Olsonet Communications

UART Communication via VNETI (TCV)

June 2016

1 Introduction

This document describes the options available for implementing VNETI style communication over the UART. The idea is to allow the praxis to take advantage of VNETI functions, which are normally used for networking. Note that traditionally the UART is visible to the praxis as a PicOS *device* accessible via the **io** operation (see the *PicOS* document). That access typically involves library functions **ser_in**, **ser_out**, and their derivatives. As at the present state of PicOS evolution UART is practically the only device using that interface, I contemplate removing it altogether. Even if there is a need for some kind of light-weight access to the UART (without involving VNETI), such access can be provided via a simpler interface than the rather messy **io** concept, which, in the UART case, is an overkill.

There are four basic options for UART-over-VNETI selectable at compile time (via settable constants), one option occurring in two flavors. If more than one UART is present in the system, then the option affects both (all) UARTs at the same time. This isn't a fundamental issue: one can think of making the options selectable on a per UART basis, possibly even dynamically, e.g., with **tcv_control** calls. At this time, I wanted to avoid introducing a complexity that might never be needed (note that it would inflate the code size and require extra RAM locations). The three options can be further augmented by library functions and/or VNETI plugins.

To specify that the UART(s) will be handled by VNETI interface, you should declare:

```
#define UART_TCV        n
```

where ***n*** is greater than zero (typically 1 or 2) and stands for the number of UARTs to be visible in the system. Note that such a declaration is incompatible with the traditional declaration of a UART device with:

```
#define UART_DRIVER     n
```

with nonzero ***n***. Thus, it is impossible to have, say, one UART appearing as a traditional device, and the other handled by VNETI. Any of the above two declarations (with nonzero ***n***) determines 1) how many UARTS there are in the system, 2) how they (ALL) are going to be handled. As stated earlier, the second declaration type is marked for removal. There is no rush, especially that declaring **UART_DRIVER 0** completely eliminates the interface from the code. Thus it may stay forever as an option while having been eliminated for all practical purposes.

Assuming that **UART_TCV** is nonzero, i.e., we are going to communicate with the UART over VNETI, one of the possible modes for this communication is selected by setting **UART_TCV_MODE** to:

UART_TCV_MODE_N	(0)	- simple packet mode (Section 3)
UART_TCV_MODE_P	(1)	- persistent packet mode (Section 4)
UART_TCV_MODE_L	(2)	- line mode (Section 5)
UART_TCV_MODE_E	(3)	- escape mode (Section 6)
UART_TCV_MODE_F	(4)	- an alternative variant of the escape mode

The first mode is the default (it is selected automatically if you do not set **UART_TCV_MODE** explicitly, but you do set **UART_TCV**). It essentially treats the UART as a (raw) networking interface. The second mode uses a different packet format and provides for automatic (internal, i.e., carried out at the PHY level) acknowledgments intended to make the communication reliable. The third mode can be used to send/receive straightforward ASCII-oriented data. It is intended as the simplest replacement for the traditional (ASCII) UART interface. The fourth and fifth modes implement simple STX-ETX-DLE serial packetization schemes.

There exist two additional options to set up reliable (data-link) communication over (in principle) any RF-type channel, where packets can be lost, but the act of receiving a single packet is assessed reliably, meaning packet integrity is verified with CRC code. These two options are described further in this document as XRS (the so-called eXternal Reliable Scheme – Section 9) and BOSS (the Binary OSS plugin – Section 10). Even though both options can be conceivably applied to PHYs other than the UART, their descriptions belong to this document as they are primarily intended (and make most sense) for UART communication. They supplement mode N (**`UART_TCV_MODE_N`**) by making sure that all sent packets are in fact received. Although mode P (**`UART_TCV_MODE_P`**) was originally intended for that purpose (with the reliability built into the PHY), XRS and BOSS are advantageous in some circumstances.

For example, the primary and fundamental difference between XRS (on top of the N mode) and P mode is that with the latter the reliability is implemented at the PHY level, whereas XRS imposes the reliability layer on top of the TCV (VNETI) API. For one thing, it means that in the P mode an unaware praxis can easily use up the buffer space of VNETI by sending packets blindly while the other party is not listening. This is because such packets are accepted (buffered) by VNETI and blocked at the PHY level (awaiting acknowledgments). With XRS (and BOSS alike), there can be at most one outstanding packet (handled by VNETI), which puts the least possible strain on VNETI's buffer space. Additionally, XRS provides functions resembling the touch and feel of the traditional UART functions, especially the ASCII-oriented ones, which may make the transition from the old interface a bit easier.

The BOSS plugin, in contrast to XRS, hides the implementation of reliability under VNETI's hood modifying the semantics of the standard API (as plugins do). Its primary advantage over XRS (and over mode P as well) is in the provision of two different types of packets: one for reliable (acknowledged) traffic, the other for spontaneous unacknowledged packets. Thus it offers both communication paradigms at the same time. Moreover, at most one unacknowledged packet (of the reliable kind) can be queued at a time. Thus, regardless of the praxis behavior, the scheme avoids the buffer space overflow problem of the P mode.

For some time I was contemplating adding a feature to the P mode whereby urgent packets (as understood by VNETI) would be sent “out of band” and without acknowledgments. That work has been put on a back burner for now. We still have to see whether the P mode finds its way into praxes.

I have removed from this document some loose and overly blanket comments about the possible advantages and disadvantages of the various schemes for OSS communication. That speculative discussion belongs elsewhere.

2 Initialization

The function to initialize the UART PHY looks the same for all five cases:

```
phys_uart (int phy, int mbs, int which);
```

where **`phy`** is the PHY ID to be assigned to the UART, **`mbs`** is the reception buffer size, and **`which`** selects the UART (if more than one UART is present in the system). When there is only one UART, the last argument must be zero.

Having initialized the PHY and configured a plugin for it (say the NULL plugin), the praxis can use the standard operations of VNETI for sending and receiving messages (**`tcv_open`**, **`tcv_wnp`**, **`tcv_rnp`**, and so on).

Standard UART configuration constants apply to the UART PHY. In particular, **`UART_RATE_SETTABLE`** can be set to 1, in which case the praxis will be able to reset the UART bit rate via a **`tcv_control`** request (**`PHYSOPT_SETRATE`**). This applies to all three modes.

The interpretation of **mbs** (the buffer length parameters) is slightly different for the different modes.

2.1 The N mode

For mode 0 (N), the length argument must be an even number between 2 and 252, inclusively. You can also specify zero, which translates into the standard size of 82 [these defaults are pointless and should be probably removed].This is the size in bytes of the packet reception buffer including the Network ID field, but excluding the CRC. Thus, the maximum payload length is **mbs**−2 bytes, assuming that Network ID is used for its original purpose, or **mbs** bytes, if it isn't (as is normally the case). For UART communication, Network ID is not needed (the PHY can be instructed to ignore it), so it can be considered part or the payload.

Unless **UART_TCV_MODE_NOCHECKSUM** is defined as nonzero, the actual packet provides two bytes for the CRC at the end (both on input and on output). Thus, for example, when requesting a new packet (via **tcv_wnp** under the NULL plugin) make sure to reserve two extra bytes. When **UART_TCV_MODE_NOCHECKSUM** is zero, no CRC is calculated and the packet perceived by the praxis is exactly the payload + the (optional) Network ID.

XRS uses the first four bytes from the payload, including the two bytes of Network ID, which leaves mbs-4 bytes for the user. BOSS uses just the Network ID (two bytes) for its header.

2.2 The P mode

For mode 1 (P), **mbs** must be between 1 and 250 (zero stands for 82, as before). Note that it can be odd. The specified length covers solely the payload. No Network ID field is used (the mode assumes two parties connected via a direct link), and the (standard) CRC field is extra. While formally the payload may consist of an odd number of bytes, the actual length of a packet carrying it will be rounded up to an even number of bytes – to facilitate CRC calculation. The received payload length communicated to the praxis (VNETI) in such a case will reflect the correct odd number of bytes (i.e., the stuffed byte will be ignored upon receipt).

In contrast to mode N, when requesting a new packet (with the NULL plugin) there is no need to provide for the CRC. The requested length must be exactly equal to the length of the payload.

2.3 The L mode

For mode 2 (L), the specified buffer length also applies to the payload. There is no Network ID field and there is no CRC. Unlike the first two modes, mode 2 does not assume any protocol on the other end: the party receives and is expected to send ASCII characters organized into lines. Thus, you can directly employ a straightforward terminal emulator at the other end. An odd number of bytes is handled correctly and without any tricks.

2.4 Modes E and F

For modes 3 and 4 (E and F), the specified buffer length (which must be between 1 and 254) exactly corresponds to the payload length. Similar to mode 2 (L), there is no Network ID field. While the modes implement simple single-byte parity/checksum control schemes, the parity/checksum byte does not count to the specified buffer length. The entire payload is usable by the application program and the system never modifies any of its bytes.

In contrast to typical RF PHYs, which start in the OFF state, all the packet modes of UART start active, i.e., you don't have to switch them ON via `tcv_control`. They can be switched off, though, as described later.

Now for some operational details.

3 The simple packet mode (`UART_TCV_MODE_N`)

With this option, the UART attempts to emulate a networking interface. The packets sent and received by the praxis have the same layout as RF packets (see Figure 1). In particular, the concept of Network ID is formally retained (by default), which, theoretically, enables multiple nodes to communicate over a shared serial link emulating a broadcast channel.

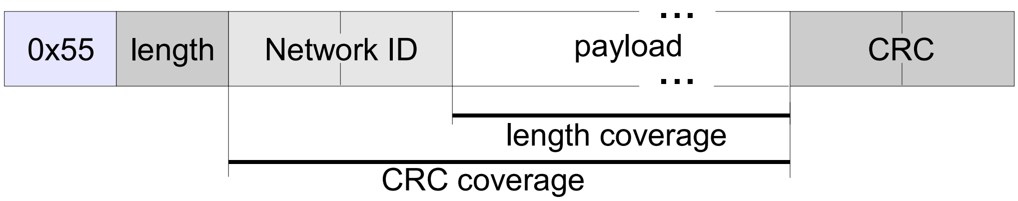

Figure 1: N-mode packet layout

Similar to a typical RF interface, the packet length is supposed to be even. You will trigger a system error trying to send a packet with an odd number of bytes. All received packets are guaranteed to have an even length.

The full format of a packet, as sent over the UART or expected to arrive from the UART, is shown in Figure 1. The payload always occupies an even number of bytes, so the CRC field is aligned at a word boundary.

The packet starts with a byte containing 0x55 (letter "U"), which can be compared to the preamble in a true RF channel. Having detected the 0x55 byte, the receiver reads the next byte, adds 4 to its unsigned value, and expects that many more bytes to complete the packet. The first two bytes, i.e., the preamble and the length byte are then discarded (they are not stored in the reception buffer).

To detect corrupted packets, the receiver first validates the sanity of the length byte: it must be even and not greater than the maximum payload size (declared with `phys_uart`) to trigger a reception. Moreover, there is a limit of 1 second (constant `RXTIME` in `PicOS/uart.h`) for the maximum time gap separating two consecutive byte arrivals. If an expected character fails to arrive before the deadline, the reception is aborted, and the receiver starts looking for another preamble.[1]

The last two bytes of every received packet are interpreted as an ISO 3309 CRC code covering the payload and Network ID. This means that the checksum evaluated on the complete packet following the length byte and including the checksum bytes should yield zero. The checksum bytes are appended in the little-endian order, i.e., the less significant byte goes first. If the CRC verification fails, the packet is discarded by the PHY.

The rules for interpreting the Network ID are the same as for, say, CC1100. As they are never written explicitly, let us spell them out [this note may find its way into some other, more pertinent document some day].

[1] This timeout is way too long for most applications. It used to be much shorter (and there is no sane reason why it should be so long), but the Windows implementation of UART (COM ports) introduces occasional long hiccups before buffered characters are expedited over the wire. That caused problems in the Heart Datalogger project and Seawolf OSS tests.

3.1 Rules for interpreting Network ID

Transmission:

If the node's Network ID (settable with **`tcv_control`**) is **`WNONE`** (**`0xFFFF`**), the packet's ID field is not touched, i.e., the driver honors the value inserted there by the praxis. Otherwise, the current setting of the node's Network ID is inserted into the packet's field before the physical transmission takes place (also if the node's Network ID is zero).

Reception:

If the node's Network ID is different from 0 and **`WNONE`**, then the packet's field must match the node's ID or be equal 0 for the packet to be received. If this is not the case, the packet is dropped. If the node's ID is 0 or **`WNONE`**, the packet is received regardless of the contents of its Network ID field. In any case, the received packet's ID field is never modified by the driver.

Note that by setting the node's Network ID to **`WNONE`**, you effectively disable the interpretation of the packet's Network ID, which means that that field is treated as part of the payload.

4 The persistent packet mode (`UART_TCV_MODE_P`)

The idea is that the sequence of packets exchanged between the node and the other party has the appearance of a reliable stream of data. The packet format is similar to the one shown in Figure 1, except that the preamble character is different and can take one of four values. Specifically, all bits in the preamble byte are zero except for the two least significant ones denoted **CU** (bit 0) and **EX** (bit 1).

The exchange is carried out according to the well-known alternating bit protocol. **CU** is the alternating bit of the packet, i.e., it is flipped for every new packet sent over the interface. The initial value of the bit (for the first packet) is 0. **EX** indicates the expected value of the **CU** bit in the next packet to arrive from the other party. The initial value of **EX** is also 0.

Having sent a packet with a given value of **CU**, the peer should refrain from sending another packet, with the opposite value of **CU**, until it receives a packet with the appropriate value of the **EX** bit, i.e., opposite to the one last-sent in **CU** (indicating that a new packet is now expected). The peer should periodically retransmit the last packet until such a notification arrives.

Having received a “normal” packet (i.e., other than a pure ACK – see below), the peer should acknowledge it by setting the **EX** bit in the next outgoing packet to the pertinent value, i.e., the inverse of the **CU** bit in the last received packet.

If the peer has no handy outgoing packet in which it could acknowledge the receipt of a last packet that has arrived from the other party, it should send a “pure ACK,” which is a packet with payload length = 0. Such a packet does not count to the normal sequence of received (data) packets, i.e., it should *not* be be acknowledged. Its **CU** bit should be always 1, and its **EX** bit should be set to the inverse of the **CU** bit in the last received data packet. Pure ACK packets include the checksum bytes as for regular packets. As there are only two legitimate versions of pure ACK packets, their full formats (together with checksums) are listed below:

EX = 0 **`0x01 0x00 0x21 0x10`**
EX = 1 **`0x03 0x00 0x63 0x30`**

The recommended message retransmission interval is 1 sec. The ACK should arrive within 0.5 sec after the packet being acknowledged has been completely received.[2]

[2]Needless to say, these “recommendations” are completely accidental. The optimum settings may depend on many things, notably on the UART rate.

Note that regardless how a peer decides to initialize its values of **EX** and **CU** (in particular after a reset, or after turning the interface off and back on), the other party will always know which packet (in terms of its **CU** bit) the peer expects. This is because both parties are supposed to persistently indicate their expectations in the packets they transmit, be they data packets or pure ACKs. However, depending on the configuration of the EX/CU bits at the time of disconnection, after a reconnection, one message (from each end) may be lost. For example, if the EX bit at party A is 1 and party B starts with CU = 0, the first message received by A from B will be deemed a duplicate of a previous message. An OSS program (re) connecting to a praxis can get around this problem by delaying its own transmission until it receives a message from the praxis. If this message is a periodic (empty) ACK, the program will learn the expected value of its CU bit for the first outgoing packet. If this is a (nonempty) message, the program can accept it unconditionally and properly set the value of its EX bit. The only potential for a loss is when the praxis suddenly creates a message after sending the ACK. Note that such a message will always be sent by the praxis at least once. The only risk is that it will be lost the first time around and then never retransmitted because of the wrong setting of EX in the first message send by the OSS program.

One can think of a safe re-connection protocol, e.g., a special request sent by the OSS program to the praxis to identify its EX and CU. We could also make sure to include the actual value of CU in an ACK (instead of setting it permanently to 1 – which simplifies generating ACKs by reducing their population). Then, the role of a reply to the special request could be simply played by an ACK. This may come later if found useful.

The above discussion also applies to XRS (Section 9) and BOSS (Section 10), which suffer from the same problem. A solution postulated for OSS communication is to have a special message (a variant of “no operation”) to be sent by a peer (any of the two) suspecting that the synchronization may have been lost (reconnection, reset, restart). A foolproof approach involves two variants of such a message, say number 1 and number 2. The re-connecting node will send both messages, one after another, and the other party will receive either both of them (having received message 1, it will know that message 2 is to follow), or just message 2 (if the first one was discarded as an apparent duplicate). Sending twice the first message would be probably enough, but having two different messages improves the clarity of perception. The other peer, having received message 2, can reply with message 2 of its own.

5 Line mode (`UART_TCV_MODE_L`)

With this mode, packets transmitted by the praxis are transformed into lines of text directly written to the UART. Also, characters retrieved from the UART are collected into lines which are then received as VNETI packets.

The length of an outgoing packet can be any number, including zero. The payload of such a packet is expected to contain a string terminated with a **`NULL`** byte. The **`NULL`** byte need not appear if the entire length of the packet is filled with ordinary characters.

Such a packet is interpreted as one line to be written to the UART “as is”. The LF+CR characters should not occur in the packet: they will be inserted by the PHY.

For reception, the PHY collects characters from the UART until LF or CR, whichever comes first. Any characters with codes below **`0x20`** following the last end of line are discarded. This way, empty lines are not received (even though they can be written out). Having spotted the first LF or CR character, the PHY terminates the received string with a **`NULL`** byte and turns it into a packet. Note that there *are no* Network ID or CRC fields.

If the sequence of characters would constitute a line longer than the size of the reception buffer, only an initial portion of the line is stored (the outstanding characters are ignored). In all cases, the PHY guarantees that the received line stored in the packet is terminated with a **`NULL`** byte. This is to make sure that the packet payload can be safely processed by line parsing tools (like scan) which expect the **`NULL`** character to terminate the string.

6 Escape modes (**`UART_TCV_MODE_E`** and **`UART_TCV_MODE_F`**)

These two modes are identical, except for the way of calculating the "integrity" (parity/checksum) byte. A packet begins with the special character STX (ASCII code 0x02) and terminates with ETX (ASCII code 0x03). Immediately preceding ETX is a single integrity byte. For mode E, this is a simple parity byte calculated as an XOR of all bytes in the packet, including the STX and ETX brackets (the XOR of all bytes, including the STX and ETX brackets plus the parity byte should yield zero). For mode F, the byte is the negated arithmetic sum of all the remaining bytes (the arithmetic sum of all bytes, including the STX and ETX brackets plus the checksum byte should yield zero).

Everything following the starting STX character and preceding the integrity byte is assumed to be the payload. One more special character, DLE (ASCII code 0x10) is used for escaping special characters within the packet. If any of the bytes between STX and ETX (i.e., any byte of the payload or the integrity byte) looks like STX, ETX, or DLE, then it is escaped by preceding it with DLE (ASCII code 0x10). For example, this sequence of bytes:

```
0x55 0x57 0x02 0x13 0x12 0x10
```

will be encoded as:

<u>`0x02 0x55 0x57 `</u>

```
0x02 0x55 0x57 0x10 0x02 0x13 0x12 0x10 0x10 0x03
```

The escape characters have been underlined. Note that the integrity (parity) byte also had to be escaped.

Note that the DLE escapes are ignored while calculating the parity/checksum.

The maximum payload length (escape DLE's excluded) is determined by the **`mbs`** argument of **`phys_uart`** (Section 2.4). The maximum value of that argument is 254.

Spurious escapes, i.e., escaping bytes that need not be escaped, are OK: the escape DLE's are removed and the escaped bytes are interpreted correctly.

7 Options

The list of options settable and testable with **`tcv_control`** includes switching on and off the receiver end of the connection. Note that the transmitter is never switched off (a packet appearing in the queue is always expedited), except for mode P which requires the receiver for transmission (in that case, switching off the receiver will also block the transmitter's queue). Also note that when XRS (or BOSS) is used on top of mode N, switching off the receiver will disable acknowledgements and thus (logically) block the transmitter.

In contrast to a typical RF module, the receiver is initially on (i.e., the module is fully operational). The concept of switching the interface on and off works similar to a typical RF interface. By switching off the receiver, i.e.,

```
tcv_control (sess, PHYSOPT_RXOFF, NULL);
```

you disable reception, but leave the transmission end available to any outgoing packets that may appear in the output queue. In other words, the transmission end of the interface is activated on demand. Note that these days **`PHYSOPT_RXOFF`** is equivalent to **`PHYSOPT_OFF`** (and also **`PHYSOPT_RXON`** is equivalent to **`PHYSOPT_ON`**) which means that switching off the interface is interpreted as switching off its reception end only.

Notably, for mode P, you cannot switch off the receiver while leaving the transmission end open, as it makes no sense to send a packet, if you cannot receive an acknowledgment for it. Consequently, **`PHYSOPT_RXOFF`** for mode P will also block (hold) the transmission causing the output queue to accumulate (equivalent to the now obsolete **`PHYSOPT_RXOFF`** + **`PHYSOPT_TXHOLD`**). Note that explicit **`PHYSOPT_TXHOLD`** is not applicable to any mode (and is always ignored). The remaining two legacy requests, **PHYSOPT_TXON** and **PHYSOPT_TXOFF**, are also ignored, except in P mode where they are used to toggle the so-called *active discipline.* By default, the discipline is turned off. When switched on (by **PHYSOPT_TXON**), the PHY will be issuing ACKs every second, even if it has nothing else to say. Normally, at least one side of a P mode connection should be doing this for reliability. In most cases, it makes better sense to assign this responsibility to the OSS program.

The remaining options are:

PHYSOPT_STATUS

Returns the module's on/off status: two bits (0 – receiver on, 1 – transmitter on). For the P mode, the single bit number 0 tells the global status of the PHY.

PHYSOPT_SETSID / **PHYSOPT_GETSID**

Only available for mode N. Sets or retrieves the Station ID assigned to the PHY. In most cases, the Station ID is set by the praxis to **WNONE** at the very beginning in order to recycle the Station ID as part of the payload (see Section 3.1). This is in fact required for XRS and BOSS. BOSS, being a plugin, does it internally, but for XRS you have to do it explicitly when initializing the session (see Section 9.1).

PHYSOPT_SETRATE / **PHYSOPT_GETRATE**

Available in all modes, if settable UART rate (**UART_RATE_SETTABLE**) has been compiled in.

PHYSOPT_GETMAXPL

Returns the maximum packet length as specified with **phys_uart** (see Section 2) when the module was initialized.

8 Diag messages

All four modes a way of accommodating diag messages that must be directly expedited to the UART as some kind of out of band data. Note that when the UART is controlled by a packet-driven interface (guarded with checksums – modes N and P), a diag message arriving out of band will appear as garbage to the other party and, in full accordance with the protocol, will be ignored. Also note that to make any sense at all, diag messages must in fact be sent out of band (they cannot pass through the same buffered and packetized interface as “normal” packets), as their role is to diagnose problems (so they must be written to the UART directly and at once).

If **DIAG_MESSAGES** is nonzero and the interface operates in mode N, P, E, or F the action of dispatching a diag message is carried out as follows:

1. If the transmitter is running (i.e., a packet is currently being sent to the UART), the driver aborts that packet (i.e., stops the transmission immediately).
2. For modes N and P, a sequence of **DLE** characters (code **0x10**) is sent to the UART. If the transmitter wasn't found running in the previous step, that sequence consists of four characters. Otherwise, the number of characters is equal to transmitter buffer size + 8. For mode E, four ETX bytes are sent to make sure that the receiver recognizes that the previous packet has ended. Then, they are followed by four consecutive DLE's.
3. The diag line is written to the UART as a straightforward ASCII string terminated with LF+CR.

The idea behind step 2 is to tell the recipient in the clearest possible way that there is a diag message coming in and avoid losing that message, e.g., as part of a packet that would be diagnosed as corrupted. If the transmitter is inactive, the code **0x10** will be interpreted as special by the recipient (note that it is different from a packet preamble). If the transmitter has been transmitting a packet, that packet will be aborted and a sufficiently long sequence of special characters will be generated to violate the length limitations at the receiver and thus force it to abort the reception.

Needless to say, the receiver (the OSS program) should be prepared to handle such diag messages (or ignore them as random errors).

In mode L, the diag action is simple, as a diag message never interrupts any “packet” being written to the UART as an atomic chunk, but, in the worst case, it may interfere with (mess up) a message being written out by the PHY.

9 The eXternal Reliable Scheme

This scheme, dubbed XRS in the sequel, is set up by initializing the UART in mode 0 (**`UART_TCV_MODE_N`**) with the NULL plugin, and then invoking a special process to implement a version of the alternating bit protocol at the session level. Two sets of API are provided: string oriented (assuming that the data items passed between the two parties are zero-terminated character strings), and binary. Both variants can be used in the same praxis. Also, they both work with VUE[2].

Note that XRS is not restricted to UART (it can be used over any “session”, which, in particular, can represent an RF link). However, as its purpose is to implement a reliable, strictly P-P link, UART interface to an OSS program looks like the most natural application. The API functions provided by XRS are intended to act as a complete replacement for the **`ser_out`**/**`ser_in`** family (for the traditional pure-ASCII UART interface).

A praxis that wants to take advantage of XRS should include **`ab.h`** or/and **`abb.h`** in its header. The first file defines the requisite function headers for the string-oriented interface, the second one is for the binary variant. The modules implementing the scheme can be found in directory **`PicOS/Libs/LibXRS/`**.

9.1 Functional description

The initialization is the same for both interface types, i.e., string-oriented and binary. Here is a typical initialization sequence to be issued from the praxis:

```
#include "sysio.h"
#include "plug_null.h"
#include "phys_uart.h"
#include "ab.h"
#include "abb.h"
...
        phys_uart (0, 96, 0);
        tcv_plug (0, &plug_null);
        SFD = tcv_open (WNONE, 0, 0);
        if (SFD < 0)
                syserror (ENODEVICE, "uart");
        w = WNONE;
        tcv_control (SFD, PHYSOPT_SETSID, &w);
        ab_init (SFD);
...
```

Note that you will normally include only one XRS API header (i.e., either **`ab.h`** or **`abb.h`**), unless you in fact want to use both interface types simultaneously.

The specific necessary initialization steps consist in 1) setting the network ID of the PHY to **`0xFFFF`** (**`WNONE`**) and 2) executing **`ab_init`** with the session ID passed as the argument. The first step is required to make sure that the PHY never interprets or sets the ID field in processed packets, as this field will be used by XRS. The second step starts a special process, which will be responsible for handling all input/output for the session. The praxis should now refrain from referencing the session ID directly. Instead, it should take advantage of the respective XRS API functions.

In addition to **`ab_init`**, one more function shared by both interface types is:

```
void ab_mode (byte mode);
```

which sets the mode of operation of XRS. The argument can be:

`AB_MODE_OFF (0)`

The interface is switched off, which means that nothing will be transmitted (the output end of XRS will be blocked. In particular, the interface will appear unavailable to **`ab_out`**, **`ab_outf`**, and **`abb_out`** (the functions will block until the mode changes). Any input arriving over the PHY will be absorbed and discarded (to avoid overflowing VNETI's buffer space). Note that this refers solely to the activity of XRS (i.e., the special process created with **`ab_init`**), and has nothing to do with the driver (PHY) state (the function doesn't invoke **`tcv_control`** on the session ID). One reason why the praxis may want to execute **`ab_off`** is to temporarily assume a different protocol on the session ID (e.g., OEP).

`AB_MODE_PASSIVE (1)`

This is the default mode assumed immediately after **`ab_init`**. The node's end of the protocol will behave passively, meaning no polling if the node has nothing to send and is not bothered by the other party. This way, the implementation of reliability is delegated to the other party, which will have to poll the node whenever it wants to make sure that the node is still alive. This is the recommended mode for a node concerned about power usage, e.g., one that goes into deep sleep where periodic polling over the UART may be too costly or inconvenient.

`AB_MODE_ACTIVE (2)`

In this mode, the node will be sending short polling messages (every two seconds), even if it has no outgoing data packet to send (in the full spirit of the alternating-bit protocol). This mode can be used when the node itself plays the master role (e.g., the other party uses the passive variant of the protocol). Note that two active ends are compatible, as are one passive and one active end. However, if both ends are passive, they may get into a stall if a packet gets lost or damaged.

9.1.1 String-oriented functions

These functions require the **`ab.h`** header.

```
char *ab_outf (word st, const char *fm, ...);
```

This function counterparts the old **`ser_outf`**. It writes to the interface a formatted output line. The expected number and type of parameters following the format string **`fm`** are determined by the sequence of special (formatting) fields found in that string (see **`PicOS.pdf`**). The first argument is the state to retry the function, if it cannot be completed immediately.

Upon success, the function returns a pointer to the encoded string that has been queued for transmission. If the state argument (**`st`**) is not **`WNONE`** (i.e., it refers to a normal state), the function either returns successfully or does not return at all. The latter case occurs when the interface is busy or there is no memory to accommodate the output buffer: the function will then block resuming at state **`st`** when it makes sense to try again.

If the state argument is **`WNONE`**, the function always returns. Then if its value is **`NULL`**, it means that the operation has failed.

If the output produced by the function exceeds the buffer size limitation, i.e., its length is more than **mbs**-4, where **mbs** is the PHY buffer size specified as the second argument to **phys_uart** (see above), **syserror** (**EREQPAR**) will be triggered.

```
char *ab_out (word st, char *str);
```

This function writes to the interface the string specified as the second argument. Similar to **ab_outf**, the first argument specifies the retry state (the function may block). On success, the function returns the value of the second argument. The interpretation of **WNONE** as the state argument is the same as for **ab_outf** (except that this time the function cannot fail on memory allocation).

The string argument of **ab_out** must have been allocated previously by **umalloc**. The praxis should not touch that string after successfully passing it to **ab_out.** If the function has succeeded (i.e., returned not **NULL**), that string has been queued for transmission and will be later deallocated automatically. If the function fails (returns **NULL**), it leaves the input string intact.

If the string to be written over XRS is a constant, or it is stored as part of some larger structure (e.g., a VNETI packet) and shouldn't be deallocated automatically by XRS, **ab_outf** should be used instead of **ab_out**. The proper way to avoid an accidental interpretation of a spurious formatting sequence in an unknown string is to use this form of call:

```
ab_outf (ST_RETRY, "%s", str);
```

Of course, when the string is a known constant (and it does not include a formatting sequence), it can be used directly, e.g.,

```
ab_outf (ST_RETRY, "Enter next line:");
```

Finally, explicit formatting sequences can be escaped, like this:

```
ab_outf (ST_RETRY, "This line is fishy: \%d is escaped");
```

These two functions are used for string-oriented input:

```
int ab_inf (word st, const char *fm, ...);
char *ab_in (word st);
```

The first one blocks (retrying at the specified state) until an input line is available. Then it reads the input line and processes it according to the specified format, storing the decoded values in the remaining arguments (which must be pointers). The value returned by the function tells the number of decoded items. This is similar to **ser_inf**.

If the state argument of **ab_inf** is **WNONE**, and no input line is readily available, the function will return immediately with the value of zero.

The second function returns an unprocessed input line via a pointer. The string represented by this pointer should be deallocated by the praxis (via **ufree**) when done. The function blocks until a line is available, unless the state argument is **WNONE**, in which case it returns **NULL**.

Lines handled by the above functions need no CR/LF terminators: it is assumed that one standard (**NULL**-terminated) string represents one line. You have to remember that each line sent or received this way must fit into a packet, with the length limitation imposed by the maximum packet length declaration for the PHY. For example, in the initialization sequence above, the PHY packet length is 96 bytes; thus, the maximum length of a line that can be passed in the above setup is 92 characters (the zero sentinel must be accommodated in the packet).

9.1.2 Binary-oriented functions

The binary interface is provided by two functions (their headers are defined in **abb.h**):

```
byte *abb_outf (word st, word len);
```

This function allocates a buffer of size `len` bytes for an outgoing message and returns its pointer. If `st` is not `WNONE`, the function blocks retrying at the specified state, if either the buffer cannot be acquired (because of momentary memory contention), or the interface is not done with a previous message. When the function returns, the buffer has been allocated and queued for transmission. The invoking process should fill the buffer before it exits the current state.

If the state argument is `WNONE`, the function always returns immediately. If the interface is busy or memory cannot be acquired, the function will return `NULL`. Regardless of the value of `st`, a `syserror` will be triggered, if `len` exceeds the maximum packet capacity, i.e., `mbs`−4.

```
byte *abb_out (word st, byte *buf, word len);
```

This function is similar to `abb_outf`, except that is directly uses a buffer supplied by the praxis as the second argument. That buffer must have been allocated by `umalloc`. If the state argument is not `WNONE`, the function only returns if it succeeds, i.e., the buffer has been queued for transmission. The returned value is then equal to `buf`. Otherwise (`st` is `WNONE`), the function always returns immediately, with `buf`, if it succeeds, and `NULL` otherwise. If the buffer length (`len`) is larger than the packet capacity, the function triggers a `syserror`.

When the function succeeds, it will eventually deallocate the buffer by itself. The praxis should not touch that buffer after passing it to the function.

```
byte *abb_in (word st, word *len);
```

If the state argument is not `WNONE`, the function blocks (retrying at the specified state) until input data is available. Then it extracts the binary block from the received packet and returns its pointer as the function value. The second argument is used to pass to the praxis the length of the received data block in bytes.

If `st` is `WNONE`, the function always returns immediately. If no data is available for immediate reception, the returned value is `NULL`.

9.2 A bit about the internals

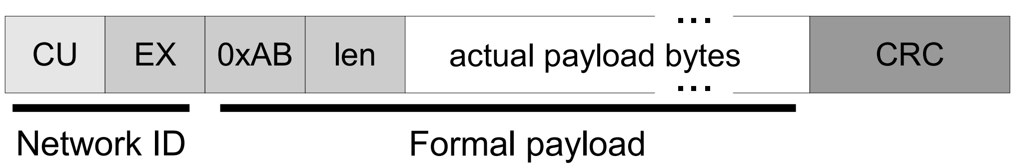

Figure 2: XRS packet header

Figure 2 shows the packet format as interpreted by XRS. The first byte of the packet contains the **CU**rrent packet number (as counted by the sender) modulo 256. Strictly speaking, this isn't an alternating bit protocol, as the packet number field consists of an entire byte (not just one bit). This value starts with zero and is incremented by one with every new packet prepared by the sender for transmission. The second byte indicates the **EX**pected number of a packet to arrive from the other party. The idea is exactly as described at **Persistent packet mode**, except that the packet numbers are (redundantly) represented by bytes rather than bits. The sender will keep re-sending the last packet until it receives a packet from the other party whose **EX** field is different from the **CU** field of the last sent packet. Then it will assume that the previous packet has been received, and set the **CU** field for the new packet to the value of **EX** received from the peer. The operation is symmetric in the two directions.

If upon receiving a valid new packet from the peer, the node has no ready data packet to send the other way, it will create a pure ACK packet, i.e., one containing empty data (len = 0). Such a packet, when received, is not interpreted as a data packet and its **CU** field is ignored. If the protocol operates in the active mode (see Section 9.1), pure ACK packets are sent periodically at 2 second intervals as a way to manifest the node's presence (heart beat), even if there is no actual traffic. This interval shortens upon receipt of an out of sequence (duplicate) packet, and returns to 2 seconds when things get back to normal. In the passive mode, the node will send one ACK packet upon receipt of a packet from the other party and stop there until a next packet arrives.

The above packet format was inspired by OEP, i.e., its intention was to provide for an easy coexistence of OEP with XRS. The location of the magic value (0xAB) in the packet header coincides with the location of the packet type code in OEP. This precludes accidental interpretation of an XRS packet as an OEP packet and vice versa (0xAB is not a valid OEP packet type). The positions of the requisite fields in the packet header (as well as the magic code) are described by symbolic constants in **`ab_params.h`** (they can be redefined easily, e.g., from **`options.sys`**).

10 The BOSS Plugin

Note that XRS uses the standard NULL plugin with the N mode UART PHY, placing its functionality on top of the VNETI API. BOSS, on the other hand, is entirely a *plugin*: it implements a very similar functionality underneath the VNETI API. Additionally, it offers an option of unacknowledged (out of band) transmission/reception. The plugin is configured the standard way, e.g.,

```
#include "sysio.h"
#include "plug_boss.h"
#include "phys_uart.h"
…
#define MAX_PPLEN    94
…
sint SFD;
…
      phys_uart (0, MAX_PPLEN + 2, 0);
      tcv_plug (0, &plug_boss);
      SFD = tcv_open (WNONE, 0, 0);
      if (SFD < 0)
            syserror (ENODEVICE, "uart");
…
```

The maximum packet length specified with **`phys_uart`** applies to the N mode packets. It covers the entire PHY payload, including the two Network ID bytes, but excluding the CRC code. As the plugin uses the Network ID bytes for its own header, the payload available to the praxis is two bytes less than indicated by the second argument of **`phys_uart`**. Note that the maximum value of that argument is 252 (translating into 250 bytes of BOSS payload).

The plugin provides two types of packets (both types sent and received over the same session descriptor) differentiated by the *urgent* attribute. Non-urgent (i.e., regular) packets are reliable; they are internally acknowledged, essentially in the same way as for XRS (or the P mode). At most one unacknowledged (non-urgent) packet can remain pending at any given time; thus, an attempt to send a new packet (**`tcv_wnps`**) while the previous one is waiting for an acknowledgment will block. Urgent packets, on the other hand, are queued immediately at the PHY (and sent as fast as possible), pretty much in the same way as with the pure N mode of the PHY coupled with the NULL plugin. Note that the integrity of those packets is still guarded with CRC by the PHY.

10.1 An illustration

Following the above initialization sequence, the session descriptor can be used with the standard API functions of VNETI. Although the plugin's name (BOSS) has been derived from Binary OSS, there is nothing inherently binary about its operation, except that the information is packetized and uninterpreted by the plugin as far as the content (payload) is concerned. For illustration, we can try to evolve the above initialization sequence into a simple data exchange setup involving ASCII strings. Here is an FSM that receives strings from the other party and echoes them back.[3]

```
fsm Echo (Boolean reliable) {

        address pka, pkb;
        sint len;

        state RDLOOP:
              pka = tcv_rnp (RDLOOP, SFD);
              len = tcv_left (pka) + 6;
              if (len > MAX_PPLEN)
                    len = MAX_PPLEN;

        state ECHOOUT:
              pkb = tcv_wnps (ECHOOUT, SFD, len, !reliable);
              tcv_write (pkb, "ECHO: ", 6);
              tcv_write (pkb, ((char*)pka)+tcv_offset (pka),
                  len - 6);
              tcv_endp (pka);
              tcv_endp (pkb)
              proceed RDLOOP;
}
```

The FSM is passed a **`Boolean`** flag indicating whether its communication should be reliable or not. The only place where the difference manifests itself is in the invocation of **`tcv_wnps`** whose last argument determines the urgent attribute of the outgoing packet.

Note that regardless of the value of **`reliable`**, the FSM will be receiving all packets (i.e., both packet types) in state **`RDLOOP`**. Non-reliable incoming packets are queued as urgent, so the FSM can check that attribute (with **`tcv_isurgent`**) if it wants to tell the difference.

10.2 A bit about the internals

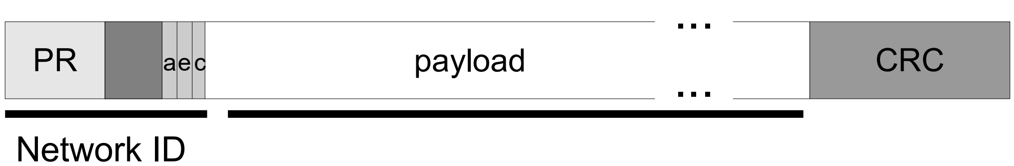

Figure 3: BOSS packet format

Figure 3 shows the packet format as seen by the BOSS plugin. The first byte (PR) tells apart the two packet types. Code 0xAD denotes a reliable (non-urgent) packet, and code 0xAC marks an unreliable (urgent) one. Any other value of the first byte causes the incoming packet to be ignored (passed). This way other plugins can be associated with the same PHY.

[3]Note that there is nothing particularly BOSS-y in this example: we only include it here to refresh your memory about some elements of VNETI's API.

If the first byte contains 0xAC (the packet is of the unreliable type), the second byte is then ignored (and can be used by the praxis as an "informal" extension of the payload). For a reliable packet, the three rightmost bits from the second header byte are used as:

c – the "current" alternating bit
e – the "expected" alternating bit
a – explicit ACK flag (set if the packet formally has no payload)

The meaning of bits *c* and *e* is essentially identical to bits CU and EX of the P mode. Bit *a* is used to mark the internal acknowledgment packets (devoid of a payload). It allows the plugin to tell the difference between an empty payload, which is legal, and no payload at all). The *c* bit of an acknowledgment-only packet is ignored.

The reliable session implemented by the plugin always operates in a *passive* mode, i.e., it will not retransmit the last unacknowledged packet (or an ACK) until it receives something from the other party (the OSS). Consequently, two nodes cannot communicate reliably with BOSS, unless the praxis (at least on one end) makes sure to send some "reliable" traffic periodically in a manner amounting to polling. In asymmetric communication, like OSS ↔ node, the OSS end should execute an active version of the scheme sending ACK packets at some intervals.

11 Piter: the PicOS Terminal Emulator

Directory **`Scripts`** includes a script named **`piter`** which can be used for direct communication with a praxis using any of the three modes of packet UART interface, also optionally including XRS or BOSS on top of the N mode. When the system is installed (**`deploy`** is executed in the **`PICOS`** directory), **`piter`** (with some adjustments in the header required to make it executable with Tcl 8.5) is copied to the user's **`bin`** directory.

11.1 Call arguments

Here is the list of parameters accepted by the script:

`-p` ***`device`***

This parameter is mandatory and identifies the UART device. Since version 1.7.13 of Cygwin, the script runs under the native version of Tcl/Tk, so it uses Unix-like names for the serial devices (**`COM1`**, **`COM2`**, and so on). **`COM1`** is translated to **`/dev/ttyS0`**, **`COM2`** to **`/dev/ttyS1`**, etc., with a shift by 1. You can specify the device directly, e.g., **`/dev/ttyS1`**, or by number, e.g., 1. In the second case, the program will try to open these devices (in order): **`/dev/ttyS1`**, **`/dev/ttyUSB1`**, until it succeeds. This applies to both Cygwin and Linux.[4] The virtual UARTs of udeamon (see the VUE[2] doc) are named **`/dev/pty0`**, **`/dev/pty1`**, and so on (on Cygwin), or **`/dev/pts/0`**, **`/dev/pts/1`**, … (under Linux). Such a device should be specified exactly as named e.g.,

```
piter -p /dev/pty6 -s 115200
```

The program can also connect directly to a VUE[2] model of UART or to a TCP port representing the UART (e.g., as RS232 over TCP). In the first case, the device should then be specified as:

`node`**`:`*****`host`*****`:`*****`port`***

where ***`node`*** is the node number whose UART you want to connect to, ***`host`*** is the Internet name (or the numerical IP address) of the host running the model, and ***`port`*** is the Internet port of the model on the host. Shortcuts to defaults are possible, e.g.,

[4] With the non-native version of Tcl (for Cygwin < 1.7.13), the devices were simply named COM1, COM2, and so on. The number would then directly translate into the COM port.

2::

means node 2 on ***localhost*** using the default port 4443. That is equivalent to:

2:

The ultimate shortcut:

:

means node 0, ***localhost***, port 4443.

For a TCP connection, replace node with tcp, e.g.,

```
tcp:localhost:9022
tcp::9022
```

-s ***speed***

This parameter specifies the UART bit rate in bps. If absent, the default rate of 9600 is assumed.

-m ***mode***

mode must be a single letter **d**, **p**, **n**, **s**, or **x**. It selects the communication mode in the following way:

- **d** – direct mode. This mode is intended for straightforward (unstructured) communication, with **piter** acting as a simple terminal emulator. Note that the L-packet mode on the node's side is also compatible with the direct mode of piter.
- **p** – matching the P mode of VNETI interface on the node's side. Information sent by piter to (and expected to arrive from) the UART is organized into packets conforming to the P mode as described in Section 4.
- **n** – matching the N mode of VNETI interface without the add-ons (XRS or BOSS). Information sent by piter to (and expected to arrive from) the UART is organized into packets conforming to the N mode as described in Section 3.
- **e** – matching the E mode of VNETI as described in Section 6. This mode should be used in connection with **-b**, unless the packet payloads are printable strings.
- **f** – matching the F mode of VNETI as described in Section 6. This mode should be used in connection with **-b**, unless the packet payloads are printable strings.
- **x** – conforming to XRS on top of the N mode, as described in Section 9.
- **s** – conforming to the BOSS plugin (on top of the N mode), as described in Section 10.

The default ***mode*** is **d**.

-l ***pktlen***

The maximum packet length. For any ***mode*** other than d, this should match the corresponding parameter of **phys_uart** as specified by the praxis. The default value is 82.

`-b` ***`macro_file_name`***

If present, this parameter indicates binary interface, i.e., instead of ASCII characters, the user will input sequences of numerical bytes and view the received information in hexadecimal, as sequences of bytes (unless a conversion plugin is provided – see below). If ***`macro_file_name`*** is present, it points to a file containing macro definitions intended to simplify entering some typical or popular sequences of bytes. If no file name is provided (but **`-b`** is present), the script will try to open the standard file named **`piter_mac.txt`** (in the current directory). If that fails, the script will be using no macros.[5]

`-P` ***`plugin_file_name`***

Note that this is capital **`P`**. This parameter plugs into the script a set of functions provided by the user for preprocessing (converting) the UART input and output, as explained below. If the file name argument is absent (but the option is present), the default file name **`piter_plug.tcl`** (looked up in the current directory) is assumed. If that file is absent, but a file named **`shared_plug.tcl`** exists, that file is used instead. This facilitates the option of sharing plugins between piter and udaemon.

`-f` ***`log_file_name`***

The program will preserve all output received from the device in the specified file. If a plugin preprocessor for output is provided, the *processed* version of output will be logged.

`-F` ***`log_file_name`***

The program will preserve all output received from the device, as well as all input from the user, in the specified file. The input is always stored exactly as entered by the user, i.e., a plugin preprocessor doesn't impact the shape of the stored input.

`-S`

Note that this is capital **`S`**. If present, this must be the only parameter. It tells the program to scan for available (openable) devices, equivalent to trying the **`-p`** option (see above) with all numbers from 0 to 255. The program will just report the openable devices and exit.

`--`

This argument ends the sequence of arguments intended for piter. Any arguments that follow will be passed to the plugin (see Section 11.7.1).

11.2 GUI mode

When called without arguments, the script runs in a GUI mode whereby the terminal is presented in a window (similar to udaemon's UART window). Initially, the script is not connected (this is indicated by the status line displayed in the window's title). By pressing the "Connect" button located in the right bottom corned of the window, you will open a dialog presenting a self-explanatory collection of connection parameters, basically equivalent to the command line arguments for a non-GUI invocation. The device (UART) can be selected from a menu (the list of available serial devices is determined by the scrip via a scan (equivalent to **`-S`** – see above). A device name can also be entered manually into the *Other* field (which takes precedence over the menu). While connected, you can hit the "Connect" button at any time to drop the current session and connect to a new device (possibly with different parameters).

[5] With the addition of plugins, the role of macros is questionable, and I am inclined to remove them. I am hesitating, because they do provide a kind of programming-free way to enhance the capabilities of **`piter`**, which may have its advantages to some (future) users.

Note that in the GUI mode the log file (equivalent to **-f** or **-F**) is controlled from the terminal window (in a manner similar to udaemon's UART window). Logging can be started and stopped on the fly.

There is an option to save the settings in an "rc" file in the home/user directory. If a saved setting is available, it will be assumed as default by the next open dialog.

11.3 Examples

Consider the following sample invocations of the program:

```
piter -p 1 -m d -s 19200
piter -p 3 -m x -s 115200 -l 96 -b macfile.txt
piter -m x -p /dev/ttyS6 -P
```

The second case looks like binary XRS, i.e., the praxis is probably using the functions **abb_out** and **abb_in** for talking to the the UART.

When called without **-b**, the script will establish an ASCII-type command line interface to the praxis, basically acting as a terminal emulator. This, of course, assumes that the praxis also talks ASCII and that the parameters of the script match the settings used by the praxis. For example, if the praxis uses P-mode packets filling them with ASCII strings (and expecting ASCII strings in response), it may make sense to invoke the script as:

```
piter -p 7 -m p -s 19200 -l 84
```

Note that if the maximum packet length setting of the script does not match that used by the praxis, then some (long) messages bona fide sent by the script may not be received by the praxis or vice versa.

11.4 Internal commands

The script defines internal commands (processed by the script instead of being sent as input to the praxis), which are indicated by **!** as the first non-blank character in a line. Here is the list of those commands:

!e [on|off|1|0] also **!echo [on|off|1|0]**

This command switches on or off the echo flag. Arguments **on** and **off** are equivalent to **1** and **0**, respectively. Without an argument, the command toggles the flag. With the echo flag on, the script will echo all input before sending it to the praxis. This is more useful in the binary mode (described below) where the input line may be obfuscated by macros. The echo flag is off by default.

!t *level* [*filename*] also **!trace *level* [*filename*]**

This command switches the trace on (when ***level*** is > 0) or off (when ***level*** is 0). If ***filename*** is specified (when ***level*** is > 0), the trace will be directed to the indicated file; otherwise, it will show up on the standard output. The largest useful trace level is 3.

!r also **!reset**

The command *resets* the protocol. It mostly makes sense for non-trivial modes (like **p**, **n**, **s**, and **x**), where it empties the message queue and resets the expected/current counters/bits to zero. There is no built-in action for mode **d**; however, the plugin reset function (if provided) is called in all cases, including the **d** mode (see below).

!o also **!oqueue**

The command reports the current size of the queue of outgoing messages (it may be useful when the protocol appears to be stuck). The reset operation (**!r**) erases the queue and brings the protocol back to the initial state. For the **d** mode, there is no reason why outgoing messages should pile up, so the output queue should be always empty in that mode.

!b [on|off|1|0] also **!bypass [on|off|1|0]**

The command switches on and off the so-called *bypass flag*, which at present only applies to the **s** (BOSS) mode. Arguments **on** and **off** are equivalent to **1** and **0**, respectively. Without an argument the command reports the current setting of the flag. The default setting of the flag is **off**.

With mode **s**, when the bypass flag is on, the lines (packets) addressed to the node are sent over the unreliable session. When the flag is off, which is the default, the packets are sent using the reliable part of the protocol.

The formal meaning of the bypass flag also applies to other schemes, although for them the flag cannot be changed. Thus, for example, for modes **d** and **n**, the bypass flag is permanently on, because any message addressed to the node is always sent immediately (it never waits for any handshake, acknowledgment of the previous message, and so on). Similarly, for mode **p**, the flag is permanently off, because all outgoing messages are queued and remain queued until acknowledged.

!q also **!x**

This command terminates the script.

Internal commands are never subjected to macro/plugin preprocessing (they are never presented to the plugin).

11.5 Binary input/output

By including **-b** in the list of arguments of the script, possibly followed by a file name, you invoke it in the binary mode which will allow you to send/receive binary data. In such a case, an input data line consists of a sequence of bytes, e.g., specified as a straightforward list of numbers like this:

```
41 0x86 0x55 0x72 0 13 79 0x1 0xf
```

The legitimate syntax for a number includes everything that can be evaluated by Tcl as an expression, e.g.,

```
41 1+2+3 (7+8)/2
```

and so on. Note that blanks separate entries; in particular:

```
8+3 +2+1 +5
```

is three numbers, not one. The result of an expression is truncated to 8 bits and *always* interpreted as the value of a single byte.

In addition to single-byte values, you can directly specify strings, e.g.,

```
45 0x7f "this is a string" 0 2+4+5
```

The above sequence stands for 20 bytes: each of the 16 characters of the string encodes one byte (stands for its ASCII code). Note that there is no implied sentinel at the end of a string; if needed, it must be provided explicitly, either as a separate byte value (like in the above list) or within the string, e.g.,

```
"Ten bytes\0"
```

which is exactly equivalent to:

```
"Ten bytes" 0
```

A block of data arriving from the praxis is displayed, in the binary mode, as a sequence of hexadecimal byte values, two digits per byte, separated by blanks, e.g.,

```
Received: 13 < 6c 68 6f 73 74 20 31 20 2d 3e 20 31 00 >
```

The number in front of the opening **<** is the number of bytes in the received block. Note that there's no **0x** in front of a value, which is in a sense inconsistent with input, because, e.g., you cannot input **6f** as a byte value without preceding it with **0x**.

11.6 Macros

Dealing with binary data is tedious, unpleasant, and error-prone. One way to make it a little bit easier is to take advantage of the simple macro mechanism of **piter**, which at least can take care of the input end. A more powerful and flexible solution, which can also transform the output into a more presentable form, involves programming a plugin and will be discussed later.

A macro file (see the file **sample_macros_ab.mac** in directory **Scripts**) consists of lines, with each line defining one macro. Empty lines, as well as those starting with **#** (possibly preceded by blanks) are ignored. A simple macro definition may look like this:

name = body

where name must start with a letter or _ and contain letters, digits, and _. Any blanks preceding or following ***body*** are ignored. If you want to preserve them, you can encapsulate ***body*** in quotes, e.g.,

```
_a_   = 0x61
t4    = " 0x61 0x62 0x63 0x64"
```

Macros are applied in a purely textual manner according to a very straightforward algorithm. After you have typed in a line, the script will scan the list of defined macros in the order of their appearance in the file, and, for each macro in turn, replace all the textual occurrences of the macro name with the macro body (disregarding any delimiters, boundaries, and so on). Note that subsequent macros will be applied to the already modified line. For example, consider this sequence of macros:

```
NL = LFCR
CR = "0x0D "
LF = "0x0A "
```

and suppose that the input line looks like this:

```
1 3 0x12 NL 0x55 CR LF NL
```

After the expansion, the line will become:

```
1 3 0x12 0x0D 0x0A  0x55 0x0D  0x0A  0x0D 0x0A
```

This is because when the macros **CR** and **LF** are expanded, the line already includes an expansion of **NL**. Note that if you move the definition of **NL** in the macro file after **CR** and/or **LF**, then the respective secondary substitution(s) will not take place.

Sometimes such an aggressive expansion is not exactly what you have in mind. If you need a simple command-like macro (as is often the case), and want to prevent inadvertent expansions, you can precede the macro name with ^ to say that it should only be expanded if the keyword appears at the beginning of a line, e.g.,

```
^reset = 0x03 0x00
```

The keyword **reset** will only be replaced in the typed text if it begins a line. Initial white space characters are admissible.

Macros can have parameters, e.g., with this macro:

```
spin(a) = “0x01 a/256 a%256”
```

a line looking like this:

```
13 spin(0x1341) 15
```

will become:

```
13 0x01 0x1341/256 0x1341%256
```

which is equivalent to:

```
13 0x01 0x13 0x41
```

Here is another example:

```
^diag(s) = 0x04 0x00 "s" 0x00
```

illustrating how strings can be used as macro parameters. A line like this:

```
diag (help me please)
```

will expand into:

```
0x04 0x00 "help me please" 0x00
```

Note that the string is terminated by an explicit **NULL** byte (the string itself doesn't include such a byte).

Macro parameters are unstructured (they are only separated by commas); thus, for example, this macro:

```
^echo(u,v) = 0x03 v u
```

invoked as:

```
echo (1 2 3 4 5 6 7 8, 99+1)
```

will expand into:

```
0x03 99+1 1 2 3 4 5 6 7 8
```

11.7 Plugins

With the **-P** argument (see Section 11.1), or from the respective widget in the GUI mode, you can augment the program with a plugin, which is a set of functions for preprocessing user input and/or the output arriving from the node. Note that, unlike macros, this kind of conversion is not restricted to the binary mode. Basically, you can put into those functions any Tcl code, so the preprocessing can be arbitrarily elaborate. For binary input, this option offers much more flexibility than macros; however, it does require some programming. Macros and plugin conversion can coexist (remember that macros only apply to the input end and to the binary mode of communication), although using both those features together probably makes little sense (and may lead to some confusion).

11.7.1 Interface functions

The complete interface of a plugin is described by seven functions taking care of these ends: initialization, reset, termination, textual input, binary input, textual output, binary output. By "textual input" I mean a line entered by the user in a textual (non-binary) mode (any run of the script without **-b**). The same way, textual output is any line arriving from the other party while the script is operating in a non-binary mode. The two "binary" variants of the functions are referenced when **piter** runs with **-b**.

There exist default (trivial) definitions of all four functions, so **piter** can operate without a plugin. The user plugin need not provide (replace) all functions, but whichever functions it does provide, they override the defaults. Here are the function names:

plug_init	initialization
plug_reset	protocol reset (the **!r** command – Section 11.4)
plug_close	UART close (GUI version only)
plug_inppp_t	textual input

plug_inppp_b	binary input
plug_outpp_t	textual output
plug_outpp_b	binary output

The first function accepts a single argument which is the the list of plugin-specific program call arguments of **piter** (see Section 11.1). These are the arguments that follow `--` on the program call line. The second function takes no arguments. The first two functions accept no arguments and are not expected to return anything. Each of the remaining four function takes a single argument and is expected to return an integer value. The argument is a by-name reference to the current "line" (a string) to be preprocessed. When invoked, the respective function is supposed to replace the original string with its converted version and return a value interpreted as a status of the operation. Here are the full default implementations of the plugin functions:

```
proc plug_init  { argmnts } { }
proc plug_reset { } { }
proc plug_close { } { }
proc plug_inppp_t { inp } { return 2 }
proc plug_inppp_b { inp } { return 2 }
proc plug_outpp_t { inp } { return 2 }
proc plug_outpp_t { inp } { return 2 }
```

As you can easily see, the default conversion functions don't touch the string and their converting actions are thus void. The value returned by a default converting function says exactly that (the string hasn't been changed). Two other sensible return values are:

0 - the line should be ignored
1 - the line has been changed

Both nonzero values mean that the line should be sent for further processing. In the case of **plug_inppp_t**/**plug_inppp_b** it means that the line should be written out to the node; for **plug_outpp_t**/**plug_outpp_b**, it means that the line should be presented to the user.

11.7.2 Examples

For illustration, suppose you want to convert all lines arriving from the device in a textual mode by transforming all letters to the upper case and compressing all white spaces to a single space, also removing all leading and trailing spaces. Lines that end up empty after the processing, should be ignored. Here is one function that will do the job:

```
proc plug_outpp_t { ln } {
	upvar $ln line
	regsub -all "\[ \t\]+" $line " " line
	set line [string toupper [string trim $line]]
	if { $line == "" } {
		return 0
	}
	return 1
}
```

Strictly speaking, when we are returning 1 at the bottom of the function, we are not absolutely sure that the original line has been modified, so perhaps we should double check and return 2 if it hasn't?. It doesn't really make a lot of difference, especially in a textual mode. While you should never return 2 for a line that *has been* modified, returning 1 when it hasn't is perfectly OK, especially if its lack of change can be viewed as accidental. Look at it this way: if you can easily tell whether the line has been modified, then you can afford to tell that to the program (it may then save it some unnecessary processing); otherwise, don't bother.

If the above function is all you need, you can put it into a file and make that file available to **piter** with **-P**. This is all you need to create a preprocessor plugin. Note that your plugin functions may want to reference other functions, which you are free to program and include along with the plugin functions. To avoid an accidental conflict with a function defined by **piter**, you can use some distinguishing prefix for all local function names or put the entire plugin into its own namespace. The plugin file is evaluated at level 0 of **piter**, so its functions see all functions and global variables of the script and can use them creatively, if they know what they are doing.

Exactly the same idea applies to preprocessing textual input; the only thing that is different is the function name. The binary mode, however, does require some additional explanation.

Let us start from the input end, which is trickier. This is the part taken care of by **plug_inppp_b**. Without the plugin, an input line entered by the user (which is always textual to begin with) is first optionally subjected to macro expansion (as explained earlier) and then parsed for bytes and strings, which are ultimately transformed into binary bytes. The plugin function kicks in after the macro expansion and before that final parsing, which means that it receives the original line entered by the user, possibly passed through the macro expansion mechanism, and its objective is to generate the string of values (something that could have been directly entered by the user), which will then be converted by **piter** into true binary bytes. The important point is that the function doesn't have to deal with binary data at all.

For an easy illustration, consider this macro:

```
^diag(s) = 0x04 0x00 "s" 0x00
```

that was discussed a while ago (in Section 11.6). Suppose that you want to program a plugin function that will accomplish basically the same feat as the above macro, i.e., it will convert any line looking like an invocation of the macro into its pertinent expansion (leaving all other lines intact). Here is how such a function may look:

```
proc plug_inppp_b { ln } {
      upvar $ln line
      if ![regexp "^diag\[ \t\]*\\((.*)\\)" $line j s] {
            # no match, do not touch
            return 2
      }
      set line "0x04 0x00 \"$s\" 0x00"
      return 1
}
```

For binary output, the function, **plug_outpp_b**, is invoked with a list of hexadecimal values (that otherwise would be presented to the user directly) as its argument. For example, here is a sample string output in binary mode with the default plugin:

```
Received: 13 < 6c 68 6f 73 74 20 31 20 2d 3e 20 31 00 >
```

The sequence of values between **<** and **>** contains all the bytes received from the device in the current packet (line). When handling this input, your plugin function would receive as its argument this Tcl string:

```
6c 68 6f 73 74 20 31 20 2d 3e 20 31 00
```

i.e., the list of hexadecimal values without any decorations. For illustration, this is a sample converter that spots all printable strings (cases when all bytes in the packet are printable) and transforms them into ASCII lines:

```
proc plug_outpp_b { ln } {
      upvar $ln line
      set out ""
```

```
        foreach b $line {
              set b [expr 0x$b]
              if { $b < 32 || $b > 126 } {
                    # found a non-printable byte
                    return 2
              }
              append out [format %c $b]
        }
        # all bytes printable
        set line $out
        return 1
  }
```

In this case, returning 2 for a line that has not been affected (instead of 1) will have the effect of adding the standard decorations to the output string:

Received ***nn*** **< ... >**

which will not be done for a string marked as "processed" (which the plugin function is assumed to have transformed into its proper final shape). If you are annoyed by the encapsulation (of unprocessed chunks) and would rather see the bytes alone, use this simple plugin function:

```
proc plug_outpp_b { ln } { return 1 }
```

By the same token, if you'd rather see the encapsulation on all lines, including the processed ones … I'm sure you know what to do.

11.7.3 Some useful functions

If you understand what a function or a variable defined within **piter** does or means, you can reference it in your plugin functions. Here are a few potentially useful operations:

proc pt_outln { line } { …

The function sends the string represented by the argument to the node. You can use it to inject lines/packets into the output. For example, with this function:

```
proc plug_inppp_t { ln } {
      upvar $ln line
      pt_outln $line
      return 1
}
```

you will be sending each input line twice to the node. Note that if you change the returned value to `0`, the function will become equivalent to the void default.

proc pt_tout { line } { …

The function provides the recommended way of writing a textual line to the terminal (so the user can see it). If the output is being logged (saved to a file), a line written this way will be logged as well.

proc pt_touf { line } { …

This function is similar to **pt_tout**, except that it is intended for special lines (like diagnoses of problems) that are normally not written to the log file. A line presented this way will only be written to the log file if the **-F** option is active, i.e., user input is being logged as well.

proc pt_bypass { on } { …

This function can be used by the plugin to change the bypass flag (see Section 11.4). At present, this can only make sense (and be useful) for a plugin intended to operate with

BOSS (the **s** mode of **piter**) where the plugin may want to switch between the reliable and unreliable sessions. The argument can be 0 or 1 (off and on, respectively). The value returned by the function tells the new value of the flag. If you invoke the function in a mode in which the flag cannot be changed (i.e., any mode other than **s**), it will return the previous (permanent) value of the flag regardless of the argument.

11.7.4 Initialization and reset

If your plugin requires specific non-trivial initialization, put it into **plug_init**. The function gets called after the UART has been opened and all the protocol hooks have been set up, so it can safely write something to the node. For illustration, consider this situation when the "normal" exchange must be preceded by some introductory handshake:

```
set P_callback ""
set P_block 1

proc plug_init { argmnts } {
      global P_callback
      pt_outln  "0x00 0x01 0x02 0x03"
      set P_callback [after 5000 plug_init]
}

proc plug_outpp_b { ln } {
      global P_block
      upvar $ln line
      if  $P_block {
            # waiting for handshake completion
            if { [llength $line] == 2 &&
              "0x[lindex $line 0]" == 0x22 &&
              "0x[lindex $line 1]" == 0x33 } {
                  # handshake OK, stop the callback
                  global P_callback
                  catch { after cancel $P_callback }
                  # normal operation
                  set P_block 0
            }
            # ignore otherwise
            return 0
      }
      ... normal processing ...
      ...
      return 1
}
```

Upon startup, **plug_init** will be periodically (at 5 seconds intervals) sending some message until a specific response arrives from the node. No other messages will be sent or received until then. We can expect that **plug_inppp_b** (not listed here) ignores (possibly with some warnings) its input for as long as **P_block** remains nonzero.

The only way for the second special function, **plug_reset**, to be called is when the user executes **!r** (see above). The function is invoked after all the relevant elements of the underlying protocol have been re-initialized.