Olsonet Communications

UART Communication via VNETI (TCV)

August, 2008

(preliminary)

Introduction

This document describes the options available for implementing VNETI style communication between a PicOS praxis and the UART. The idea is to allow the praxis to take advantage of VNETI functions, which are normally used for networking. Note that traditionally a 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, looks like an overkill.

The status of the tools described here is preliminary. One feels that some additions to VNETI are in order because of the somewhat ill defined concept of headers an trailers that VNETI plugins are not able to resolve properly. For example, in most cases, UART packets (esp. modes 1 and 2) need no Network ID field. Additionally, mode 2 packets need no CRC. The way we are used to treating VNETI packets, many praxes expect both fields to be present and available. While the praxis doesn't touch the CRC field of an outgoing packet, it has to explicitly reserve room for it when calling tcv_wnp. Upon reception, the CRC field is overwritten by the (radio) PHY with reception characteristics of the packet (RSSI, quality) and made available to the praxis – de facto as part of the payload. While the Network ID was originally intended for the PHY only, some praxes insist on setting it by themselves (my fault) to differentiate among different packet classes.

Theoretically, the plugin can declare the boundary of packet header and trailer and make sure that what the application sees is pure payload. Note, however, that formally both the CRC field as well as Network ID formally fall under the jurisdiction of the PHY rather than the plugin. It is conceivable to have the same plugin handle different PHYs, some of them using the CRC field, some of them not. In particular, neither Network ID nor CRC fields are used by mode 2 of UART interface. Does it make sense to add those fields to the packet format for no reason other than the compatibility with RF interface? I don't think so. Instead, those fields should always be used in strict accordance with their original meanings, and the PHY should provide a method to account for those fields upon packet creation and reception by the praxis. The way things stand now, we will need extra tools to extract reception parameters from RF packets. If it turns out that praxis access to the Network ID field is required, we shall need extra functions to reference it (which will do nothing if that field is absent).

Comments?

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 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 and that would inflate the code size as well as require extra RAM locations.

In order 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.

Assuming that UART_TCV is nonzero, one of the three possible options is selected by setting UART_TCV_MODE to one of the following three values:

```
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 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 (which does not pretend to be a radio packet) with built-in acknowledgments intended for implementing reliable communication. The third mode is the simplest one and can be used to send/receive line-oriented data. It is intended as a replacement for the traditional (ASCII) UART interface.

Mode 1 (P) may not be very useful (I implemented it four years ago, and it has never been used except in tests). Our experience suggests that even when talking to a sophisticated OSS program, it is usually better to implement reliability in the praxis.

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

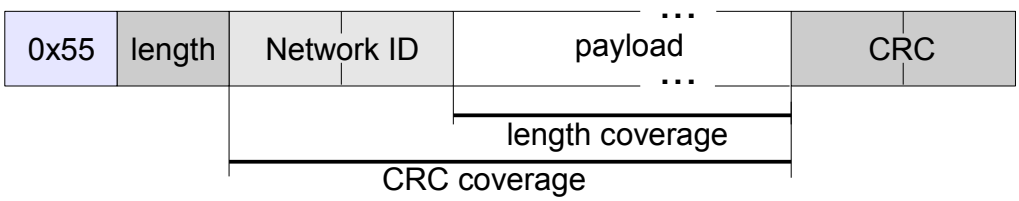
Now for some details.

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 viewed as a preamble. Having detected the 0x55 byte, the receiver reads the next byte, adds 4 to it, and expects that many more bytes to complete the packet. The first two bytes, i.e., the preamble and the length byte are 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. [Note: this timeout is probably too long for some applications. It used to be much shorter, but the Windows implementation of UART (COM ports) introduces occasional long hiccups in the way 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 actual 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 M (bit 0) and A (bit 1).

The exchange is carried out according to the well-known alternating bit protocol. M 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. A indicates the expected value of the M bit in the next packet to arrive from the peer. The initial value of A is also 0.

Having sent a packet, with a given value of M, the peer should refrain from sending another packet, with the opposite value of M, until it receives a packet with the appropriate value of the A bit, i.e., opposite to the one last-sent in M (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 A bit in the next outgoing packet to the pertinent value, i.e., the inverse of the M bit in the last received packet.

If the peer has no handy outgoing packet in which it could acknowledge the receipt of a received packet, 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 be be acknowledged. Its M bit should be always 1, and its A bit should be set to the inverse of the M 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 types of pure ACK packets, their full formats (together with checksums) are listed below:

A = 0 0x01 0x00 0x21 0x10
A = 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 complete message has been received.

Note that regardless how a peer decides to initialize its values of A and M (in particular after a reset, or turning the interface off and back on), the other party will always know which packet (in terms of its M) 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 in 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 need 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 least 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 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 makes sure 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.