UART Communication via VNETI (TCV)

December 2009

Note: modifications with respect to the September 2009 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 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 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/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.

It is recommended to use mode 1 combined with XRS for communicating with those OSS programs whose interface is up to us to shape. This combination provides 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.

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 the 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.

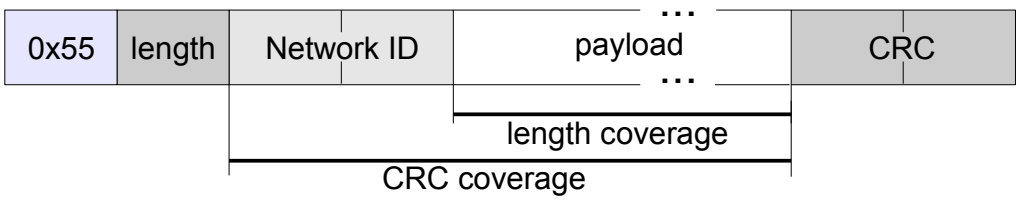
Now for some details.

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.[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

[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.

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.

Diag messages

The driver provides 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), 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 UART is driven in the 'N' mode, 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 characters with code **`0x54`** 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 + 6.

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 **`0x54`** 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, e.g., by running XRS on top of the 'N' mode).

The 'N' mode is the only one of the three modes providing for such a special treatment of diag messages. Note that in the 'L' mode diag messages can be sent directly, as all the OSS program ever sees are straightforward ASCII lines.

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 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 easily get into a stall if a packet gets lost or damaged.

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

```
int 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.

Normally, the function returns zero to indicate a success. If the formatted string turns out to be longer than what can be accommodated in a maximum length packet, the function returns **ERROR** (–1) and does nothing. The maximum acceptable string length, including the terminating NULL byte, is equal to *N* – *6*, where *N* is the PHY buffer size specified as the second argument to **phys_uart** (see above).

```
int 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 identifies the retry state (the function may block). The function returns zero on success and **ERROR** when the string is longer than the maximum payload capacity of a packet (see above).

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.** If the function has succeeded (returned zero), that string will be queued for transmission and later deallocated automatically. If the function fails (returns **ERROR**), it deallocates the string nonetheless. This is to facilitate simple usage where the value returned by the function is ignored by the praxis.

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), **ab_outf** should be 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**.

The second function simply returns the unprocessed input line via a pointer. The string represented by this pointer should 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 zero sentinel must 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.~~

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

```
int abb_out (word st, address buf, word len);
```

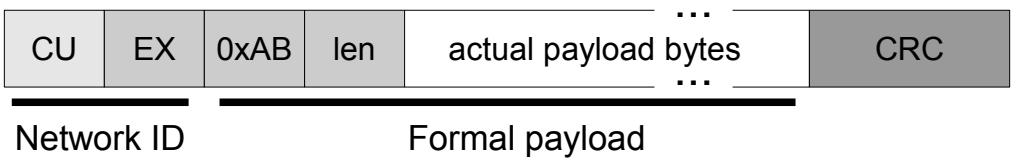
This function sends to the other party **len** bytes from the buffer **buf**. It returns zero on success and **ERROR** when the specified length exceeds the maximum payload capacity of a packet (see above). Similar to **ab_out**, the buffer must have been allocated previously by **umalloc**. If the function has succeeded (returned zero), that buffer will be queued for transmission and later deallocated automatically. If the function fails (returns **ERROR**), it deallocates the buffer nonetheless.

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

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.

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 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**).

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 one way to call it:

```
ufront.tcl -p uart_device -s bit_rate -l max_packet_length
```

e.g.,

```
ufront.tcl -p 8 -s 115200 -l 96
```

In practically all cases, you will have to specify all three arguments. Without arguments, the script will try to open device **CNCA0** (a virtual UART whose second end is available to VUE[2]s **udaemon**) and using 115200 bps as the default rate. It will also assume the default maximum packet length of 56. Note that this number refers to the maximum length available for actual data, i.e., it should equal the argument of **phys_uart** used by the praxis at the other end (see page 8) minus 4.

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.

When called in one of the above-described ways, the script will establish a command line interface, basically acting as a terminal emulator. This assumes that the praxis uses the string-oriented variant of XRS. The chunks of data arriving from the praxis are interpreted as NULL-terminated ASCII strings to be directly shown on the terminal, and vice-versa: any lines typed in by the user will be sent in XRS packets as NULL-terminated character strings. An attempt to enter a string whose total length (including the terminating NULL byte) is larger than the specified maximum length will be signaled as an error and the line will be ignored.

With this simple command:

```
!e or !echo
```

which is interpreted by the script and never sent over XRS (all lines starting with **!** have this property), you can toggle the "echo" flag (which is OFF by default). With the flag ON, the script will echo all input before sending it to the praxis. This is more useful in the binary mode described below.

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. The optional file name points to a file with macro-definitions, i.e., symbolic shortcuts that you may assign to some frequently used sequences of bytes. For example:

```
ufront.tcl -p CNCA1 -s 115200 -l 62 -b my_macros.mac
```

If no macro file is specified, the script will try to open a file named **bin.mac** in the current directory. If that fails, no macros will be used.

An input data line consists of a sequence of bytes specified as a straightforward list of numbers, e.g.,

```
41 x86 $55 0x72 0 13 79 x1 0xf
```

All three special prefixes: **x**, **0x**, and **$**, announce a hexadecimal number. If a number starts with a digit, then it is assumed to be in decimal. The numbers must be non-negative and not larger than 255. No separating blanks are needed if the numbers are separable otherwise, e.g.,

```
41 x86$55 0x72 0 13 79 x1 0xf
```

is OK.

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 value in front of the opening **<** gives the number of bytes in the received block.

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 looks like this:

```
name = body
```


where name must start from 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 way. 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 = "x0D "
LF = "x0A "
```

and suppose that the input line looks like this:

```
1 3 x12 NL $55 CR LF NL
```

After expansion, the line will become:

```
1 3 x12 x0D x0A  $55 x0D  x0A  x0D x0A
```

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.

Macros can have parameters (I don't think this is extremely useful, but you never know), e.g., with this macro:

```
rpt(a,b) = 0xa 0xb 0xaa 0xbb
```

a line looking like this:

```
13 x55 rpt(7,a) 0
```

will become:

```
13 x55 0x7 0xa 0x77 0xaa 0
```

Note that in combination with the multiple passes of macro expansion this may bring in some useful features. The way macro arguments are treated is again purely textual. For each argument (from left to right), all the occurrences of a given formal parameter in the macro body are replaced with the actual value. This is done in the same staged manner as for a macro expansion. Then the modified macro body is applied to the line, exactly as before.