Asynchronous Serial Communication Protocol (without Flow Control) using TLM 2.0

(Example of Non memory based protocol)

Developed by:
Manish Aggarwal, Ruchir Bharti
Circuitsutra Technologies Pvt Ltd

16th April 2009

Table of Contents

1 Purpose

The purpose of this document is to propose TLM2 based Asynchronous Serial communication protocol which can be used to model industry standard serial interfaces like UART Model. As these protocol are non memory mapped based, thus the existing TLM2 features are not sufficient to develop such Asynchronous Serial based model. So, this document should define the necessary payload (and its fields) with commands and response status required for the Asynchronous Serial communication protocol(without Flow Control).

2 Overview

A "asynchronous receiver/transmitter", which translates data between parallel and serial forms. UART is most common example used to explain such protocol where as application they are commonly used in conjunction with other communication standards such as RS-232(Recommended Standard 232).

The such communication devices performs serial-to-parallel conversion on data characters received from a peripheral device or a MODEM, and parallel-to-serial conversion on data characters received from the CPU Bus. The CPU Bus can read/update the complete status at any time during the functional operation. The connection between Serial devices and CPU BUS is shown in Fig1.

Since serial communication is comparatively slower, so to have higher simulation speed all communication ports gets abstracted such that they will share only the high level data to connected device.

Asynchronous Transmission/Reception

In asynchronous transmitting, SERIAL devices normally send a "start" bit, five to eight data bits, least-significant-bit first, an optional "parity" bit, and then one, one and a half, or two "stop" bits. The start bit is always a **0** (logic low), which is also called a **space**. The start bit signals the receiving DTE (Data Terminal Equipment**)** that a character code is coming. After the transmission of next five to eight bits and the optional parity bit, Device(transmitter) send one or two bits are always a **1** (logic high) condition and called the stop bit(s). They provide a "rest" interval for the receiving DTE so that it may prepare for the next character which may be after the stop bit(s). All operations of the SERIAL hardware are controlled by a clock signal which runs at a multiple (say, 16) of the data rate - each data bit is as long as 16 clock pulses.

Following errors could be occurred (depending on the controller) while transmitting or receiving a character:-

1. **Overrun error**:- When the next character is arrived before the previous one is read. It can be used is also an indication to transmitter to slow down. In the model it can be shown by keeping a flag in the Receiver.
2. **Underrun error**:- When the transmitter buffer is empty that is there is no character to transmit then this flag is raised. A flag can be kept in the transmitter to show this.
3. **Framing error**:- When the receiver does not get the valid start or stop bit, it produces framing error. For the model purpose, this is generated when the receiver is expecting the stop bit but does not get the stop bit.
4. **Parity Error**:- If parity does not matches. Again a parity error flag can be kept in the receiver.
5. **Break Condition**:- After sending a bit, if a character time is passed but the receiver does not get the next bit, it produces break condition. This is not very relevant for implementing the functional model as all the bits can be sent in a single function call.

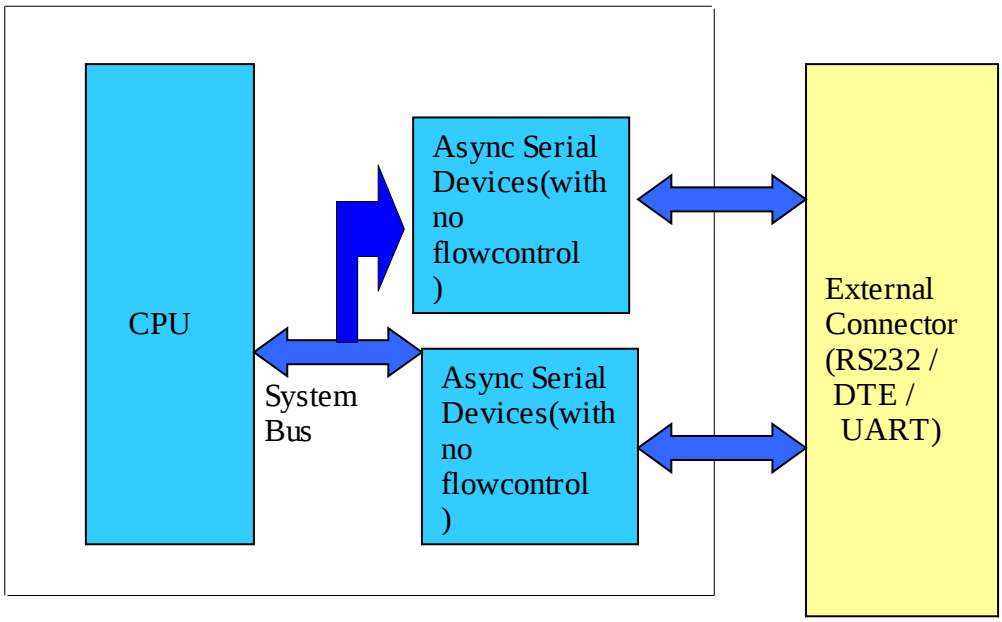

Fig1. Connecting the Async Serial device with the CPU BUS and the external connectors

3 SERIAL Model Connectivity

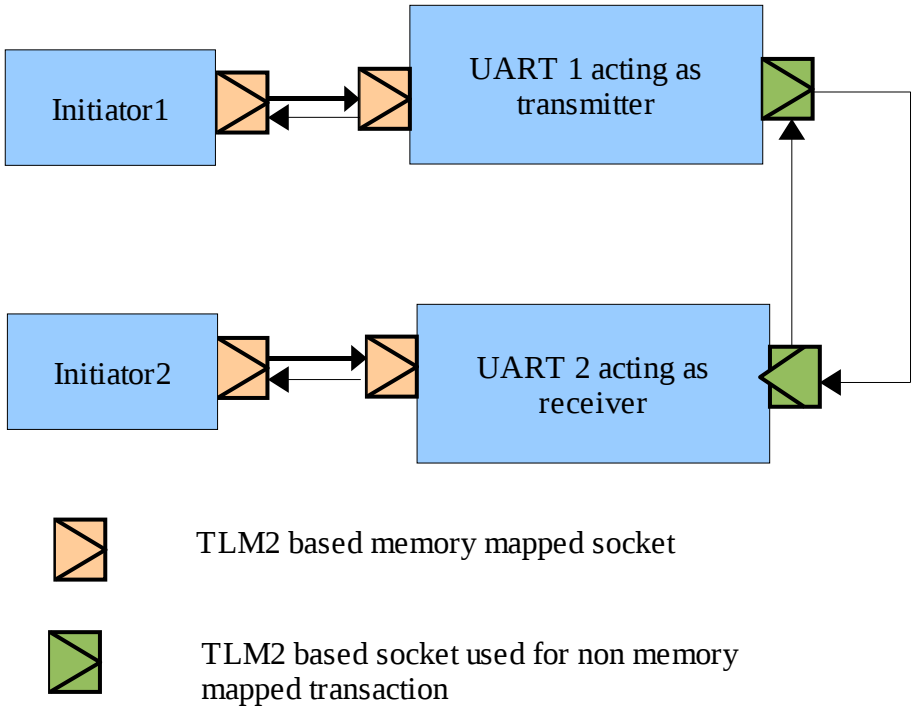

Fig2. Connecting the2 Async Serial device with its initiator

In this model an SERIAL devices are connected with TLM2based socket, one acts as transmitter and another as receiver. The socket between initiator1 and SERIAL device1 are simple memory based socket to read or write the serial device registers by initiator. As the SERIAL protocol is non memory mapped, thus the TLM2 socket can not be used as it is while communicating between two devices. Thus a new `tlm_serial_protocol_types` based protocol (using `tlm_serial_payload, tlm_serial_phase`) is defined (see section 4) and used for the communication between two Serial devices like UARTs.

Socket Decleration

The non memory mapped TLM2 based socket can be declared as follows:

```
struct tlm_serial_protocol_types
{
  typedef tlm_serial::tlm_serial_payload tlm_payload_type;
  typedef tlm_serial::tlm_serial_phase tlm_phase_type;
};

simple_initiator_socket<Initiator, 32, tlm_serial_protocol_types>
init_socket;
simple_target_socket<target, 32, tlm_serial_protocol_types> tar_socket;
```

4 tlm serial payload

For the purpose of Asynchronous Serial(No Flow Control) communication, a new generic payload has been developed which has specific fields like:

Fields	type	Description
`m_command`	enum	There are 2 command supported for AsyncSerial payload transmission : `SERIAL_SEND_CHAR_COMMAND`, `SERIAL_BREAK_CHAR_COMMAND.`
`m_data`	uint8_t *	The pointer to data array. Each element is of type unsigned char which represents the data bits of a character sent in the Asynchronous Serial transmission
`m_length`	uint32_t	Number of characters in the data arary
`m_response_status`	enum	Response status, when a module receives the packet(default : `SERIAL_OK_RESPONSE`) .
`m_enable_bits`	uint8_t	Each bit in this field is to enable an option as defined below.
`m_parity_bit`	Bool*	Array of bools. Length of array equals to m_length. Each element specify the parity bit corresponding to each character in m_data field. Also this field is valid only if `m_enable_bits[0]` is set.
`m_valid_bits`	uint8_t	Number of valid bits in each character data sent. Should be between 5-8, depending on the register settings. In case this fields get mismatched with received frame then controller can ignore the received packet.
`m_stop_bits`	uint8_t	Number of stop bits in the end of a character. Permissible values are 1, 2 or 3 (3 represents 1.5 stop bits). In case this fields get mismatched with received frame then controller can ignore the received packet or generates Frame Error at target end.
`m_baudrate`	uint32_t	The baud rate at which the communication is taking place. So in case of target baud-rate is different this value then target can discard all further incoming data till the baud-rate values matched again.

Field Values:

m_response_status is of type serial_response_status (enum)

```
enum serial_response_status {
    SERIAL_OK_RESPONSE = 1,
    SERIAL_ERROR_RESPONSE = 0,
};
```

Note:- Here default response is `SERIAL_OK_RESPONSE`, where it is recommended that target should not change this response. Since in serial link there are no acknowledgement shared between 2 communication blocks and if target changes the response to `SERIAL_ERROR_RESPONSE` which indicates that serial link will break. And no further communication will happen.

serial_response_status fields	Explanation
SERIAL_OK_RESPONSE	If the transaction is successfully delivered (without any errors).
SERIAL_ERROR_RESPONSE	If any other error occurred then error response will be generated which will cause the break of serial link and no further communication will happen.

m_enable_bits is to enable some options as explained in the table below:

Bit No	Explanation
0	To enable the parity. Field m_parity_bit is valid if this bit is set to 1.
1	Field m_valid_bits is valid if this bit is set to 1.
2	Field m_stop_bits is valid if this bit is set to 1.
3	Field m_baudrate valid if this bit is set to 1.

For convenience following enum is defined:

```
enum serial_enable_flags{
 SERIAL_PARITY_ENABLED=0x1,
 SERIAL_VALID_BITS_ENABLED=0x2,
 SERIAL_STOP_BITS_ENABLED=0x4,
 SERIAL_BAUD_RATE_ENABLED=0x8
};
```

m_command is enum type, which state type of payload send for communication :

```
enum serial_command {
     SERIAL_SEND_CHAR_COMMAND,
     SERIAL_BREAK_CHAR_COMMAND
};
```

5 tlm serial phase

New phases are defined for Serial Asynchronous(No Flow Control) protocol to communicate. The phases that would be added are :-

BEG_TRANSMISSION

As there is no acknowledgement that has to be received by the transmitter, there is only one phase that needs to be send. One or more than one characters can be sent depending on the modelling methodology.

Note : User as extended phases as per requirements, but currently only BEG_TRANSMISSION phase is supported.

6 Communication Through Blocking Interface

Uart generic payload can be used for the communication between the transmitter and receiver through blocking as well as non blocking interface. A b_transport function can be called to transmit many characters in one go.

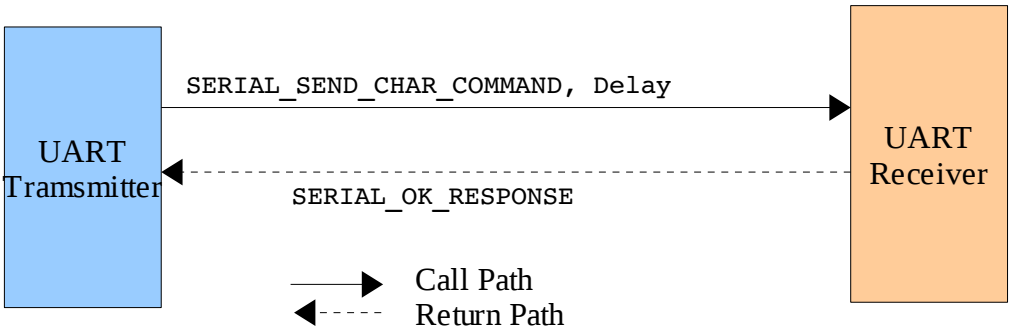

Fig3. Connecting the Async Serial devices like UART with frw and return path response

Some points:

- `SERIAL_SEND_CHAR_COMMAND` can be used to send multi characters at a single go using blocking transport calls.
- If a user wants to send the single character the `m_length` will have the value 1.
- All the fields `m_data, m_length, m_parity_bit, m_valid_bits` and `m_stop_bits` should be set to the appropriate values.
- `m_stop_bits` specifies the stop bits accompanying each character. Thus, may be many characters are send through a single function call, for all those characters it is assumed that the number of stop bits equals to `m_stop_bits.`
- `m_parity_bit` contains the elements equal to `m_length.` Each element specify the parity bit for each character in `m_data` array.
- Each command sent have only two timing points, start and end.
- With each transaction the delay is sent. The delay parameter signifies the transmission delay of all the bits passed. Delay depends on the baud rate of the transmission.
- It is recommended that Transmitter should not generated Break character under `SERIAL_SEND_CHAR_COMMAND`, but new payload will be generated with `SERIAL_BREAK_CHAR_COMMAND.`

Timing Diagram For Blocking Calls :

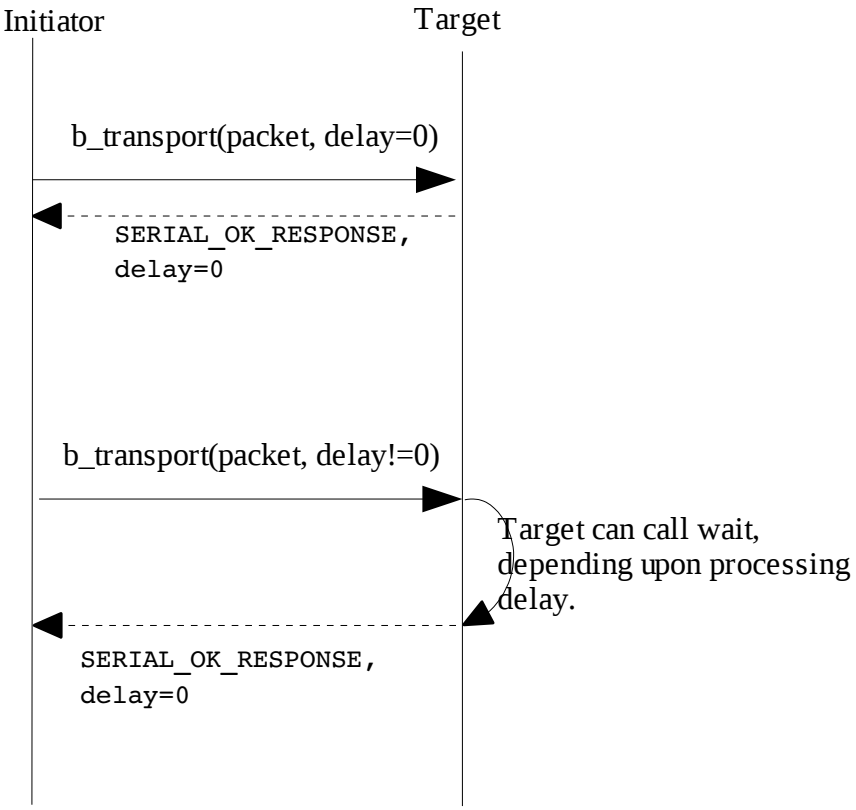

Fig4.1. Timing Diagram between two serial device showing execution flow of packet when target is calling wait..

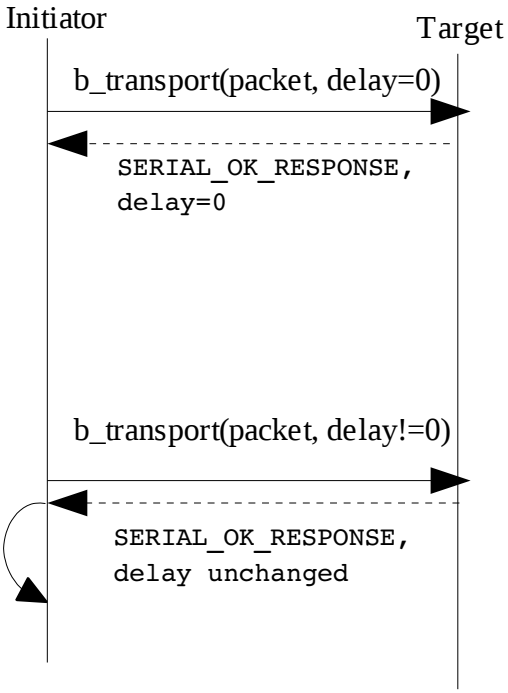

Fig4.2. Timing Diagram between two serial device showing execution flow of packet when Initiator is calling wait..

7 Communication through Non Blocking Interface

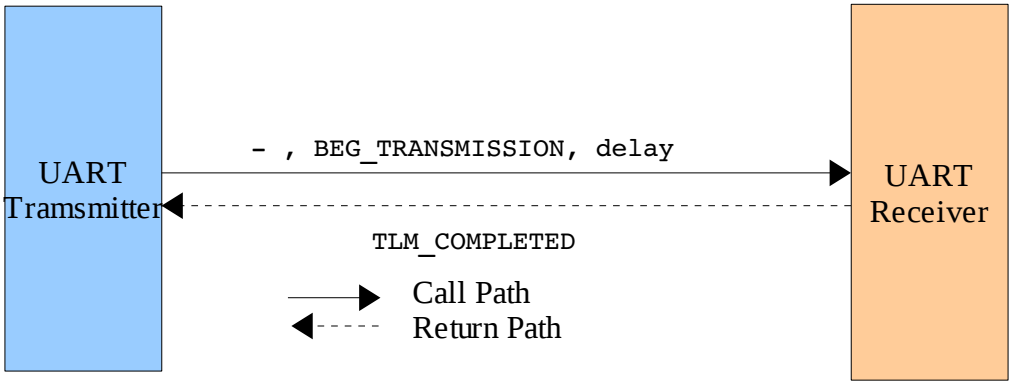

Fig5. Connecting the Serial devices like UART with frw and return path response using Non-Blocking interface.

Some Points:

- As Asynchronous Serial communication is a simple protocol, thus the non blocking calls shrinks just to blocking calls only difference being that the target can call wait in the blocking calls while no wait can be called in non blocking calls.
- Depending on the requirement transmitter can send one or more character in a single non blocking call.
- ***delay*** parameter in the non blocking transport call signifies the transmission time of a bit(s) passed.
- delay parameter can also be used by the receiver to send back the time passed in the transmission of delay, so that transmitter may or may not call wait depending on the requirement. Also on return path this delay should not be changed by receiver.
- `TLM_COMPLETED` is send by the receiver to mark the end of transaction.
- `SERIAL_OK_RESPONSE` will not be changed by receiver if its changed then serial link will break.
- At present there is only single phase is supported ie `BEG_TRANSMISSION` but user can extends phases depending upon requirements.

Timing Diagram For Non-Blocking Calls :

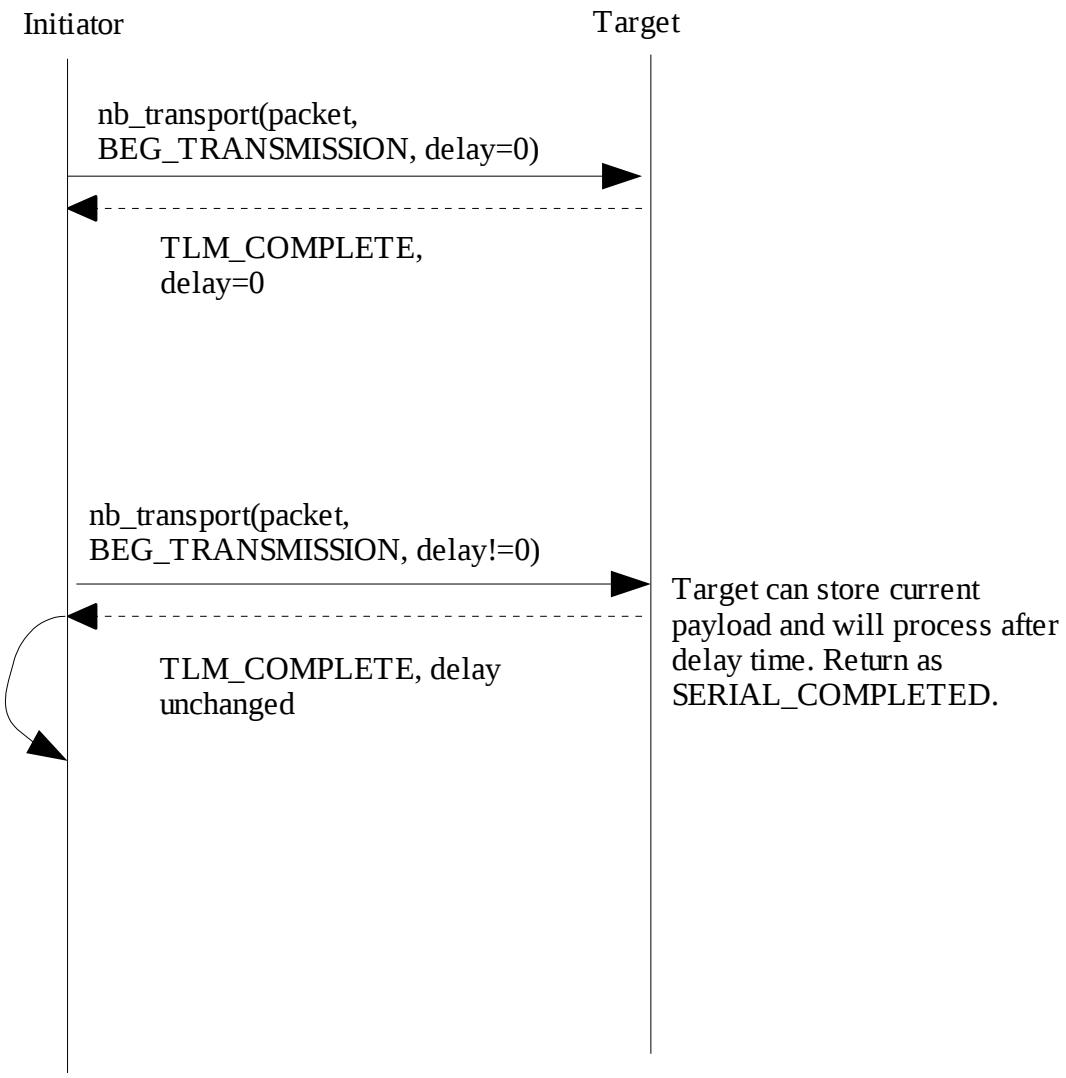

Fig6. Timing Diagram for Non-Blocking interface between two serial device showing execution flow of packet..

8 Illustration Example / Use Cases

(1)Transfer 1 or more Byte with Higher Abstraction level

This example shows the scenario when Tx wants to send byte to Rx at higher abstraction..

```
// For blocking Calls, where initiator is calling wait
void my_thread(){
        uint8_t data[SIZE]; // SIZE is defined by initiator
        tlm_serial_payload sp;
        sp.set_data_ptr(&data[0]);
        sp.set_data_length(5); // 5 denotes number of valid bytes in array in case of single byte
                               // transfer this could be 1
        sp.set_baudrate(bit_time);
        ... //set up other stuff
        socket->b_transport(sp , delay=0);
        wait(5*10*bit_time);  //8 bit data 2 bit start/stop
        ...
        ...
        // update payload
        socket->b_transport(sp , delay=0);
        wait(5*10*bit_time);  //8 bit data 2 bit start/stop
        ...
        ...
}

// For Non-blocking Calls
void my_thread(){
        uint8_t data[SIZE];
        tlm_serial_payload sp;
        sp.set_data_ptr(&data[0]);
        sp.set_data_length(1); // single byte transfer call.
        sp.set_baudrate(bit_time);
        ...
        ... //set up other stuff
        ...
        socket->nb_transport_fw(sp , BEG_TRANSMISSION, ZERO_TIME); //simplified not
compilable
        wait(10*bit_time);  //8 bit data 2 bit start/stop
        ...
        ...
        ...
}
```

(2)Transfer 1 or more Byte with Lower Abstraction level

This example shows the scenario when Tx wants to send bulk byte to Rx at lower abstraction..

```
void my_thread(){
        uint8_t data[SIZE];
        tlm_serial_payload sp;
        sp.set_baudrate(bit_time);
        ... //set up other stuff
        // here chunk of 5 bytes gets split into single transactions to have intermediate points
        // between each transfer bytes
        for (unsigned int i=0; i<5; i++)
        {
                sp.set_data_ptr(&data[j]);
                sp.set_data_length(1);
                socket->nb_transport_fw(sp , BEG_TRANSMISSION, ZERO_TIME); //simplified
not compilable
                wait(10*bit_time);  //8 bit data 2 bit start/stop
        }
        ...
        ...
        // update payload
        sp.set_data_ptr(&data[0]);
        sp.set_data_length(1); // single byte transfer call.
        socket->nb_transport_fw(sp , BEG_TRANSMISSION, ZERO_TIME); //simplified not
compilable
        wait(10*bit_time);  //8 bit data 2 bit start/stop
        ...
        ...
}
```