Olesinski

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

September 26, 2016 *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 Figure 1) is called the *OSS frame*. The part between the starting STX byte and the checksum byte (exclusively) is called the *OSS [frame] content*.

offormat.wmf

Figure 1. 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. 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 which translates into the 79-byte limit for OSS data. 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 (otherwise, the Peg will completely ignore the request).[[3]](#footnote-3) Some request types may admit broadcast (or "unassociated") interpretation; for such requests, node\_id equal zero is interpreted as "any", meaning that any Peg will treat it as its own Id and respond to the request. The decision is left to individual requests. For a request going from the Peg to the Tap, node\_id is always the Id of the sending Peg (and it can never be zero).

## 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 OSS data. 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). The status field is a single byte. Zero means "success" and a nonzero value indicates a failure (with one exception described below). 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).[[7]](#footnote-7) 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 request 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

Note that a frame with FG\_ACKR cleared is never acknowledged, even if the request immediately fails. On the other hand, a correctly received frame with a nonzero opcode[[8]](#footnote-8) 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.

If the frame is *not* a duplicate, the status byte in the corresponding acknowledgment frame reflects the actual status of the request, i.e., 0 meaning "success" and non-zero meaning an immediate failure.

For a duplicate frame, if the original request has failed, the status field in the acknowledgment will be the same as that in the original ACK (indicating the same kind of failure). However, if the original request has succeeded (i.e., the original status was zero), the status sent in response to the duplicate (any duplicate of the same request based on the value of seq) will be 0x05 (RC\_DUPOK). This special value allows the other party 1) to conclude that its request has in fact succeeded, 2) to learn that no further copies of the request are necessary (so it can stop retrying).

Note that frames are acknowledged individually and (in a sense) independently. In particular, if the first (non-duplicate) request was issued with FG\_ACKR cleared and a subsequent duplicate is issued with FG\_ACKR set, the duplicate will be consistently acknowledged (according to the acknowledgment rules for duplicates), even though the original (non-duplicate) request wasn't.

Here is the preliminary list of status codes:

|  |  |
| --- | --- |
| 0x00 (RC\_OK) | OK |
| 0x01 (RC\_EVAL) | illegal request, e.g., an attempt to dethrone the (hardwired) master node |
| 0x02 (RC\_EPAR) | illegal request parameter, e.g., bad parameter tag for CMD\_SET |
| 0x03 (RC\_EADDR) | illegal address, e.g., the command looks like being addressed to another Peg |
| 0x04 (RC\_ENIMP) | command not implemented |
| 0x05 (RC\_DUPOK) | duplicate previously succeeded (see above) |
| 0x06 (RC\_ELEN) | illegal length |
| 0x07 (RC\_ERES) | no resources; this basically means that the Peg is experiencing intermittent memory problems (like being unable to allocate a buffer to service the request) |

The list of codes will likely grow as we proceed.

## 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. The option to be able to send extra bytes in an ACK should be kept open for future extensions.

If the OSS data part of a frame begins with a nonzero byte, the data part contains some message to the other party, which 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 Figure 2. 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.

coformat.wmf

Figure 2. Command format.

Note that, in principle, the node\_id field of a command need not be the same as the node\_id field of the encapsulating frame, i.e., commands can be addressed to remote Pegs.[[9]](#footnote-9) Typically a command describes some *action* to be carried out by the target Peg.

The maximum length of a command payload is 79-4=75 bytes.

## 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 3.

reformat.wmf

Figure 3. Response format.

The fields: op\_code, op\_rc, and node\_id 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 prescribed response. For a non-query command, no automatic (application-level) acknowledgement is generated, unless the most significant bit of the op\_ref is set (its role is similar to FG\_ACKR, but at the application level). 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.

Similar to the seq field of the OSS frame, the most significant (FG\_ACKR) bit of op\_ref is ignored when matching responses to commands.

The op\_rc code of zero (RC\_OK) means success. A nonzero code indicates an error or a problem. A query response (which normally returns a payload) may return a nonempty payload even if its op\_rc signals an error. Generally, if there is a payload in a response, then it can be interpreted regardless of the value of op\_rc.

The maximum length of a response payload is 74 bytes.

## Report

Reports (see Figure 4) are unsolicited messages sent from the Peg to the Tap. In contrast to responses, they are not directly associated with any prior commands.

rpformat.wmf

Figure 4. Report format.

The rep\_type field identifies the report type. The contents of *payload* are report-type-specific. Note that 0xFF acts as a reserved "opcode" telling reports from command responses and acknowledgments.

The maximum length of a report payload is 77 bytes.

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

# The list of messages

This list is (permanently) preliminary and will grow.

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 75 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 as little-endian. Here is the list of *ptypes* and the corresponding *pvalues*:

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbolic** | **PType** | **Size** | **PValue** |
| 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 (encoded into one 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\_PMOD | 0x10 | 1 (UB) | Peg mode: PMOD\_REG(0) – regular mode  PMOD\_REG(1) – configurable mode  PMOD\_CUST(2) – custodian mode  PMOD\_EXCC(3) – exclusive custodian mode |
| PAR\_TARP\_RSS | 0x11 | 1 (UB) | RSSI threshold for SPD cache updates |
| PAR\_RFCHAN | 0x12 | 1 (UB) | RF channel (0-255) |
| PAR\_SNIFF | 0x1D | 1 (UB) | Sniffing mode: the value consists of two one-nibble fields (NID, REP). E.g., 0x23 represents NID = 2 and REP = 3. Zeros are placeholders for ‘don’t change’.  NID codes: 0 (default) – sniffer plugin not registered; 1 – plugin closed (or not opened yet); 2 – SID unchanged, all packets passed to TARP; 3 – SID set to NONE, TARP sees packets with SID = net\_id or 0; 4 - SID set to NONE, TARP sees no packets.  REP 1 (default) – packets with SID = net\_id or 0 are reported (with the REP\_SNIFF report type); 2 – all packets are reported; 3 – reported what is not passed to TARP or not addressed to the host nor broadcast. |

|  |  |  |
| --- | --- | --- |
| **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** | **PType** | **Size** | **Parameter/Attribute** |
| 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) |
| PAR\_PMODE | 0x10 | 1 (UB) | Peg mode, see CMD\_SET |
| PAR\_TARP\_RSS | 0x11 | 1 (UB) | RSSI threshold for SPD cache updates |
| PAR\_RFCHAN | 0x12 | 1 (UB) | RF channel (0-255) |
| ATTR\_UPTIME | 0x1A | 4 (UL) | since boot in seconds |
| ATTR\_MEM1 | 0x1B | 4 | 2xUI (mem free, min mem free so far) |
| ATTR\_MEM2 | 0x1C | 4 | 2xUI (max free chunk, free stack size) |
| PAR\_SNIFF | 0x1D | 1 (UB) | Sniffing mode, see CMD\_SET |

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

The command is an exact implementation of the corresponding 1.0 function (its payload looks exactly as the payload of the 1.0 variant). This includes the option for setting/clearing the learning mode. 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 |

Payload layout:

RelPayload

2

variable

TargetId

The command implements 1.0’s relays to be used in 1.5. Likely, it will 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, already 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 multi-hopping 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.

|  |  |  |
| --- | --- | --- |
| **Neighborhood** | |  |
| **Symbolic** | **OpCode** | **Type** |
| CMD\_NHOOD | 0x53 | request |

The command cannot be broadcast (node\_id in the command’s frame must be equal to local\_host, otherwise EADDR is returned). All parameters in payload are obligatory (ELEN is returned otherwise).

The command triggers a chain of messages to relay and report the addressed node’s neighborhood. Note that when the command is issued from Custodian or uses its multi-hop capacity, ‘reverse RSSI’ shows the reading from triggered reports’ last hop destination.

The payload:

TargetHost

Ref

HOC

2

1

1

TargetHost: The msg that solicits responses is sent there. Broadcast (TargetHost = 0) is allowed and natural if with common sense constraints on the radius – nonzero HOC may be useful, but resulting data don’t describe any neighborhood.

Ref: reference quoted in resulting reports.

HOC: maximum number of hops allowed – radius with TargetHost as the centre. Zero means proximity (one hop exactly). Note that HOC = 0 usually is very different from HOC = 1 if TARP’s slack is nonzero, as nonzero HOC triggers routing and implicit multi-hopping both ways.

For the results, see REP\_NHOOD.

|  |  |  |
| --- | --- | --- |
| **Reset** | |  |
| **Symbolic** | **OpCode** | **Type** |
| CMD\_RESET | 0x54 | request |

The command resets the receiving node. Note that it can be issued (is designed for) from Custodian. Any (or no) payload is allowed and ignored. However, within the 2.0 timeframe, we’re likely to add a ‘level’ parameter, for ‘factory defaults’ (optional NVM clear).

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 their forwarding in 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 case 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.[[10]](#footnote-10)

|  |  |  |
| --- | --- | --- |
| **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.[[11]](#footnote-11)

**...**

## Responses

...

## Reports

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

The report notifies the OSS about an event.

Payload layout:

erpayload.wmf

|  |  |
| --- | --- |
| **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 in principle 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[[12]](#footnote-12) is typically an ASCII string encoding the geographic position information, whose length may vary depending on its momentary content. All event reports in Alphanet 1.5 carry (at least) three *standard* argument bytes, as explained below.

The formula to transform the reported battery voltage indication (BV) into actual voltage is:

v = BV x 0.009768 + 1.221 (Volts)

The transmission attributes field (XA) reports at present only the transmit power level of the original event packet at the Tag (which is a number from 0 to 7).

The event type byte (ET) is a concatenation of two values. The more significant nibble (4 bits) stores the device type of the event originator. In Alphanet 1.5 those types are as follows:

0 - unused

1 - AP320

2 - AP321

3 - CHRONOS (black)

4 - CHRONOS (white)

5 - WARSAW

6 - AP319

7 – AP331

In a production network, only types 1, 6 and 7 are valid. The less significant nibble of ET returns the event number. In Alphanet 1.5, this number is 1 through 6 for an actual triggered event, or 0 for the periodic heartbeat event. Loop-triggered events from AP331 are reported as button #2.

For expected migration of Bruges network from 1.0 to a current version, the 1.0 device types are allowed: 3 – 1.0 AP319, 4 – 1.0 AP320, reported and should be handled by the migrating OSS.

The sequence number carried in the SN field is an 8-bit counter, local for the Tag, incremented by 1 for every new event generated by the Tag.

In Alphanet version 1.8, every event carries at least six argument bytes looking like this:

0

2-4

DS

2-4

XI

5

DS

0

1

DS

1

Where the bytes labeled DS0, DS1 and DS2-4 are device specific, and XI (Extended Information) indicates the type of additional information (if any) appended to the event report. If present, that information directly follows the XI byte.

For AP319 (device type 6), AP320 (1) and AP331 (7) the device-specific bytes look as follows:

DS

0

n

DS

1

try

g

dial

7

6

4

3

1

0

DS

2-4

lid(3)

The most significant bit of DS0, if set, indicates that the Peg that has received the event packet from the originating Tag did not acknowledge the event to the Tag. This means that the event is not global and the Tag-button pair does not occur on the Peg's association list (see section 4.1.3). In other words, a Peg receiving an event packet from a Tag does acknowledge the packet to the Tag (and n is cleared), if and only if the event is global or the Tag and the event are associated with the Peg.

The "try" field contains the retransmission count of the event packet at the Tag (for the first attempt the value is 0). The least significant bit of the first byte, g, indicates whether the event is global. Bits 1-3 are unused. For AP319, the second byte (DS1) returns the value set up with the rotary dial. For AP320, the value is always zero. For AP331, it contains the least significant byte of the loop id last entered or zero if none. DS2-4 contains the remaining bytes of the loop id for AP331, zeroes for other tag types.

For completeness, here are the values returned in DS0 and DS1 for the remaining device types:

CHRONOS (both versions, device types 3 and 4):

DS0 - the number of seconds elapsing from the last motion event

DS1 - the number of motion events counted from the previous report

DS2-4 - zeroes

WARSAW (device type 5):

DS0 - a random value

DS1 - a counter incremented by 1 with every new event

DS2-4 - zeroes

AP321 (device type 2):

DS0 - fixed value 9

DS1 - fixed value 4

DS2-4 - zeroes

The XI byte is used at present (Alphanet 1.8) to indicate whether the event report includes a piggybacked location reading, and it can contain one of three values:

0 – no location data is present in the report (the report carries no more bytes)

1 – full location data is present (the XI byte is followed by a location reading)

2 – short location data is present (the XI byte is followed by a location reading)

We anticipate refinements before version 2.0, so the function of XI may be extended.

If XI equals 1 or 2, the byte is immediately followed a location reading in this format:

2-byte reference id (LREF)

8 (short, XI = 2) or 32 (full, XI = 1) single-byte RSSI values (LRSS)

Thus, the total length of the piggybacked location reading (following the XI byte) is 10 or 34 bytes, for short and full location data, respectively. LREF lets the location server bundle the readings related to the same burst and arriving from different Pegs. At present, it is a short timestamp of the location burst (from the Tag), i.e., the number of seconds since reset modulo 64K. Note that it is consistent with Ref from the dedicated location burst report (cf. Appendix B). In fact, the piggybacked reading looks like a location report without the first two items (the Peg Id and the Tag Id) which are included in the event report.

LRSS is a series of RSSI readings (unsigned 8-bit values) from the location burst, at present 4 packets on all eight (0 through 7) power levels for full data, or 8 averaged readings on all power levels for short data.

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

The report passes a relayed sequence of bytes (see CMD\_RELAY) to the Tap.

Payload layout:

reprelformat.wmf

The TP field contains the opcode (one of 0x41, 0x42, 0x43) of the specific relay command used by the sender. SourceId is the Node Id of the relaying Peg.

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

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

Payload layout:

fwpayload.wmf

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.225

|  |  |
| --- | --- |
| **Neighborhood** | |
| **Symbolic** | **RepType** |
| REP\_NHOOD | 0xE2 |

Reports the neighborhood, with parameters set by triggering CMD\_NHOOD. Layout:

2

SND

HOC

1

Ref

1

RSB

1

RSF

1

SND: Responding peg (induced by CMD\_NHOOD).

Ref: Reference from CMD\_NHOOD.

HOC: hop count from SND to the node producing this report.

RSF: Forward RSSI – of original cmd received by SND

RSB: Backward RSSI – of reporting msg received by the node producing this report.

There are parameter variants with potentially useful interpretation of the data, for advanced debugging. However, only the simplest form, with HOC=0, the node that directly receives CMD\_NHOOD shows its immediate neighborhood, with RSSIs in both directions. Similarly, Custodian collocated with command executing peg. Sanity of other variants’ interpretations depends on the operator. The main points: RSF and RSB come from the ‘border nodes’ and nonzero HOC may produce data that don’t describe any locality.

The 0xD? opcode family is reserved for debugging reports.

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

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

lgpayload.wmf

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 single-byte item type and item is the type-specific element of the message (whose length may vary depending on IT). 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 approaching 2.0, with the new OSSI. At the moment, we’re at 1.8.

# How far from 2.0?

* Some commands / responses are not implemented.
* Master is fixed at Node Id 1.
* Master change must be reworked for multiple masters (that includes the Eye Butler project).

# Testing, quick start, illustrations, divagations

VUEE configuration for ‘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.) Read the uart\_plugin document in DOC). VUEE configuration in PIP should read as:



Compile the project for VUEE and run it. Open the "UART (plug)" window for node 1 (the master). Enter:

control -nodeid 1 -confirm both -quiet off -dump 3 -show

to tell the plugin the Node Id of the tag and set some parameters: to request ACKs (on both levels) for all OSS messages going to the Peg and to see all acknowledgments and whatever is there to see. The above settings for -confirm, -quiet, and -dump are now the defaults, but it doesn't hurt to make sure (and see them with -show).

**CMD\_GET**

Now enter:

getparams -nodeid -master -esn

You will see this in response:

-->: [81 01 00 01 80 01 00 01 03 02]

-ACK: OK nid=1 seq=1

<-A: [81 01 00 00 00]

-RSP: OK nid=1 ref=0 getparams: nodeid=1 master=1 esn=baca0001

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

The lines starting with -->, <-A, <-O are dumps of the complete sequences[[13]](#footnote-13) of bytes actually exchanged between the OSS and the Peg. You can switch them off with control -dump 0. You see an ACK for your command (as requested) followed by a response (which the plugin formats for you).

When you go like this:

getparams -nodeid -master -esn -ack off

the command will be issued with FG\_ACKR cleared, so you will get no ACK:

-->: [02 01 00 01 81 01 00 01 03 02]

-RSP: OK nid=1 ref=1 getparams: nodeid=1 master=1 esn=baca0001

<-O: [05 01 00 01 81 00 01 00 01 01 00 03 01 00 02 01 00 ca ba]

You can disable FG\_ACKR permanently with

control -confirm off.

Note that we have requested FG\_ACKR to be set in op\_ref as well, but it makes no difference for this command, because it is a query, so it always produces a reply.

We can try to construct the command by hand, e.g.,

send 1 1 3 2

This operation issues a framed command with the opcode 1 (the first byte), and the three bytes of command payload being 1, 3, and 2 (the remaining elements of the packet are filled automatically). You will see basically the same response as before, i.e.,

-->: [83 01 00 01 82 01 00 01 03 02]

-ACK: OK nid=1 seq=3

<-A: [83 01 00 00 00]

-RSP: OK nid=1 ref=2 getparams: nodeid=1 master=1 esn=baca0001

<-O: [06 01 00 01 82 00 01 00 01 01 00 03 01 00 02 01 00 ca ba]

There only difference is that the sequence numbers have advanced (the plugin does it automatically).

Let's try some illegal parameter, e.g.,

send 1 1 77 2

We shall see something like this:

-->: [84 01 00 01 83 01 00 01 4d 02]

-ACK: OK nid=1 seq=4

<-A: [84 01 00 00 00]

-RSP: EPAR nid=1 ref=3 getparams: nodeid=1 esn=baca0001

<-O: [07 01 00 01 83 02 01 00 01 01 00 02 01 00 ca ba]

Note that the command is positively acknowledged, but the response contains a nonzero return code (2 = EPAR). Despite the error, the correctly requested parameters are returned in the payload and presented.

Now, let us try a duplicate frame. The way to do it is to force a specific sequence number), e.g.,

send -seq 4 1 1 77 2

we will see:

-->: [84 01 00 01 84 01 00 01 4d 02]

-ACK: DUPOK nid=1 seq=4

<-A: [84 01 00 00 05]

The frame ACK status is 5 == DUPOK indicating that the previous duplicate has succeeded (at the frame level).

**CMD\_SET**

Only local host (Node Id) and TARP forwarding can be set in 1.5 (as it was in 1.0). The Node Id can’t be changed from or to 1 (reserved for the 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 the OSS (should we?) The parameters are set until the 1st error (returned only if op\_ref has the FG\_ACKR bit set).

Do this on Peg 2:

control -nodeid 2

setparam -nodeid 1

We get:

-->: [81 02 00 02 80 02 00 01 01 00]

-ACK: OK nid=2 seq=1

<-A: [81 02 00 00 00]

-RSP: EVAL nid=2 ref=0 setparams

<-O: [04 02 00 02 80 01 02 00]

So at the frame level it is OK, but the command level response is 1 (EVAL).

We can try to change the Node Id (of Peg 2) to something innocent, e.g.,

setparam -nodeid 521

We get:

-->: [82 02 00 02 81 02 00 01 09 02]

-ACK: OK nid=2 seq=2

<-A: [82 02 00 00 00]

<-N: [05 09 02 02 81 00 09 02]

Note that the frame-level ACK is still sent with the old Node Id (2), but the command-level response is rejected by the plugin. We only see its dump marked <-N, which means "different Node Id". To continue communicating with the Peg, we now have to do:

control -nodeid 521

Now when we go:

getparams -nodeid

we see the Peg's new identity:

-->: [83 09 02 01 82 09 02 01]

-ACK: OK nid=521 seq=3

<-A: [83 09 02 00 00]

-RSP: OK nid=521 ref=2 getparams: nodeid=521

<-O: [06 09 02 01 82 00 09 02 01 09 02]

**CMD\_GET\_ASSOC, CMD\_SET\_ASSOC, CMD\_CLR\_ASSOC**

Note: these commands require the Node Id to be set correctly to the actual Node Id of the Peg. The (unset) Node Id of zero will not work.

We stay at Peg 2:

control -nodeid 2

control -show

nodeid=2, echo=on, confirm=both, quiet=off, repeat=0, dump=3

The learning command illustrated below translates into a variant of 0x14 (set association, aka CMD\_SET\_ASSOC) with an invalid index (0x14):

learning -show

-->: [84 02 00 14 83 02 00 14]

-ACK: OK nid=2 seq=4

<-A: [84 02 00 00 00]

-RSP: OK nid=2 ref=3 learning: off

<-O: [07 02 00 14 83 00 02 00 14 00]

It lets us see the learning status of the Peg. We can switch the learning mode on:

learning -on

-->: [85 02 00 13 84 02 00 14 01 01 00]

-ACK: OK nid=2 seq=5

<-A: [85 02 00 00 00]

-RSP: OK nid=2 ref=4 setassoc

<-O: [08 02 00 13 84 00 02 00]

Learning –ON/OFF are setassoc commands and must conform to the 1.0 requirements about the data format, hence a dummy Node Id (0x0101).

learning -off

-->: [86 02 00 13 85 02 00 14 01 01 ff]

-ACK: OK nid=2 seq=6

<-A: [86 02 00 00 00]

-RSP: OK nid=2 ref=5 setassoc

<-O: [09 02 00 13 85 00 02 00]

Again, we follow 1.0: 0xff (to invalid index) means OFF. So: 0 – ON, 0xff – OFF, all other values are illegal. I believe we can do better in 2.0: I would leave 0x13, 0x14 to legitimate SET/GET and either have 0x16 for LEARN\_ASSOC or overload 0x15 with CLR, LEARN, … activities. In fact, we may have a single CMD\_ASSOC with various tasks. As we are planning to update this functionality anyway, these cosmetics should be part of it.

learning -on

….

learning -show

-->: [88 02 00 14 87 02 00 14]

-ACK: OK nid=2 seq=8

<-A: [88 02 00 00 00]

-RSP: OK nid=2 ref=7 learning: on

<-O: [0b 02 00 14 87 00 02 00 14 01]

Another careless inconsistency: learning ON is shown (properly) as a ‘1’ flag, but is set by ‘0’ value. OFF is shown ‘0’ as it should, but the flag is cleared with 0xff.

setassoc -from 15 -mask 0xff -tags 101 102

-->: [89 02 00 13 88 02 00 0f 65 00 ff 66 00 ff]

-ACK: OK nid=2 seq=9

<-A: [89 02 00 00 00]

-RSP: OK nid=2 ref=8 setassoc

<-O: [0c 02 00 13 88 00 02 00]

getassoc -from 15

-->: [8a 02 00 14 89 02 00 0f]

-ACK: OK nid=2 seq=10

<-A: [8a 02 00 00 00]

-RSP: OK nid=2 ref=9 getassoc: <15> tag=101/mask=ff tag=102/mask=ff

<-O: [0d 02 00 14 89 00 02 00 0f 65 00 ff 66 00 ff 00 00 00 00 00 00 00 00 00]

Note the discrepancy about the actual payload and the plugin’s formatted variant. Yet another food for thoughts for improving the association commands. However, all this should be thoughtful, as we want to accommodate quite a few scenarios in the seemingly unrelated application spaces. Just an example: perhaps a temporary optional associations between a car (tag) and parking space (peg) is an innovative feature?

setassoc -clear

-->: [8b 02 00 15 8a 02 00 00]

-ACK: OK nid=2 seq=11

<-A: [8b 02 00 00 00]

-RSP: 06 nid=2 ref=10 clrassoc

<-O: [0e 02 00 15 8a 06 02 00]

...

getassoc

-->: [8c 02 00 14 8b 02 00 00]

-ACK: OK nid=2 seq=12

<-A: [8c 02 00 00 00]

-RSP: OK nid=2 ref=11 getassoc: <0>

<-O: [0f 02 00 14 8b 00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00]

**CMD\_RELAY**

Most of the functionality is as in 1.0. There is a small (optional) extension: if ACK is requested in op\_rc and the operation is successful, RC\_OK (0) will arrive from the remote destination. This way end to end reliability is introduced. Likely, in 2.0, this will be further parameterized (no ACK, local, remote, both…).

relay -m 1 -dest 1 3 4 5

-->: [8d 02 00 42 8c 02 00 01 00 03 04 05]

-ACK: OK nid=2 seq=13

<-A: [8d 02 00 00 00]

!RSP: OK nid=2 ref=12

<-O: [10 02 00 42 8c 00 02 00]

Broadcast not allowed. Note that the ACK comes from Peg 2 (on which the command is issued), whereas we see this at the destination Peg (node 1):

-REP: relay:[mode=1,src=2] <03 04 05>

<-O: [09 01 00 ff 01 42 02 00 03 04 05]

Appendix B

Here we document any tentative or additional interface which formally does not belong to Version 1.5 or even Version 2.0, but has been added for experiments and testing.

## B.1 Location burst reports

As of August 3, 2015, we have included a special report type representing survey messages issued to the master by Pegs receiving location bursts from Tags. As forwarded in the form of a report, a location burst amounts to 32 RSSI values collected from one Tag (see the ATOLS document). Here is the layout of a burst report.

|  |  |
| --- | --- |
| **Location burst report** | |
| **Symbolic** | **RepType** |
| REP\_LOCA | 0xB1 |

The reports conveys to the OSS the 32 RSSI values collected from a Tag issuing a location burst.

Payload layout:

lrpayformat.wmf

|  |  |
| --- | --- |
| **PegId** | The identifier of the Peg that has received the burst from the originating Tag. |
| **TagId** | The identifier of the Tag issuing the burst. |
| **Ref** | The reference number (identifying a specific burst and, e.g., preventing duplicates). |
| **RSS** | An RSS value. |

Appendix C

This appendix describes RFID functionality implemented in Alphanet 1.82 and clearly belongs to a growing majority of topics which do NOT belong to *Interface* documentation or belong here partially. The intention, somewhat utopic, is to split this document into *Interface* and *Operations*. For now, we keep all here, a convenient placeholder for development curves and u-turns.

Perhaps it should be explicitly underlined that by *RFID functionality* we mean a mnemonic label for a general functionality within Alphanet, not a wild hype or reasonable standards out there.

For reasons out of scope here, a tag may trigger (and stop) an RFID beacon with flexible payload and several pertinent values. Intentionally, the beacon is relatively frequent and blind, but LBT is ON. These are proxy messages meant to be received by pegs, in current implementation only to be passed to other system components over UART.

## C.1 Operations

AP331 tags (with AS3932 circuitry and driver) sense proximity of a loop, potentially overlapping with another. Loops are distinguished by 4-byte ids offered by the driver. Hard limit of the readings is about 10 Hz. A loop represents a door and, prominently, its lock. There is a 1-1 relationship between a peg and lock; however, the peg does NOT know about it. Some other system component (Renesas) connected to the peg via UART do know the loop id and can (selectively) act on information passed from pegs. Inability to provision loop id at pegs is set as a hard requirement, otherwise the solution makes little sense.

#define SENSE\_LOOP\_331 1/0 compiles the AS3932 in/out of AP331, i.e. there are two loadable images and no choice for AP331 nodes in a VUEE run. Hereafter we assume 1.

AP331 turns AS3932 ON and after HALFTIME\_331 PicOS’s milliseconds (1024) reads the sensor. Therefore, it receives either 0 (out of loop) or 1st loop id reading after ON. It turns AS3932 OFF. The 4-byte value is stored. If it differs from the previous reading, a global alarm LOOP\_ALRM\_ID (2) is issued and the *rfid* process started or reset (for nonzero values) or signaled to stop (for zero value). After another HALFTIME\_331, the above repeates. The process runs all time.

The *rfid* process waits *ini* seconds, updates its data, sends *msg\_rfid* message and delays for (updated) *next* seconds (*next = next + inc* up to *max*). All values are easy to set, but stay constant within a tag’s image. Presently they are ini=1, inc=1,max=5, so rfid beacon goes out 1s after the global alarm and then 2, 3, 4, 5, 5, … after, until the tag is our of any loop.

All pegs that hear msg\_rfid, report it over the UART. The report format follows:

## C.2 RFID report

|  |  |
| --- | --- |
| **RFID report** | |
| **Symbolic** | **RepType** |
| REP\_RFID | 0xB2 |

Payload layout:

2

snd

btyp

1

rssi

1

1

ttyp

next

1

len

loopid

cnt

1

len

1

|  |  |
| --- | --- |
| **snd** | Tag turned rfid (node id) |
| **rssi** | Rssi the mag\_rfid was received with |
| **btyp** | Board type e.g. AP331 is BTYPE\_AT\_LOOP (7) |
| **ttyp** | Tag can assume multiple rfids, e.g. RFID\_TYPE\_LOOP (1) |
| **next** | Next message is expected in that many seconds |
| **cnt** | Counter ++, wrapped at 255 |
| **len** | That many bytes of rfid payload follow (4 for loop id) |
| **loopid** | Loop id |

## C.3 RFID testing in VUEE

Although loops don’t exist in our models, any AP331 node can be agitated in a way similar to AS3932 by changing values of Sensor 1 (in its SENSOR panel).

1. Run VUEE with alphatest.xml. Open SENSORS, EMUL for node 701. Open UART (plug) for 39525. Open UART (plug) for 47818 (the master). Connect PANEL, click ‘On’ for 701. You may close PANEL.
2. UART on 39525 shows the global heartbeat (but=0) from 701. UART on 47818 shows it forwarded from several pegs. EMUL on 701 shows (faked) AS3932 cycling ON / OFF (if #define SENSE\_LOOP\_331 in ap331\_tag.cc:17 is set to 1 – we assume it is).
3. Move Sen 1 slide to a nonzero value. Global loop alarms (but=2) should be visible on UARTs on 39525 and 47818.
4. On 39525, 1 second after (note that it may precede global alarm or global alarm may even be missing) local RFID reports start appear with intervals 2, 3, 4, 5, ... seconds (there will be a plugin update for them). Sen 1 moved to another value resets the loop, moved to 0 stops the reports. Any change generates global loop alarm as well. Note that every time global alarm is issued, EMUL on 701 shows uptime and memory stats, handy to dispel suspicions about memory leaks.

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 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. Note that to acknowledge a request one has to know with confidence that the request's sequence number has been recognized correctly. [↑](#footnote-ref-7)
8. I.e., a non-ACK frame (obviously, ACKs are never acknowledged). [↑](#footnote-ref-8)
9. This feature need not be taken advantage of in Alphanet 2.0, but it is included as an option. [↑](#footnote-ref-9)
10. At least this possibility is formally available. [↑](#footnote-ref-10)
11. Probably, to simplify things, it should be ignored. [↑](#footnote-ref-11)
12. This doesn't mean that we want to have GPS modules in Alphanet Tags. [↑](#footnote-ref-12)
13. Minus the lowest-level encapsulation: STX, ETX, checksum, DLEs. [↑](#footnote-ref-13)