UART Communication via VNETI (TCV)

May 2009

Note: modifications with respect to the October 2008 version are in blue

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 **PicOS.pdf**). 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.

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 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.

I have recently added 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, it does not appear very useful (I implemented it four years ago, and it has never been used except in tests). My experience suggests that even when talking to a sophisticated OSS program, it is usually better to implement reliability 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.

[So perhaps modes 1 and 2 are not needed at all. The reason for mode 2 was to make it possible to use a simple terminal-like interface when (and if) the old UART driver is eliminated.]

In essence, the role of XRS is to provide 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.

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 configuration constants for UART apply to the UART PHY. For the first two modes, **UART_BITS** must be set to 8, as otherwise the UART will not be able to send/receive full bytes, which is required for the correct operation of the interface. For the third mode, everything is going to work fine as long as the UART is able to send and receive ASCII characters.

UART_RATE_SETTABLE can also 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 different modes. For mode 0 (N), the 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.

For mode 1 (P), **mbs** must be between 1 and 250 (zero stands for 82, as before). Note that it can be odd. With modes 1 and 2, it is legitimate to send and receive odd-length packets. For mode 1, the specified length covers solely the payload. There is no Network ID field, and the CRC field is extra.

Similarly, for mode 2 (L), the specified buffer length 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.

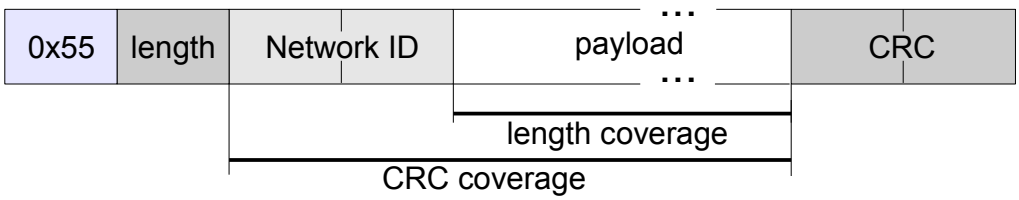
Now for some details.

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, 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 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. [This timeout is probably too long for some 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.]

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 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 packet's field before the physical transmission takes place.

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 packet's field is not modified by the driver.

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 a regular packet. 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 1.5 sec with a slight randomization. The ACK should arrive within 1 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.

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 legitimate 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 automatically 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.

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 variant of the alternating bit protocol at the session level. A praxis that wants to take advantage of this option should include `ab.h` in its header. The module implementing the scheme can be found in `ab.[ch]` in `PicOS/Libs/Lib/`. The functions provided by the module are intended to act as a complete replacement for the `ser_out`/`ser_in` family (for the traditional pure-ASCII UART interface).

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.

Functional description

Here is a typical initialization sequence issued from the praxis:

```
#include "sysio.h"
#include "plug_null.h"
#include "phys_uart.h"
#include "ab.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);
…
```

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 following functions:

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

These function 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** and **ab_outf** (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 (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 with 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 easily get into a stall if a packet gets lost or damaged.

The initial protocol mode, immediately after **ab_init**, is **AB_MODE_PASSIVE**.

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

This function is synonymous with 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.

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

This function writes to the interface the raw string specified as the second argument. Similar to **ab_outf**, the first argument identifies the retry state (the function may block).

The string argument of **ab_out** must have been allocated previously by **umalloc**. The praxis should not touch that string after passing it to **ab_out**, as it will be queued for transmission and later deallocated automatically. If the string to be written is a constant, or it is stored as part of some larger structure (e.g., a VNETI packet), **ab_outf** should be used instead. 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 can be used for 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**.

The second function simply returns the unprocessed input line via a pointer. The string represented by this pointer must be deallocated by the praxis (via **ufree**) when done.

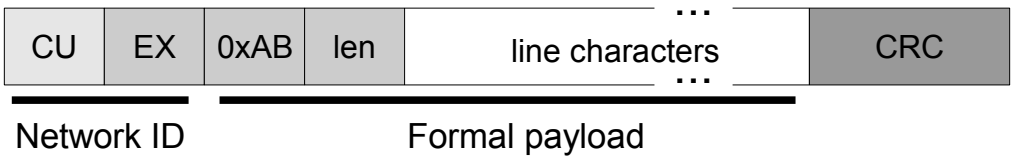
Lines handled by the above functions need no CR/LF terminators: it is assumed that one standard (NULL-terminated) string represents one line. You have to remember that each line sent or received this way must fit into a packet, with the length limitation imposed by

the maximum packet length declaration for the PHY. For example, in the initialization sequence above, the PHY packet length is 96 bytes (note that this does not cover the checksum). XRS uses four initial bytes of the packet for its header; thus, the maximum length of a line that can be passed in the above setup is 92 characters (the NULL sentinel need not be accommodated in the packet).

It is not illegal to try to send lines longer than the PHY-imposed limit, but such lines will be silently truncated to whatever can fit in the maximum-length packet.

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 both 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 an empty line (len = 0). Such a packet, when received, is not interpreted as a data packet (empty lines are never received) 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 normal 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.c`** (they can be redefined easily).

The ufront script

Directory **`Scripts`** includes a generic script named **`ufront.tcl`** which can be used for communication with a praxis employing XRS over UART. Here is the way to call it:

```
ufront.tcl uart_device  bit_rate  max_packet_length
```

e.g.,

```
ufront.tcl 8 115200 96
```

In practically all cases, you will have to specify all three arguments. Without arguments, the script will scan ports **COM1** through **COM9** (on Windows), or **/dev/ttyUSB0** through **/dev/ttyUSB9** and **/dev/tty0** through **/dev/tty9** (on UNIX), stopping at the first that opens and using 9600 bps as the default rate.

The UART device can be specified as a full device name, e.g., **COM8:**, **/dev/tty0**, or a number, e.g., **8** corresponding to **COM8:** on Windows, or **/dev/ttyUSB8** on Linux. In the latter case, the script will also try **/dev/tty8**, if the first guess fails. You can also use the names of virtual (null modem) UART emulators introduced by the *com0com* package (from http://com0com.sourceforge.net/), e.g., **CNCA0**, **CNCB0**, and so on.

The last argument specifies the maximum packet length and should match (exactly) the value used by the PHY in the praxis, i.e., the second argument of **phys_uart**. Otherwise, for some (long) lines the protocol may get stuck, as the packets sent by one party will not be receivable by the other.

Functionally, the script is very simple: it implements a command line interface substituting for a terminal emulator, which, of course, cannot be used directly with the packet-based interface.