UART Communication via VNETI (TCV)

September 2011

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, we 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 three options for UART-over-VNETI selectable at compile time (via settable constants). 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.

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 three possible flavors for this communication is selected by setting **`UART_TCV_MODE`** to:

`UART_TCV_MODE_N`	(0)	- simple packet mode
`UART_TCV_MODE_P`	(1)	- persistent (acknowledged) packet mode
`UART_TCV_MODE_L`	(2)	- line mode

The first mode is the default (it is selected automatically if you do not set **`UART_TCV_MODE`** explicitly). It essentially treats the UART as a 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 is the simplest one and can be used to send/receive line-oriented data. It is intended as a simple replacement for the traditional (ASCII) UART interface.

There is also an option 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 a checksum. This is described further in this document as the **External Reliable Scheme** (XRS); it belongs to this document, as it is primarily intended for the UART. Its role is to supplement mode 0 (**`UART_TCV_MODE_N`**) by making sure that all packets are actually received in a good shape. Although mode 1 (P) was originally intended for that, XRS may be more convenient in some cases, as it gives (on top of the reliable delivery) the same flavor as the traditional character oriented (formatted) “direct” operations on the UART. On the other hand, the P mode may be better suited for binary communication, even though a binary variant of XRS exists as well (as described further in this document).

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 by VNETI and blocked at the PHY level (awaiting acknowledgments). With XRS, there can be at most one outstanding packet (handled by VNETI), which puts the least possible strain on VNETI's buffer space.

I am contemplating adding a feature to the P mode whereby urgent packets (as understood by VNETI) will be sent “out of band” and without acknowledgments. This way it will be possible to have two coexisting types of traffic: reliable and traditional (e.g., some notifications that are not essential for a consistent operation of OSS). Arguably, the praxis will then have no need to send more than one “reliable” packet at a time before hearing back from the other party. While at present a flavor of “unreliable notifications” is offered by **`diag`**, diags are disruptive to the normal stream of data and should thus be reserved for true emergencies.

Anyway, we shall see if the P mode finds its way into praxes. It was implemented a few years ago and has not been touched since then, until my recent cleanups (and its incorporation into VUE[2]). My intuitions would suggest that even when talking to a sophisticated OSS program, it is usually better to implement reliability/persistence in the praxis. This is especially true when the praxis switches the channel among multiple modes of communication (e.g., invoking OEP). If the reliability is implemented in the PHY (as with **`UART_TCV_MODE_P`**), such switching becomes impossible. Then again, not all praxes have to do this kind of switching, so perhaps either solution has its niche.

It is recommended to use mode P or N (the latter combined with XRS) for communicating with those OSS programs whose interface is up to us to shape. This will provide for a two-way reliable handshake with the OSS program, whereby everything can be accounted for and practically all communication errors can be diagnosed and resolved. Mode 2 is still needed for those cases where we have no control over the OSS interface, e.g., talking to a second party satellite station.

Another difference between XRS and the P mode is that while the latter (being implemented at the PHY level) is restricted to the UART, XRS can be added to any VNETI session (it accepts a VNETI session ID as a parameter). Consequently, it can be viewed as a standard way of implementing in the praxis symmetric and reliable exchange of data between a pair of peers using an RF-type channel. Note that it is different from OEP (the Object Exchange Protocol) in that the latter is intended for asymmetric (and episodic) exchange of large objects, while XRS works well for sustained, command-response-type sessions.

Initialization

The function to initialize the UART PHY looks the same for all three basic 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). If 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.

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.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. For UART communication, Network ID is not needed (the PHY is instructed to ignore it), so it can be considered part or the payload. XRS uses the first four bytes from the payload, including the two bytes of Network ID, which leaves mbs-4 bytes for the user.

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).

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.

In contrast to typical RF PHYs, which start in the OFF state, all three 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.

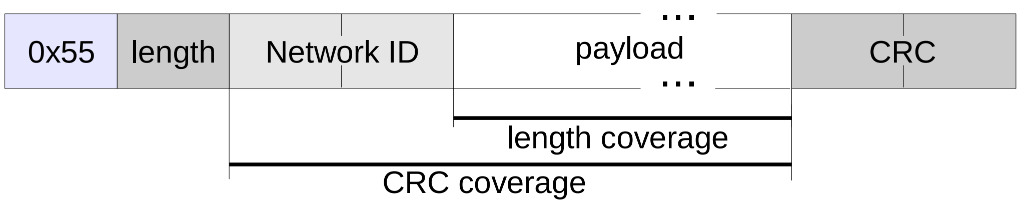
Now for some operational details.

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. In particular, the concept of Network ID is retained, which, theoretically, enables multiple nodes to communicate over a shared serial link emulating a broadcast channel.

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 below. 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].

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

[1]This timeout is probably 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.

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 not 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 basically treated as part of the payload.

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 from the previous mode, 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 peer. 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.

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, which suffers 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.

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.

Options

The list of options settable and testable with **`tcv_control`** includes switching on and off the transmitter/receiver end of the connection. In contrast to a typical RF module, both the transmitter and the receiver are initially on (i.e., the module is fully operational). For mode P, you cannot switch the two components individually, as it makes no sense: you cannot have reception without being able to transmit acknowledgments. Consequently, by switching off the receiver, i.e.,

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

you switch off the entire PHY. This is equivalent to **`RXOFF`** + **`TXHOLD`** for a typical PHY module (and for the remaining two UART modes). In fact, **`PHYSOPT_TXHOLD`** has exactly the same effect in this mode as **`PHYSOPT_RXOFF`**. There is a special symbolic constant, **`PHYSOPT_OFF`** = **`PHYSOPT_RXOFF`**, to be used for modules that can only be switched off globally. Similarly, **`PHYSOPT_ON`** = **`PHYSOPT_RXON`** is intended for switching them on.

For modes N and L, instead of switching the transmitter off, you can put it in the hold state (**`PHYSOPT_TXHOLD`**) where it will queue the packets arriving from the praxis, instead of discarding them (as in the normal OFF state). Note that the global OFF state for the module operating in P mode *holds* the transmitter. In the spirit of the loss-less communication, outgoing messages are never discarded in this mode.

Actions triggered by **`PHYSOPT_TXON`**/**`PHYSOPT_TXOFF`** in P mode are special: they toggle the so-called *active discipline*, which is switched off by default. When "active" (after **`PHYSOPT_TXON`**), the PHY will be issuing periodic ACKs (every second), even if it has nothing else to say. Note that at least one side of the connection should be doing that for reliability, but it is formally sufficient if only one side does the periodic polling. 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.

`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 above) when the module was initialized.

Diag messages

All three modes provide a crude, but generally quite effective, way of accommodating diag messages that must be expedited to the UART. 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 or P, 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 its transmission immediately).

2. A sequence of NULL characters (code **`0x00`**) 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.

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 **`0x00`** 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. A program that isn't, will ignore them, interpreting the apparent disturbance as a random error (from which it should be prepared to recover).

In mode L, the action is a bit simpler, as a diag message never interrupts any “packet” but, in the worst case, it may interfere with a message being written out by the PHY.

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/`**.

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 = 0xffff;
tcv_control (SFD, PHYSOPT_SETSID, &w);
tcv_control (SFD, PHYSOPT_TXON, NULL);
tcv_control (SFD, PHYSOPT_RXON, NULL);
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.

The specific necessary initialization steps consist in 1) setting the network ID of the PHY to **`0xFFFF`** 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 in:

```
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 TCV'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.

Here are the string-oriented functions (requiring **`ab.h`**):

```
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).

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.

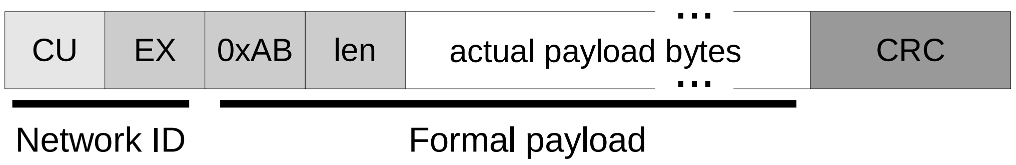
```
byte *abb_in (word st, word *len);
```

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.

A bit about the internals

This is the packet format used by XRS (starting from the Network ID field):



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 above), 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).

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 including XRS 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.

Here is the list of parameters accepted by the script:

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

This parameter is mandatory and identifies the UART device. If the device is a number, then its interpretation depends on the system. On Windows, the program will try to open the COM port with this number. On Linux, it will try these devices (in the listed order): **`/dev/ttyUSB`*`n`***, **`/dev/tty`*`n`***, **`/dev/pts/`*`n`***, where ***`n`*** is the specified number, stopping at the first one that opens. Note that the last case corresponds to a pseudo-device which, e.g., may map to a VUE[2] virtual terminal. Otherwise, i.e., if the argument of **`-p`** is not a number, the script will treat it literally as the (system-dependent) device name to open.

Note: serial devices on Windows can be specified in several ways, some of which I am probably not even aware of. Instead of just the number, you can specify the full name of the COM device, e.g., **`COM7:`** (the colon is important). This may not work for COM numbers 10 and higher, but something like **`\\\\COM14`** (no colon this time) may do the trick. The virtual UARTs od udeamon (see the VUE[2] doc) are named **`CNCA0`**, **`CNCA1`**, etc. Such a device should be specified as **`\\\\CNCAx`**, e.g.,

```
piter -p \\\\CNCA0 -s 115200
```

`-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`**, or **`x`**. It describes the communication mode (corresponding to the packet UART modes) in the following way: **`d`** – direct mode, which can be the traditional (packet-less) ASCII interface or L packet mode; **`p`** – the P packet mode; **`x`** – XRS (with N mode). The default ***`mode`*** is **`d`**.

`-l` ***`pktlen`***

The maximum packet length. For ***`mode`*** equal **`p`** or **`x`**, 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.[2]

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

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.

`-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 stored in the file.

`-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.

Note: on Windows, **`-S`** does not scan for the virtual UARTs created by udaemon.

When called without parameters, the script runs in a simple GUI mode whereby the terminal is presented in a window (similar to udaemon's UART window). The script's parameters (corresponding to the above list of command-line arguments) are entered in a special dialog, which should be self-explanatory. The device (UART) can be selected from a menu (the list of available UARTs is determined by the scrip via a scan (equivalent to **`-S`** – see above). A device that does not appear on the menu can be entered manually into the *Other* field (which takes precedence over the menu).

Note that 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 upon the next call.

Examples:

[2] With the recent 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.

```
piter -p COM1: -m d -s 19200
piter -p 3 -m x -s 115200 -l 96 -b macfile.txt
piter -m x -p \\\\CNCA3 -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.

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. Two such commands are implemented at present:

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

This command switches on or off the echo flag (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 and **x**), where it empties the message queue and resets the expected/current counters. 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.

!q

This command terminates the script.

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

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 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`**.

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.

Macros:

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 lines 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. 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
```

Plugins:

With **`-P`** (see above) you can declare 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 function 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.

The complete interface to the plugin is described by six functions taking care of these ends: initialization, reset, 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. The user plugin need not provide 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 – see above)
`plug_inppp_t`	textual input

plug_inppp_b	binary input
plug_outpp_t	textual output
plug_outpp_b	binary output

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 default implementations of the plugin functions:

```
proc plug_init  { } { }
proc plug_reset { } { }
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 the 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.

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.

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. Suppose 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., a list of hexadecimal values. For illustration, his 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 encapsulating the input string into the brackets:

Received *nn* < ... >

which will not be done for a string marked as “processed” (which you have probably 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.

Some useful functions defined by `piter`:

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 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 NULL 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 saved to a log file, a line written this way will be saved 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.

[To be continued]

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 { } {
      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 &&
                 [lindex $line 0] == 0x22 &&
                 [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.