ZeBu UART Transactor User Manual

Version V-2024.03-SP1, February 2025

SYNOPSYS®

Copyright and Proprietary Information Notice

Contents

Contents

Contents

Preface

The ZeBu®UART Transactor User Manual describes how to use the ZeBu UART Transactor while emulating your design in ZeBu.

Related Documentation

For more information about the ZeBu supported features and limitations, see ZeBu Release Notes in the ZeBu documentation package corresponding to your software version.

For more information about the usage of the present transactor, see Using Transactors in the training material.

Typographical Conventions

This document uses the following typographical conventions:

To indicate	Convention Used
Program code	*OUT <= IN;*
Object names	*OUT*
Variables representing objects names	*<sig-name>*
Message	Active low signal name '<sig-name>' must end with _X.
Message location	*OUT* <= IN;
Reworked example with message removed	*OUT_X* <= IN;
Important Information	**NOTE:** This rule...

The following table describes the syntax used in this document:

Syntax	Description
[ ] (Square brackets)	An optional entry

Syntax	Description
{ } (Curly braces)	An entry that can be specified once or multiple times
| (Vertical bar)	A list of choices out of which you can choose one
... (Horizontal ellipsis)	Other options that you can specify

Synopsys Statement on Inclusivity and Diversity

Synopsys is committed to creating an inclusive environment where every employee, customer, and partner feels welcomed. We are reviewing and removing exclusionary language from our products and supporting customer-facing collateral. Our effort also includes internal initiatives to remove biased language from our engineering and working environment, including terms that are embedded in our software and IPs. At the same time, we are working to ensure that our web content and software applications are usable to people of varying abilities. You may still find examples of non-inclusive language in our software or documentation as our IPs implement industry-standard specifications that are currently under review to remove exclusionary language.

1

About the Manual

The ZeBu UART transactor implements a serial RS232 interface. It contains the TXD/RXD data signals and CTS/RTS control signals as described in the EIA-232 (RS-232) standard at the logical level. This transactor is configurable to support the various operation modes of the RS232 protocol.

Figure 1   ZeBu Transactor Overview
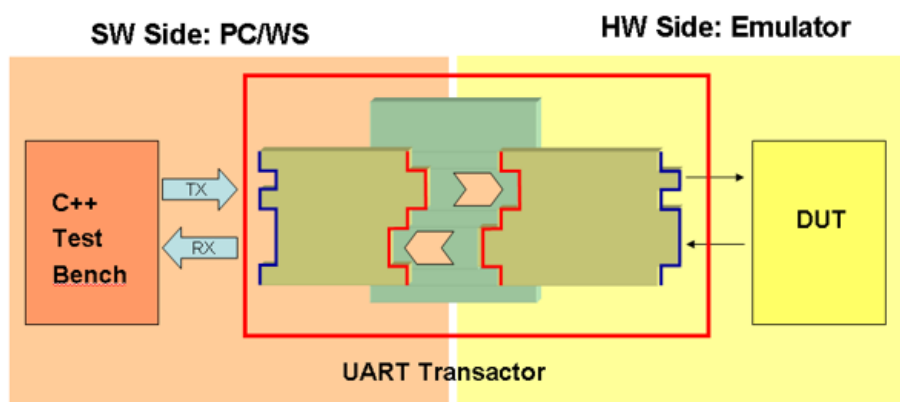
This section explains the following topics:

- Supported Protocol Specifications
- Supported Features
- Components of UART Transactor
- Performance
- Limitations

Supported Protocol Specifications

This UART implementation supports, according to its configuration, 5-bit to 8-bit data with or without parity (odd or even) and 1 or 2 stop bits. The baud rate can be set as a fraction of the reference clock frequency (1 – 1/16777215).

Figure 2   ZeBu UART Transactor Detailed Architecture
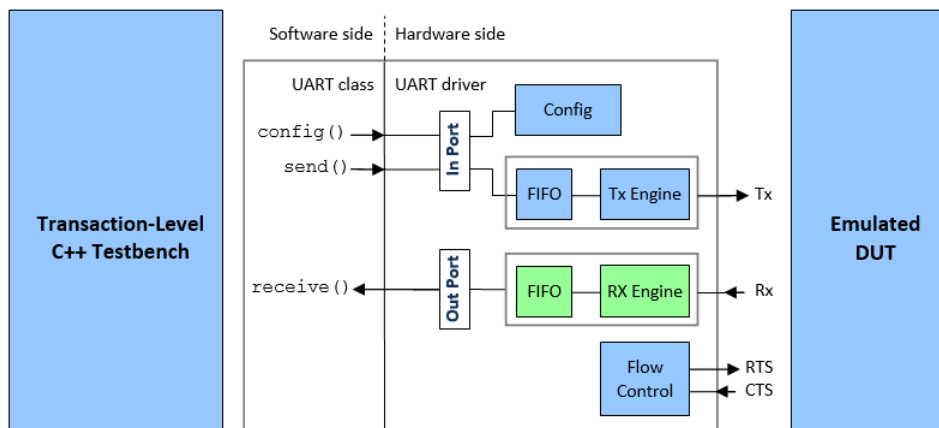
The ZeBu UART transactor proposes different operating modes:

- An application mode where the application testbench running on the ZeBu workstation reads/writes characters from/to the RS232 DUT interface.

- A server mode with remote connection via a TCP socket. This interface mechanism supports remote user application drivers running on Linux or Windows® operating systems.

- A ZeBu UART transactor with an interactive terminal interface. This feature launches an xterm terminal that communicates with the UART transactor. The data exchange between the UART interface and the terminal are automatically handled by the transactor.

Supported Features

Protocol Features

The UART transactor supports the following protocol related features:

- Data width of 5 - 8 bits

- One or two stop bits

- Parity bit

- RTS/CTS handshaking in full duplex mode

Verification Features

The UART transactor supports the following verification related functions:

- UART transactor functionality log files and Traffic log files

- Debug Probe

- Timestamp feature

- Dynamic Reset

- Auto detection of different baud rate

- Parity error injection

Components of UART Transactor

Once the ZeBu UART transactor is installed, the following are the major components of UART Transactor under ZEBU_IP_ROOT/XTOR/xtor_uart_svs.<version>:

- Driver (xtor_uart_svs.sv)

- Documentation: Contains the Transactor documents.

- lib: Contains .so libraries of the transactor.

- Example: Contains Testbench, DUT, and environment files with the makefile necessary to run the transactor examples described in the Tutorial chapter of this manual.

- include: Contains .hh header files of the transactor.

- Vlog/vcs: Contains protected verilog source code for the transactor.

The following is the transactor file tree after package installation:

```
$ZEBU_IP_ROOT
`--XTOR
`-- xtor_uart_svs.<version>
+-- bin
+-- doc
¦ +--ZeBu_XTOR_UART_svs_UM.pdf
¦ +-- foss
¦ ¦ +-- ZX-XTOR-Library_<ver>_FOSS.pdf
+-- example
¦ +-- pc_files
¦ +-- src
¦ ¦ +-- bench
¦ ¦ ¦ +-- zRci
¦ ¦ +-- dut
¦ ¦ +-- env
```

```
¦ +-- sverilog
¦ ¦ +-- src
¦ ¦ ¦ +-- bench
¦ ¦ ¦ +-- dut
¦ ¦ ¦ +-- env
¦ ¦ +-- zebu
¦ +-- zebu
+-- include
¦ +-- xtor_uart_svs.hh
+-- lib
+-- vlog
+-- vcs
```

During installation, symbolic links are created in the following directories for an easy access from all ZeBu tools:

- $ZEBU_IP_ROOT/vlog

- $ZEBU_IP_ROOT/include

- $ZEBU_IP_ROOT/lib

Performance

The hardware synthesized BFM part of the UART transactor uses:

- 751 registers and 706 LUTs in the Virtex 7 technology

- 751 registers and 809 LUTs in the Virtex 8 technology.

This transactor supports baud rates up to 10 Mbps full duplex for user application.

Limitations

The current version of this transactor has the following limitations:

- The ZeBu UART transactor is only a Data Terminal Equipment (DTE) device.

- The software flow control (XON/XOFF) is not implemented.

- RTS/CTS handshaking in half-duplex mode is not supported.

2

Establishing an Interface between UART Transactor and DUT

This section explains the following topics;

- Overview

- Signal List

- Instantiating the Hardware Drivers

- Connecting the Design Clocks

- Connecting the Transactor's Reset

Overview

Instantiating a hardware driver requires establishing a connection between then DUT and transactor. RxD/TxD signal of DUT should be connected with TxD/RxD signal of UART transactor. If flow is to be controlled, then CTS/RTS of DUT should be connected with RTS/CTS of UART transactor. Reset signal of UART transactor is active low and should toggle from assert to deassert when clock is running. The following figure illustrates the connection between the DUT and the transactor:

Figure 3   ZeBu UART Transactor Hardware Interface
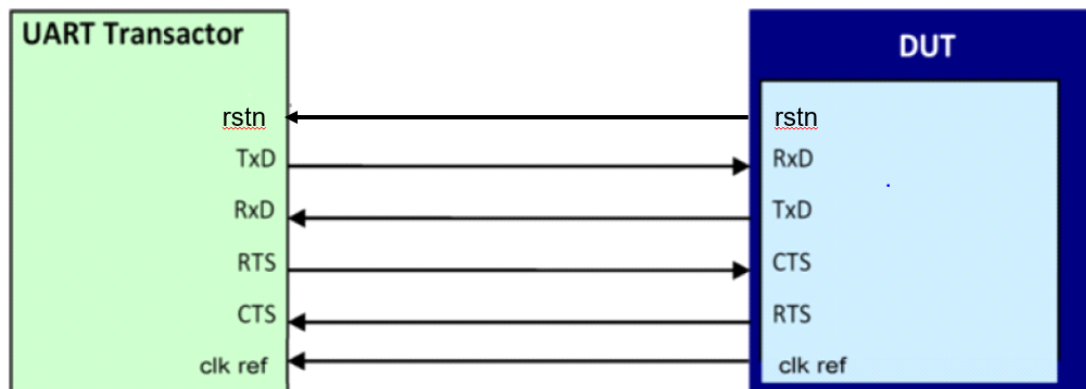
Ensure the following for debugging purpose of connections:

- Reset is active low signal.

- If RTS and CTS is not controlled by SoC then, Input CTS should be driven 0.

- Clock should be running before reset is applied.

Signal List

The following table list the signals for UART transactor hardware interface:

Table 1   List of Signals for ZeBu UART Transactor Hardware Interface

Symbol	Size	Type(Transactor)	Type (DUT)	Description
Clk_ref	1	Input	Output	Reference clock to calculate the ratio.
rstn	1	Input	Output	Synchronous reset
RxD	1	Input	Output	Serial Receive Data (to transactor).
TXD	1	Output	Input	Serial Transmit Data (from Transactor)
RTS	1	Output	Input	Ready To Send.
CTS	1	Input	Output	Clear To Send.

CTS (Clear To Send) and RTS (Request To Send) are active low control signals of the transactor for enabling the Tx and Rx flow, respectively.

This means that the CTS input should be low when the Tx flow from the transactor is required and RTS is driven low when transactor is to receive data from the DUT.

Instantiating the Hardware Drivers

The design wrapper is the top-level Verilog file in which the DUT and transactors are instantiated. The wrapper does not have any I/O ports. An example wrapper is present in the example/src/dut directory of the transactor and can be used as a reference for creating a design wrapper. The following example shows the instantiation of the ZeBu UART Transactor in the design wrapper. (Save and Restore feature is not used).

```
zceiClockPort clockPort0(
.cclock         (clk),
.cresetn        (rstn)
;

xtor_uart_svs uart_driver_0(
.CTS(DUT_RTS),
.RTS(DUT_CTS),
.TxD(DUT_RxD),
.RxD(DUT_TxD),
clk_ref(clk)
.rstn(rstn),
);
```

To create a connection between the DUT and the transactor, perform the following

steps:

1. Create a top-level, hw_top module. An example wrapper is present in the example/src/
   dut directory of the transactor and can be used as a reference for creating a design
   wrapper.

2. Instantiate following mandatory components inside hw_top module

   ◦ XTOR: xtor_uart_svs module

   ◦ DUT

3. zceiClockPort for Clock

Connecting the Design Clocks

The controlled clock can be one of the following:

• the UART controller DUT clock (if the clock is a primary clock).

• the DUT primary clock driving the UART controller clock.

Connecting the Transactor's Reset

The UART transactor supports synchronous reset, which means the clock should be
running while reset is applied. For any synchronous design, if reset is not applied properly,
the behavior is totally indeterministic.

If required, the reset should be delayed by few clock cycles so that all required zemi3
initializations are completed successfully to avoid initialization issues.

3

Instantiating and Accessing the Transactor

This section explains the following topics:

Interface Description

The ZeBu UART transactor can be instantiated and accessed through a C++ interface defined in the *$ZEBU_IP_ROOT/include/xtor_uart_svs.hh* file.

The ZeBu UART transactor's API provides the Uartxtor_uart_svs API class .

This API is included in the *ZEBU_IP::XTOR_UART_SVS* namespace for the three classes.

Example:

A typical testbench starts with the following lines:

```
#include "xtor_uart_svs.hh"
using namespace ZEBU_IP
using namespace XTOR_UART_SVS
```

Class Description

The ZeBu UART transactor interface can be driven and implemented in the following C++ class:

Table 2   C++ Class for the ZeBu UART Transactor

Class	Description
xtor_uart_svs	class providing a low level interface (data send and receive).

You can choose to operate the transactor in different modes by configuring the API setOpMode().

Note:

> The API setOpMode() should always be called in the main thread and not in any other thread. It should be called serially before clocks start.

The default mode is the Normal mode where you use the low level API (send/receive). You can set the operating mode to ServerMode or XtermMode.

Common Interface

The common interface is a set of methods common to different mode, that is,*normal,ServerMode,XtermMode* classes. The following table lists these common methods:

Table 3   List of Methods Common to all modes

Method	Description
Transactor Initialization and Configuration see Transactor Initialization and Configuration	
init	Connects the UART transactor to the ZeBu system.
setWidth	Sets the data word length.
setParity	Sets the parity.
injectParityError	Enables inject Parity Error with Transmitted data.
setStopBit	Sets the number of stop bits.
setRatio	Sets the clock divider ratio.

Table 3   List of Methods Common to all modes (Continued)

Method	Description
adjustRatio	Adjusts the clock divider ratio to the value detected by the transactor.
setBdRDetectWindow	Sets the baud rate detect window.
config	Sends the configuration of the UART transactor communication mode.
getWidth	Obtains the data width.
getParity	Obtains the parity.
getStopBit	Obtains the number of stop bits.
getRatio	Obtains the clock divider ratio.
getBdRDetectWindow	Obtains the baud rate detect window.
getDetectedRatio	Obtains the clock divider ratio detected by UART transactor
getClkCycle	returns the clock count of the last hardware to software communication of UART
Transactor Logging see Transactor Logging	
setName	Sets the transactor name shown in the messages.
getName	Obtains the transactor name shown in the messages.
setDebugLevel	Sets the messages debug level.
setLog	Sets the messages log file or stream.
Transactor Dumping see Transactor Dumping	
dumpSetRxPrefix	Sets the received data prefix in the dump file.
dumpSetTxPrefix	Sets the transmitted data prefix in the dump file.
dumpSetDisplayErrors	Enables dumping of the received erroneous data.
dumpSetFormat	Sets the dump format.
dumpSetDisplay	Sets the dump display.
dumpSetMaxLineWidth	Sets the maximum line width in the dump file.

Table 3   List of Methods Common to all modes (Continued)

Method	Description
dumpGetRxPrefix	Obtains the dump file received data prefix.
dumpGetTxPrefix	Obtains the dump file transmitted data prefix.
dumpGetDisplayErrors	Obtains the erroneous data handling configuration.
dumpGetFormat	Obtains the dump format.
dumpGetDisplay	Obtains the dump display.
dumpGetMaxLineWidth	Obtains the dump file maximum line width.
openDumpFile	Opens the dump file.
closeDumpFile	Closes the dump file.
dumpSetRxSuffix	Sets the received data suffix in the dump file.
dumpSetTxSuffix	Sets the transmitted data suffix in the dump file.
dumpGetRxSuffix	Obtains the dump file received data suffix.
dumpGetTxSuffix	Obtains the dump file transmitted data suffix.
Runtime clock control	
runClk	advanced transactor clock and also service the Tx Callbacks
runUntilReset ()	Wait for reset de-assertion
Timestamp/Cycle Count Printing	
enableTimestamp	enables printing timestamp in case of RTL CLK used and cycle count in case zCei clockport is used.
dumpSetDataTS	enables dumping of Timestamp value along with data in data logs.

Transactor Initialization and Configuration

This section describes the methods used to initialize and configure the transactor.

setOpMode ( ) Method

This method sets the operating mode for the ZEBU UART transactor. By default, mode is defaultMode, which uses low level APIs. However, you can also select ServerMode or XtermMode. Ensure to configure this API before the init () call as shown below:

```
bool   setOpMode (UartOperationMode_t mode) ;
```

init() Method

This method connects the Zebu UART transactor to the ZeBu system. After calling the *Board::open()* method, *init()* method must be called first before performing any configuration or operation on the ZeBu UART transactor.

```
void init (Board *zebu, const char *driverName);
```

where:

- *zebu* is the Zebu system identifier.

- *driverName* is the driver instance name in the design wrapper file.

setWidth() Method

This method sets the data word length, that is, the numbers of bits per data. The *config()* method must be called to make the setting effective in the hardware.

```
bool setWidth   (uint8_t width);
```

where, *width* is the data word length in number of bits. It can range from 5 through 8 (default).

The method returns:

- *true*, if successful.

- *false*, if the value is an invalid parameter.

setParity() Method

This method sets the parity bit generation and check. The *config()* method must be called to make the setting effective in the hardware.

```
bool setParity   (Parity_t parity);
```

where, *parity* is of *Parity_t* enumeration type as follows:

- *NoParity* (default): No parity bit is generated.

- *OddParity*: A parity bit is generated and set to one if the data contains an odd number of one bits; the incoming data parity bit is checked.

- *EvenParity*: A parity bit is generated and set to one if the data contains an even number of one bits; the incoming data parity bit is checked.

The method returns:

- *true*, if successful.

- *False*, if the parity value is an invalid parameter.

injectParityError() Method

This method is used to enable inject Parity Error with Transmitted data. Parity errors can be randomly injected into transmitted data by setting err_probability argument from 0 to 100, which determines the probability of parity error injection in data packets.

This API is to be used only when Parity is set either to even parity or odd parity otherwise it will show error.

```
void injectParityError (uint32_t err_probability)
```

where, err_probability sets the probability value for inject parity error with each transmitted data.

- 0 for no parity error injecting with the transmitted data.

- 100 for parity error injection with each transmitted data.

setStopBit() Method

This method sets the number of stop bits. The *config()* method must be called to make the setting effective in the hardware.

```
bool setStopBit (StopBit_t stop);
```

where, *stop* is of *StopBit_t* enumeration type and defined as follows:

- *OneStopBit*: the transactor generates one stop bit at the end of each data.

- *TwoStopBit* (default): the transactor generates two stop bits at the end of each data.

The configuration does not have any impact on incoming data, because only one stop bit is needed and then two stop bits are also supported.

The method returns:

- *true* if successful.

- *false* if the stop value is an invalid parameter.

setRatio() Method

This method sets the clock divider ratio. The clock divider ratio defines the baud rate. For details on the baud rate calculation, see Baud Rate Calculation.

The *config()* method must be called to make the setting effective in the hardware.

```
bool setRatio  (uint ratio);
```

where, *ratio* is the clock divider ratio. It can range from *1* (default) through *0xFFFFFF*.

The method returns:

- *true*, if successful.

- *false*, if the ratio value is an invalid parameter.

adjustRatio() Method

This method adjusts the clock divider ratio to the value detected by the ZeBu UART transactor.

```
uint adjustRatio (void);
```

The method returns the current clock divider ratio.

setBdRDetectWindow() Method

This method sets the minimum number of Rx edges of the baud rate detect window to validate the new upward ratio detected in the transactor.

The *config()* method must be called to make the setting effective in the hardware.

```
bool setBdRDetectWindow  (uint8_t n);
```

where, *n* defines the minimum number of *Rx* edges as follows:

number of Rx edges = n x 16.

n can range from 1 through 15.

By default, the baud rate detect window is set to 16 (n = 1).

The method returns:

- *true*, if successful.

- *false*, if the n value is an invalid parameter.

config() Method

This method sends the configuration parameters (defined with *setWidth()*, *setParity()*, *setStopBit()*, *setRatio()*, and *setBdRDetectWindow()*) to the UART transactor and starts the transactor BFM clock. This method applies these parameters to the hardware.

```
bool config  (void);
```

The method returns *true* if successful; otherwise, *false*.

getWidth() Method

This method returns the current data word length setting.

```
uint8_t getWidth  (void);
```

getParity() Method

This method returns the current parity setting.

```
const_char* getParity  (void);
```

Possible returned values are:

- *NoParity*: No parity bit.

- *OddParity*: Odd parity.

- *EvenParity*: Even parity.

getStopBit() Method

This method returns the current number of stop bits.

```
const_char* getStopBit  (void);
```

Possible returned values are:

- *OneStopBit*: 1 stop bit.

- *TwoStopBit*: 2 stop bits.

getRatio() Method

This method returns the clock divider ratio setting corresponding to the current baud rate.

```
uint getRatio (void);
```

getBdRDetectWindow() Method

This method returns the number of Rx edges to validate the new upward ratio detected in the transactor:

```
uint getBdRDetectWindow  (void);
```

getDetectedRatio() Method

This method returns the clock divider ratio detected by the ZeBu UART transactor.

```
uint getDetectedRatio  (void);
```

This may be useful when integrating the ZeBu UART transactor to avoid calculation of the right value. To detect the value, the transactor needs to receive at least one data. For more details on how to use this method, see Using the UART Transactor Baud Rate Detector.

The method returns the following values:

- *Zero*: no ratio is detected.

- greater than zero: ratio value detected by the transactor.

getClkCycle() Method

This method returns the clock count of the last hardware to software communication of the UART.

```
uint_t getClkCycle (void);
```

Transactor Logging

setName() Method

This method sets the transactor name shown in the message prefixes.

```
void setName  (const char *name);
```

where, *name* is the pointer to the name string.

getName() Method

This method returns the transactor name shown in the message prefixes.

```
const char* getName  (void);
```

The method returns the following values:

- *Zero*: no name was defined.

- Non-zero values: pointer to the name string.

setDebugLevel() Method

This method sets the maximum information level for messages. The higher the level, the more detailed the messages.

```
void setDebugLevel  (uint lvl);
```

where, *lvl* is the information level. Its values are:

- 0: no messages.

- 1: main steps of the testbench (settings, connections, and so on).

- 2: level 1 messages with internal high-level information about the transactor (status, data, transmission/reception, and so on).

- 3: level 2 messages with low-level information (message contents, callbacks, and so on).

setLog() Method

This method activates and sets parameters for the transactor's log generation.

The log contains transactor's debug and information messages, which can be output into a log file. The log file can be defined with a file descriptor or by a filename.

The log file is closed upon ZeBu UART transactor object destruction.

Log File Assigned through a File Descriptor:

The log file where output messages are assigned through a file descriptor.

```
void setLog  (FILE *stream, bool stdoutDup = false);
```

where:

- *stream* is the output stream (file descriptor).

- *stdoutDup* is the output mode:

  ◦ *true*: messages are output both to the file and the standard output.

  ◦ *false* (default): messages are only output to the file.

Log File defined by a Filename:

The log file where to output messages is defined by its filename.

If the log file you specify already exists, it is overwritten. If it does not exist, the method creates it automatically.

```
bool setLog (char *fname, bool stdoutDup);
```

where:

- *fname* is the log file name.

- *stdoutDup* is the output mode:

  ◦ *true*: messages are output both to the file and the standard output.

  ◦ *false* (default): messages are only output to the file.

The method returns:

- *true* upon success.

- *false* if the specified log file cannot be overwritten or if the method failed in creating the file.

Transactor Dumping

dumpSetRxPrefix() Method

This method sets the "received data" prefix in the dump file. This prefix is added at the beginning of each received data sequence in the output file.

```
void dumpSetRxPrefix  (const char *str);
```

where, *str* is the "received data" prefix. By default, this prefix is "*RX>*".

dumpSetTxPrefix() Method

This method sets the "transmitted data" prefix in the dump file. This prefix is added at the beginning of each sent data sequence in the output file.

```
void dumpSetTxPrefix  (const char *str);
```

where, *str* is the "transmitted data" prefix. By default, this prefix is "*TX>*".

dumpSetDisplayErrors() Method

This method enables dumping of received erroneous data.

```
void dumpSetDisplayErrors  (bool enable);
```

where, *enable* enables or disables dumping:

- *true*: dumping is enabled; an error message is generated in the dump file for all the received data, which is corrupted.

- *false* (default): dumping is disabled.

When enabled, an error message is generated in the dump file for all the received data, which is corrupted.

When disabled, the error is ignored and the erroneous data is discarded.

dumpSetFormat() Method

This method sets the dump file format.

```
void dumpSetFormat  (DumpFormat_t format);
```

where, *format* is of *DumpFormat_t* enumeration type and it values are as follows:

- *DumpRaw*: All data are written out directly in the output file.

- *DumpASCII*: Only data value between 32 and 127 are written out directly. All other values are dumped in following decimal format: \ddd(default setting).

For instance, the following data sequence:

TX	0x54	0x65	0x73	0x74	0x20	0x31	0x32	0x33	0x0a
RX	0x31	0x32	0x33	0x0a					

It results in the following in the dump file in Raw Format (and sequential display):

Figure 4   Dump File in Raw Format
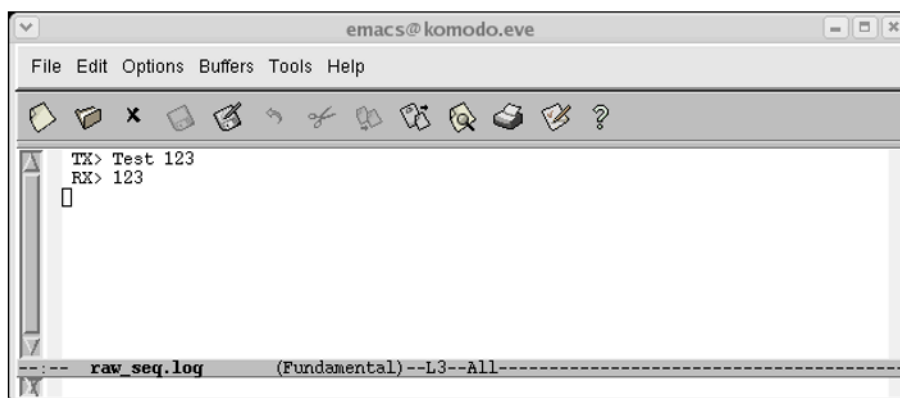
or the following in the dump file in ASCII format (and sequential display):

Figure 5   Dump File in ASCII Format
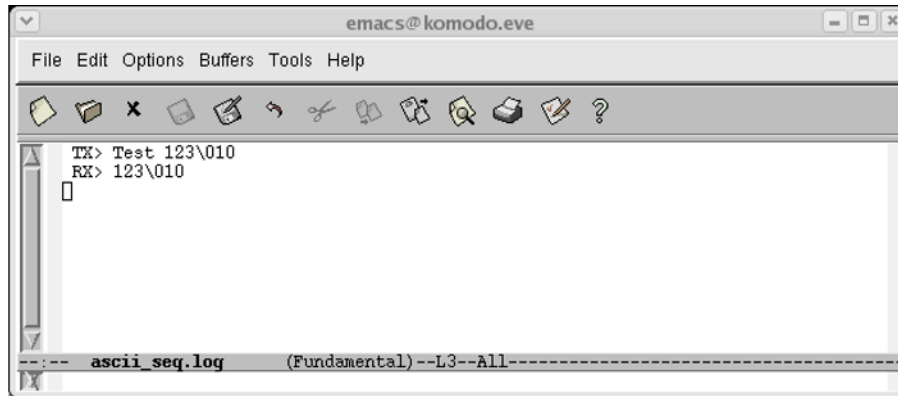
dumpSetDisplay() Method

This method sets the way transmitted and received data are displayed in the dump file.

```
void dumpSetDisplay   (DumpDisplay_t disp);
```
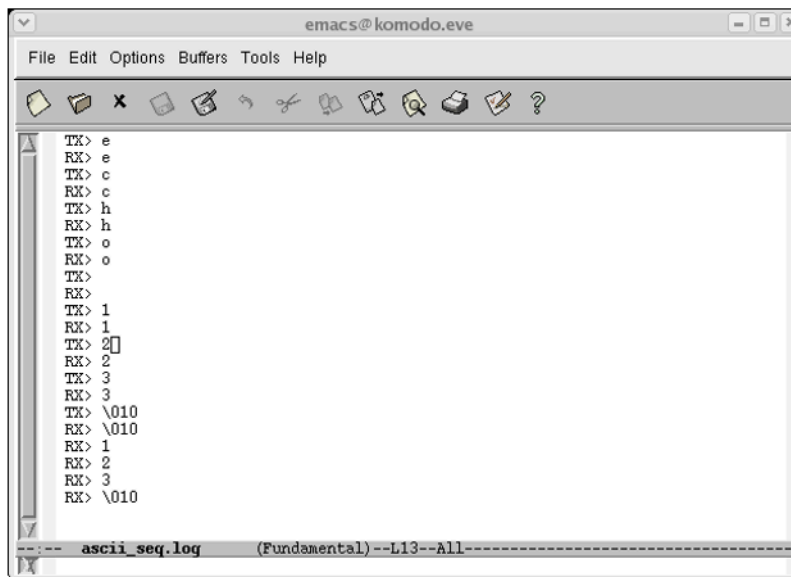
where, *disp* is of *DumpDisplay_t* enumeration type and its values are as follows:

*   *DumpSequential* (default): Transmitted and received data sequences are dumped sequentially. Each received or sent data sequence is dumped in a new line beginning with relevant prefix.

*   *DumpSplit*: Received and transmitted data are displayed side by side (transmitted data on the left, received data on the right).

Example:
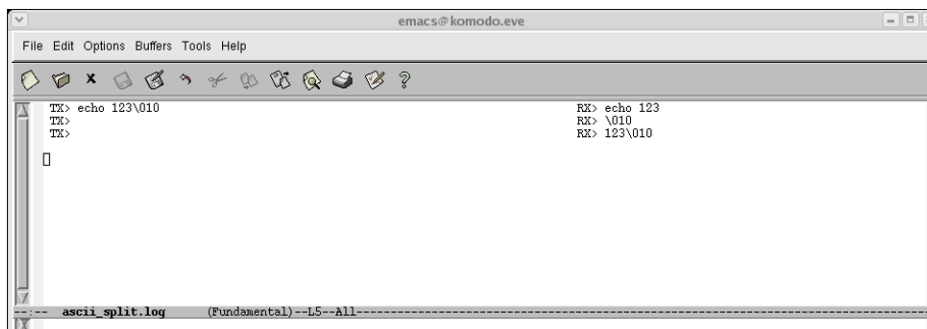
For instance, if the testbench sends a command "*echo 123*" followed by a line feed (*0x0a*), the DUT echoes all the characters and sends the results of the command execution "*123*" followed by a line feed (0x0a).

• Results using the *DumpSequential* setting (and *DumpASCII* format):

• Results using the *DumpSplit* setting (and *DumpASCII* format):

dumpSetMaxLineWidth() Method

This method sets the maximum line width in the dump file.

```
void dumpSetMaxLineWidth  (uint maxwidth);
```

where, *maxwidth* is the maximum line width in number of characters.

If set to *0*, the line width is unlimited.

By default, the maximum line width is *80*.

dumpGetRxPrefix() Method

This method returns the prefix dumped at the beginning of each received data sequence.

```
const char* dumpGetRxPrefix  (void);
```

dumpGetTxPrefix() Method

This method returns the prefix dumped at the beginning of each transmitted data sequence.

```
const char* dumpGetTxPrefix  (void);
```

dumpGetDisplayErrors() Method

This method returns the current error display setting.

```
bool dumpGetDisplayErrors  (void);
```

The method returns *true* when an error message is generated on erroneous data reception. It returns *false* when erroneous received data are ignored.

dumpGetFormat() Method

This method returns the current dump format setting.

```
const_char* dumpGetFormat (void);
```

The returned value can be:

- *DumpASCII*: the current dump format is ASCII.

- *DumpRaw*: the current dump format is Raw.

dumpGetDisplay() Method

This method returns the current dump display configuration.

```
Dconst_char* dumpGetDisplay  (void);
```

The returned value can be:

- *DumpSplit*: the current dump display configuration is the split display.

- *DumpSequential*: the current dump display configuration is the sequential dump display.

dumpGetMaxLineWidth() Method

This method returns the maximum line width in the dump file.

```
uint dumpGetMaxLineWidth  (void);
```

openDumpFile() Method

This method opens the dump file and starts dumping UART traffic.

```
bool openDumpFile  (const char *fname);
```

This method returns *true* when the operation is successful; otherwise, *false*.

closeDumpFile() Method

This method stops dumping UART traffic and closes the dump file.

```
bool closeDumpFile  (void);
```

This method returns *true* when the operation is successful; otherwise, *false*.

dumpSetRxSuffix() Method

This method sets the "received data" suffix in the dump file. This suffix is added at the end of each received data sequence in the output file.

```
void dumpSetRxSuffix (const char *str);
```

where, str is the "received data" suffix. By default, this suffix is "\n".

dumpSetTxSuffix() Method

This method sets the "transmitted data" suffix in the dump file. This suffix is added at the end of each received data sequence in the output file.

```
void dumpSetTxSuffix (const char *str);
```

where, str is the "transmitted data" suffix. By default, this suffix is "\n".

dumpGetRxSuffix() Method

This method returns the suffix dumped at the end of each received data sequence.

```
const char* dumpGetRxSuffix (void);
```

dumpGetTxSuffix() Method

This method returns the suffix dumped at the end of each transmitted data sequence.

```
const char* dumpGetTxSuffix (void);
```

Timestamp Printing

enableTimestamp() Method

This method prints the cycle count and timestamp values when Zcei clockport and RTL clock are used respectively. These values are printed with all debug statements with debug level 2.

For enabling cycle count/timestamp printing, call the following API and enable it in the testbench as shown below:

```
test->enableTimestamp(true);
```

If timestamp is not needed, this API should be disabled as (By default disabled):

```
test->enableTimestamp(false);
```

For enabling timestamp in case of RTL, following UTF option must be passed at compile time:

```
zemi3 -timestamp true
```

dumpSetDataTS() Method

This method is used to enable timestamps for data in data logs with enable being true. By default, this API is disabled.

API enableTimestamp() should be enabled before calling this API otherwise it will show error.

It will print protocol clock count in case of zcei clockport and timestamp in case of RTL CLOCK.

```
void dumpSetDataTS ( bool enable  =  false )
```

where: enable, enables or disables timestamp values in data logs.

"true: dumping of Timestamp is enabled

"false: dumping of Timestamp is disabled

Uart Class Specific Interface

This section provides information about Uart class methods.

The Uart class provides a set of methods to send and receive data over the UART interface.

Table 4   Uart Class Specific Methods

Method	Description
getNewXtor	Static method for transactor constructor. DO NOT use the default constructor.
~xtor_uart_svs	Transactor destructor.
setReceiveCB	Registers the data reception callbacks.

Table 4   Uart Class Specific Methods (Continued)

Method	Description
setSendCB	Registers the data transmission callbacks.
setTimeout	Sets a timeout.
send	Sends the data over a serial link.
sendBreak	Sends a break condition over a serial link.
receive	Returns the received data over a serial link, if any.
txQueueFlush	Sends data remaining in the queue to the transactor hardware.
txQueueLength	Returns the number of data present in the transmission queue.
rxQueueLength	Returns the number of data present in the reception queue.
runClk ()	Advances the clock for specified number of clocks when the transactor is operating in controlled the clock mode.

getNewXtor () and ~xtor_uart_svs Methods

For xtor_uart_svs class constructor, it is recommended to use the unified API-based constructor method. For this use the Xtor::getNewXtor() method as shown below:

```
static Xtor * getNewXtor( ZEBU::Board * board,
const char * xtorTypeName,
XtorScheduler * sched,
const char * driverName = NULL,
svt_c_runtime_cfg* runtime = NULL
) ;
```

Here,

- board helps you handle ZEBU::Board

- xtorTypeName signifies the name that is used to register the class

- sched signifies handle to XtorScheduler

- driverName is the path to the transactor instance in HW. Can be left NULL, API can automatically detect it.

- runtime is the handle to global runtime configuration for testbench.

Example:

The following is the testbench example for constructing the UART transactor object.

```
svt_c_threading* threading = new svt_pthread_threading();
XtorSchedParams->useVcsSimulation   = false;
XtorSchedParams->useZemiManager     = true;
XtorSchedParams->noParams           = true;
XtorSchedParams->zebuInitCb         = NULL;
XtorSchedParams->zebuInitCbParams   = NULL;
xsched->configure(XtorSchedParams) ;
zemi3 = ZEMI3Manager::open(zebuWork,designFeatures);
board = zemi3->getBoard();
zemi3->buildXtorList(); // manually add the xtor or use buildXtorList
zemi3->init();

runtime->set_threading_api(threading);
runtime->set_platform(new svt_zebu_platform(board, false));
svt_c_runtime_cfg::set_default(runtime);

cerr << "#TB : Register UART Transactor..." << board << endl;
xtor_uart_svs::Register("xtor_uart_svs");

test =
static_cast<xtor_uart_svs*>(Xtor::getNewXtor( Board::getBoard(),
"xtor_uart_svs",  xsched, NULL,runtime)) ;
```

```
~xtor_uart_svs
```

The following is the testbench example for destructing the UART transactor object:

setReceiveCB() Method

This method registers the data reception callback. The callback function is called each time a data is received by the UART transactor.

```
void setReceiveCB  (rxFunc_typ rxcb, void *userData = NULL);
```

where:

- *rxcb* is the reception callback.

- *userData* is the pointer to the user data passed to the callback. By default, it is set to NULL.

The reception callback must have the following prototype:

```
void rcvCB (uint8_t data, bool valid, void *userData);
```

where:

- *data* is the received data.

- *valid* is:

    ◦ *true*, if no error is detected during transmission.

    ◦ *False*, if an error is detected during transmission.

- *userData* is the pointer specified when the callback is registered.

The reception callback can be disabled by setting *rxcb* to *NULL*.

setSendCB() Method

This method registers the data transmission callback. The callback function is called each time the UART transactor is ready to send data.

```
void setSendCB  (rxFunc_typ txcb, void *userData = NULL);
```

where:

- *txcb* is the transmission callback.

- *userData* is the pointer to the user data passed to callback. By default, it is set to *NULL*.

The transmission callback must have the following prototype:

```
bool tmitCB (uint char *data, void *userData);
```

where:

- *data* is a pointer to the data to be transmitted.

- *userData* is the pointer specified when registering the callback.

The callback must return *true*, if the data is ready to be sent; otherwise, *false*.

The reception callback can be disabled by setting *txcb* to *NULL*. Note that to service the TxCallback, you must call the runClk () cycle from the testbench regularly.

setTimeout() Method

This method sets the timeout in seconds. When called in blocking mode, and to avoid locking the testbench, the *send()* and *receive()* methods are forced to return when the timeout is reached.

```
void setTimeout  (uint sec);
```

where, *sec* is the timeout value in seconds.

send() Method

This method sends a single data byte over the UART interface in blocking or non-blocking mode in the following way:

- In non-blocking mode (default), it tries to send data and almost immediately returns control to your program (returns false if data not sent).

- In blocking mode, control is not returned to your program until data is sent to the transactor or the timeout is reached (see setTimeout() Method).

```
bool send (uint8_t data, bool blocking = false);
```

where:

- *data* is the data to be sent.

- *blocking* enables or disables the blocking mode:

  - *true*: blocking mode is activated.

  - *false*: non-blocking mode is activated.

    Note:

    If hardware FIFO is full, the data might not be actually sent to the transactor hardware part and it is stored in a transmission queue (see txQueueLength() Method and txQueueLength() Method).

The *send()* method returns *true* when the data is sent or stored in transmission queue; otherwise, *false*.

This method is to be used only in default mode and not in xterm/server mode.

sendBreak() Method

This method sends a break condition (sending continuous 0 values with no Start and no Stop bits) over the UART interface in blocking or non-blocking mode in the following way:

- In non-blocking mode (default), it tries to send the break condition and almost immediately returns control to your program (returns false if the data is not sent).

- In blocking mode, control is not returned to your program until the break condition is sent to the transactor or the timeout is reached (see setTimeout() Method).

```
bool sendBreak  (bool blocking = false);
```

- **Note:**

  IIf hardware FIFO is full, the break condition might not be sent to the transactor hardware part and it is stored in a transmission queue (see txQueueFlush() Method and "txQueueLength() Method").

The *sendBreak()* method returns *true* when the break condition is sent or stored in transmission queue; otherwise, *false*.

receive() Method

This method is to be used in default mode only. It returns a single data received from the UART interface in blocking or non-blocking mode in the following way:

- In non-blocking mode (default), it returns the data received over the serial link, if any, and immediately returns control to the program.

- In blocking mode, control is not returned to your program until received data is available or the timeout is reached (see setTimeout() Method).

```
int receive (uint8_t *data, bool blocking = false);
```

where:

- *data* is the data to be received.

- *blocking* enables or disables the blocking mode:

  - *true*: blocking mode is activated.

  - *false*: non-blocking mode is activated.

This method returns:

- *0*: no data is received.

- *1*: data is received.

- *-1*: data is received, but a parity error is detected.

txQueueFlush() Method

This method sends remaining data from transmission queue to the hardware part of the transactor.

```
bool txQueueFlush  (void);
```

The method returns:

- *true* when the transmission queue is empty.

- *false* when data is remaining in the transmission queue.

txQueueLength() Method

This method returns the data length present in the transmission queue. If a data cannot be sent immediately in non-blocking mode or if the timeout is elapsed in blocking mode, the *send()* method stores the data in a transmission queue for later transmission.

The data stored in the transmission queue is consumed, the runClk() cycle or next send() non-blocking command, is called, by the *txQueueFlush()* method or each time the *send()* method is called. This method can then be used to ensure that all the data is sent.

```
uint txQueueLength  (void);
```

rxQueueLength() Method

This method returns the data length present in the reception queue.

The data stored in the reception queue can be consumed using the *receive()* method. Use this method to check if all the received data is consumed.

```
uint rxQueueLength  (void);
```

runClk () Method

This method advances the clock for specified number of clocks when the transactor is operating in controlled the clock mode. Also this method services the Tx Callbacks registered. Therefore, you must call this method if the testbench is registering a TX callback using setSendCB ().

The following is the syntax of the runClk() Method

```
void runClk  (uint32_t numClks) ;
```

Reset check APIs

Use the runUntilReset() API in the testbench before configuration to wait for the reset assertion.

However, for a setup with multiple transactors or threaded environment, this can cause halt as it is a blocking API. In such cases, use the non-blocking counterpart, that is, isResetActive(). This returns the reset status hence it needs to be called in a while loop.

Register/unregister tx and rx callback functions

Following APIs are used to register and unregister tx callback function:

- test->registerTxTxnCb(dataCbFnTx, test);

- test->unRegisterTxTxnCb(dataCbFnTx);

Following APIs are used to register and unregister rx callback function:

- test->registerRxTxnCb(dataCbFnRx, test);

- test->unRegisterRxTxnCb(dataCbFnRx);

Server Mode Specific Interface

This section provides information about the class methods applicable when the operating mode is set as ServerMode.

The UART server uses the UART transactor and the TCP/IP protocol to relay data from the UART transactor to the TCP socket. It allows the user to send and receive data from a remote PC, running on the Linux or the Windows operating system.

Table 5   Server Mode Specific Methods

Method	Description
startServer	Starts the TCP server.
startClient	Connects to the remote TCP server.
isConnected	Returns the TCP connection status.
closeConnection	Closes the TCP connection.
setDisplayErrors	Enables the generation of an error message on erroneous data reception.
getDisplayErrors	Obtains the error display setting.
setInputCharCB	Defines the conversion callback function for input characters.
setOutputCharCB	Defines the conversion callback function for the output characters
printTermString	Converts, formats, and prints its arguments to the file descriptor.
setApproxThreshold	set approximate threshold value to get Ratio in software side when Baud Rate changing from lower to higher value
enableMassiveSocket() Method	enable massive socket operation, allowing multiple socket connections > 1000 by using poll().

Description

startServer() Method

This method opens a TCP connection in server mode. The method starts the server and returns immediately without waiting for an incoming connection.

```
bool startServer (uint16_t tcpPort, uint frameSize = 1534);
```

where:

- *tcpPort* is the TCP port number. It ranges from *0* through *65553*, generally 1024 through 65553 are used.

- *frameSize* is the maximum size of TCP frames. Default value is *1534* (optional parameter).

The method returns *true* when the server was launched successfully; otherwise, *false*.

startClient() Method

This method opens a TCP connection in the client mode to an existing server. The method tries to establish a connection with the server and returns immediately.

```
bool startClient (const char *host,
                  uint16_t tcpPort,
                  uint frameSize = 1534 );
```

where,

- *host* is the host name.

- *tcpPort* is the TCP port number. It can range from *0* through *65553*, generally *1024* through *65553* are used.

- *frameSize* is the maximum size of TCP frames. Default value is *1534* (optional parameter).

The method returns *true* when the connection is successful; otherwise, *false*.

isConnected() Method

This method internally handles the connection along with returning the connection status.

```
bool isConnected (void);
```

The method returns *true* if the TCP is connected, *false* if it is disconnected.

For multiple connections and disconnections, this method is called in a while loop until connect/disconnect is performed, as shown below:

```
while(TBXtor::stop_tb == 0){
  if(xtor->isConnected() == 1)
     {
      xtor->runClk(1000);
    }
}
```

If the connection is pending, it internally re-establishes the connection.

closeConnection() Method

This method closes the TCP connection in client or server mode.

```
bool closeConnection  (void);
```

The method returns *true* when the TCP connection is closed, *false* if an error occurs while closing connection.

setDisplayErrors() Method

This method enables the generation of an error message when erroneous data is received.

```
void setDisplayErrors (bool enable);
```

where, *enable* activates or deactivates the error message generation in the following way:

- *true*: an error message is generated in the terminal.

- *false* (default): the error is ignored and erroneous data is discarded.

getDisplayErrors() Method

This method returns the error display setting.

```
bool getDisplayErrors  (void);
```

It returns:

- *true*: Transmission errors are monitored and displayed.

- *false*: Data with transmission errors is not displayed.

setInputCharCB() Method

This method defines the callback function that translates all input characters from the server/client before sending them to the DUT interface.

```
void setInputCharCB (ConvFunct_t cb, void *user_data);
```

where:

- cb is the pointer to the callback translation function. The function type must be bool fconv (char ichar, char &ochar, void *user_data). Here, ochar is same as ichar.

- user_data is the user-defined data.

setOutputCharCB() Method

This method defines the callback function that translates all output characters from the DUT before sending them to the server/client.

```
void setOutputCharCB (ConvFunct_t cb, void *user_data);
```

where:

- cb is the pointer to the callback translation function. The function type must be bool fconv (char ichar, char &ochar, void *user_data). Where, ochar is same as ichar.

- user_data is the user-defined data.

printTermString() Method

This method converts, formats and prints its arguments to file descriptor.

```
void printTermString (const char *format, ... );
```

The *printTermString()* method works like the standard *printf*.

setApproxThreshold Method

This method sets the approximate threshold value when ratio changes from lower to higher value. If the Baud rate value changed from lower to higher value, (then depending upon Threshold value) hardware side takes some clock cycles to update Ratio in API getDetectedRatio, so some packets may be corrupted during that time.

The config() method must be called to make the setting effective in the hardware. By default, value is 0xFFFFFF which is maximum value.

```
virtual bool setApproxThreshold ( unsigned int threshold );
```

where:

- threshold is the user-defined data

enableMassiveSocket() Method

This method is used to enable massive socket operation. It allows multiple socket connections > 1000 by using poll(). By default the UART uses select(), which has a limitation of 1000 connections.

test->enableMassiveSocket(<flag>);

where:

? <flag> = true enables massive socket operation.

XtermMode Specific Interface

This section provides information about the *UartTerm* class methods.

The UART terminal provides a graphical terminal (*xterm*).

Table 6   UartTerm Class Specific Methods

Method	Description
setConvMode	Sets the conversion mode for xterm
setTermName	Sets the terminal name.
getTermName	Obtains the terminal name.
isAlive	Obtains the terminal status.
printTermString	Converts, formats, and prints its arguments to the xterm terminal
setDisplayErrors	Enables the generation of an error message on erroneous data reception
getDisplayErrors	Obtains the error display setting
setInputCharCB	Defines the conversion callback function for the input characters
setOutputCharCB	Defines the conversion callback function for the output characters
setFilterIn	Enables or disables filtering on input characters
setFilterOut	Enables or disables filtering on output characters
isFilterInEnabled	Obtains the filtering status on input characters
isFilterOutEnabled	Obtains the filtering status on output characters
enXtermEcho	Enables echo of data transmitted from transactor to DUT side on xterm terminal.
xtermClearAfterRst	clear xterm terminal after the reset is asserted.
setXtermGeometry()	Sets the geometry of Xterm window i.e. size and position of Xterm window.

Description

setConvMode() Method

This method accepts a parameter that is used to apply some character translation to special OS-dependent characters such as CR, NL, LF, and so on.

```
setConvMode (ConvMode_t char_conversion_mode);
```

Where, *char_conversion_mode* defines the type of predefined conversion algorithm already defined in the transactor:

- *Conv_None*: No conversion.
- *Conv_DOS*: DOS conversion.
- *Conv_DOS_BSR*: DOS conversion with BSR (*0x13*) only for *CRLF*.
- *Conv_ISO*: ISO conversion.
- *Conv_MAC*: MAC character conversion.
- Conv_7bits: Conversion to 7-bit ASCII code.

setTermName() Method

This method sets the terminal name (*xterm* window title).

```
bool setTermName  (const char *name);
```

where, *name* is the pointer to the terminal name string.

The method returns *true* when the operation is successful; otherwise, *false*.

Note:

Call this API before setOpMode() to enable setting of terminal name in case Xterm Mode is set.

getTermName() Method

This method returns the terminal name (*xterm* window title).

```
const char *name getTermName  (void);
```

isAlive() Method

This method returns the terminal status.

```
bool isAlive  (void);
```

The method returns:

- *true* when the terminal is active.
- *false* when the terminal is inactive.

The terminal can be closed by terminating the *xterm* window.

printTermString() Method

This method converts, formats, and prints its arguments to the xterm terminal.

```
void printTermString (const char *format, ... );
```

The *printTermString()* method works like the standard *printf*.

setDisplayErrors() Method

This method enables the generation of an error message when an erroneous data is received.

```
void setDisplayErrors (bool enable);
```

where, enable activates or deactivates the error message generation in the following way:

- *true*: an error message is generated in the terminal.

- *false* (default): the error is ignored and erroneous data is discarded.

getDisplayErrors() Method

This method returns the error display setting.

```
bool getDisplayErrors  (void);
```

It returns:

- *true*: Transmission errors are monitored and displayed.

- *false*: Data with transmission errors is not displayed.

setInputCharCB() Method

This method defines the callback function that translates all input characters from the terminal before sending them to the DUT interface.

ConvMode_t must be set to *Conv_None* in the *UartTerm* constructor.

```
void setInputCharCB (ConvFunct_t cb, void *user_data);
```

where:

- *cb* is the pointer to the callback translation function. The function type must be *bool fconv* (*char ichar, char &ochar, void *user_data*). where, *ochar* is the result of *ichar* conversion.

- *user_data* is the user-defined data.

setOutputCharCB() Method

This method defines the callback function that translates all output characters from the DUT before sending them to the terminal.

ConvMode_t must be set to *Conv_None* in the *UartTerm* constructor.

```
void setOutputCharCB (ConvFunct_t cb, void *user_data);
```

where:

- *cb* is the pointer to the callback translation function. The function type must be *bool fconv*(*char ichar, char &ochar, void *user_data*). Where, *ochar* is the result of *ichar* conversion.

- *user_data* is the user-defined data.

setFilterIn() Method

This method enables or disables the filtering on the input characters.

```
void setFilterIn (bool enable);
```

where enable activates or deactivates the filtering in the following way:

- *true*: filtering is enabled.

- *false*: filtering is disabled.

setFilterOut() Method

This method enables or disables filtering on output characters.

```
void setFilterOut (bool enable);
```

where enable activates or deactivates the filtering in following way:

- *true*: filtering is enabled.

- *false*: filtering is disabled.

isFilterInEnabled() Method

This method gets the filtering status on input characters.

```
bool isFilterInEnabled ();
```

The method returns:

- *true*: filtering is enabled.

- *false*: filtering is disabled.

isFilterOutEnabled() Method

This method gets the filtering status on the output characters.

```
bool isFilterOutEnabled ();
```

The method returns:

- *true*: filtering is enabled.

- *false*: filtering is disabled.

enXtermEcho() Method

This method is used to enable echo of data transmitted from XTOR to DUT side on xterm terminal. Data from DUT to xtor side is by default visible on xterm terminal. By default, this API is disable.

```
void enXtermEcho ( bool enable_xterm_echo = false)
```

where, enable_xterm_echo activates or deactivates the echo of data transmitted from transactor to DUT side.

- true: filtering is enabled

- false: filtering is disabled

xtermClearAfterRst()

This method is used to clear the contents of the xterm terminal after the reset is asserted. By default, this API is disabled.

```
void xtermClearAfterRst (bool xterm_clear = false)
```

where, xterm_clear enables or disables the clearing of the xterm terminal after the reset is asserted.

- True: enables the clearing of the xterm terminal

- False: disables the clearing of the xterm terminal

setXtermGeometry()

This method is used to set the geometry of Xterm window i.e. size and position of Xterm window. This method should always be called before setOpMode only.

```
void setXtermGeometry (int xtermWidth, int xtermHeight, int xtermXoffset,
 int xtermYoffset)
```

where:

- xtermWidth: sets the width of xterm window in characters.

- xtermHeight: sets the height of xterm window in lines.

- xtermXoffset: sets the horizontal offset of xterm window, that is, the number of pixels from the left edge of the screen.

- xtermYoffset: sets the vertical offset of xterm window, that is, the number of pixels from the top edge of the screen.

Key Combinations Supported by xterm Terminal

The xterm runs a tool that connects to the UART transactor and automatically exchanges data between the xterm terminal and the UART transactor.

The following combinations are not filtered out by the terminal and are transmitted to the DUT:

- *Ctrl-[A-Z]*

- *Ctrl-*

Initializing and Configuring the UART Transactor

The following figure illustrates the initialization of UART transactor:

Figure 6   Transactor Initialization
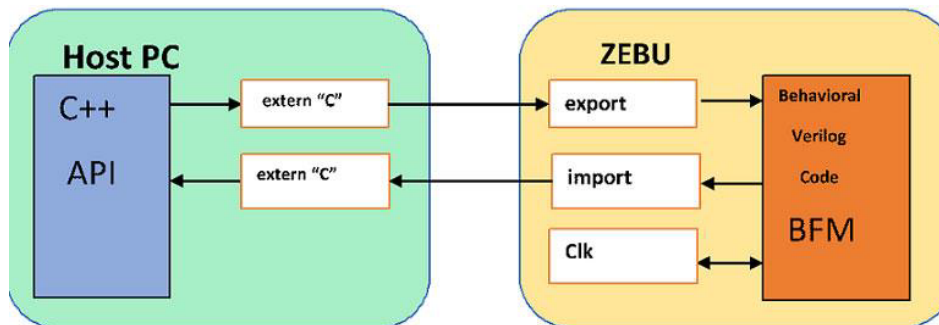
The following are the steps to initialize the transactor:

1. Include necessary UART header files along with other ZeBu/Zemi3 header files

#include "xtor_uart_svs.hh"

…..

using namespace *XTOR_UART_SVS*;

2. Declare and initialize UART driver handles (*xtor_uart_svs*) in the main method:

*int main (int argc, char * argv[]) {*

return run_test ("uart_server_test") ;

}

*int run_test (const char *testName){*

xtor_uart_svs test = NULL;*

xtor_uart_svs replier = NULL;*

xtor_uart_svs::Register("xtor_uart_svs");

*ZEMI3Manager *zemi3 = NULL ;*

zemi3 = ZEMI3Manager::open(zebuWork,designFeatures);

zemi3->init();

test = static_cast<xtor_uart_svs>(Xtor::getNewXtor(Board::getBoard(), "xtor_uart_svs" , xsched, NULL,runtime));*

replier = static_cast<xtor_uart_svs>(Xtor::getNewXtor(Board::getBoard(),"xtor_uart_svs" , xsched, NULL,runtime)) ;*

3. Start zemi3

zemi3->start();

4. Reset

cerr <<"--------------------------Wait for Reset---------------------------" << endl;

#ifdef NON_BLOCKING

while(test->isResetActive());

#else

test->runUntilReset();

#endif;

5. Set transactor parameters to configure the transactor:

ok = test->setWidth(8);

ok &= test->setParity(NoParity);

ok &= test->setStopBit(TwoStopBit);

ok &= test->setRatio(16);

ok &= test->config();

```
if(!ok) { throw ("Could not configure UART"); }

ok = replier->setWidth(8);

ok &= replier->setParity(NoParity);

ok &= replier->setStopBit(TwoStopBit);

ok &= replier->setRatio(16);

ok &= replier->config();

if(!ok) { throw ("Could not configure UART"); }
```

4

Example

This example is available in the *XTOR/xtor_uart_svs.<version>/example* directory and the testbench source code for each is available in the example/src/bench subdirectory.

README is available in path example/zebu.

Typical Implementation

This chapter provides typical testbench implementation using three UART modes.

This section describes the following sub-topics.

- Using the xtor_uart_svs class for default mode

- Using the Server Mode

- Using the XtermMode

- Using the zRci Flow

Using the xtor_uart_svs class for default mode

This section provides the following three typical testbench implementations using the *Uart* class:

- Testbench Using the Uart Class With Non-Blocking Send and Receive

- Testbench Using the Uart Class With Blocking Send and Receive

- Testbench Using the Uart Class With Callbacks

Testbench Using the Uart Class With Non-Blocking Send and Receive

```
#include <stdexcept>
#include <exception>
#include <queue>
#include <libZebu.hh>
#include "xtor_uart_svs.hh"
```

```
#include "svt_report.hh"
#include "svt_pthread_threading.hh"
#include "svt_cr_threading.hh"
#include "svt_c_threading.hh"
#include "svt_zebu_platform.hh"
#include "TopScheduler.hh"
#include "XtorScheduler.hh"

#include "libZebuZEMI3.hh"
#include "svt_systemverilog_threading.hh"
#include "svt_simulator_platform.hh"
#define THREADING svt_systemverilog_threading

using namespace ZEBU;
using namespace ZEBU_IP;
using namespace XTOR_UART_SVS;
using namespace std;


uint8_t convertData ( uint8_t data )
{
    uint8_t ret = data;
    if (ret > 'a' && ret < 'Z') {
        ret = (ret < 'z')?(ret+0x20):(ret-0x20);
    }
    return ret;
}
//################################
//  main
//################################
int main ( ) {
    int ret = 0;
    xtor_uart_svs  *test    = NULL;
    xtor_uart_svs  *replier = NULL;
    Board *board  = NULL;
    ZEMI3Manager *zemi3 = NULL ;
    bool ok;
    TbCtxt context;

    uint8_t data         = 0;
    bool    dataSent     = false;
    uint8_t tmpData      = 0;
    queue<uint8_t> dQueue;
    bool    end          = false;
    unsigned int   errors        = 0;
    try {
        //open ZeBu
        XtorScheduler     * xsched = XtorScheduler::get();
        XtorSchedParams_t * XtorSchedParams = xsched->getDefaultParams();
        svt_c_runtime_cfg * runtime   = new svt_c_runtime_cfg();
        char *zebuWork        = ZEBUWORK;
        char *designFeatures = DFFILE;
        svt_c_threading *threading = new svt_pthread_threading();
```

```
        XtorSchedParams->useVcsSimulation   = false;
        XtorSchedParams->useZemiManager     = true;
        XtorSchedParams->noParams           = true;
        XtorSchedParams->zebuInitCb         = NULL;
        XtorSchedParams->zebuInitCbParams   = NULL;
        xsched->configure(XtorSchedParams) ;
        zemi3 = ZEMI3Manager::open(zebuWork,designFeatures);
        board = zemi3->getBoard();
        zemi3->buildXtorList(); // manually add the xtor or use
buildXtorList
        zemi3->init();

        runtime->set_threading_api(threading);
        runtime->set_platform(new svt_zebu_platform(board, false));
        svt_c_runtime_cfg::set_default(runtime);


        cerr << "#TB : Register UART Transactor..." << board << endl;
        xtor_uart_svs::Register("xtor_uart_svs");

        test =
static_cast<xtor_uart_svs*>(Xtor::getNewXtor( Board::getBoard(),
"xtor_uart_svs",  xsched, NULL,runtime)) ;
        fprintf (stderr, "Found - Drivers [%s][%s] ...\n",
test->getDriverModelName (),test->getDriverInstanceName ());
        replier =
static_cast<xtor_uart_svs*>(Xtor::getNewXtor( Board::getBoard(),
"xtor_uart_svs",  xsched, NULL,runtime)) ;
        fprintf (stderr, "Found - Drivers [%s][%s] ...\n",
replier->getDriverModelName (),replier->getDriverInstanceName ());

        //open ZeBu
        test->init(board,"uart_device_wrapper.uart_driver_0");
        replier->init(board,"uart_device_wrapper.uart_driver_1");

        cerr <<"------------------------Zemi3
Start---------------------------" << endl;
        zemi3->start();   //This will start the Zemi3 service loop


        cerr <<"------------------------Wait for Reset
--------------------------" << endl;
        test->runUntilReset() ;
        ok = test->setWidth(8);
        ok &= test->setParity(NoParity);
        ok &= test->setStopBit(TwoStopBit);
        ok &= test->setRatio(16);
        ok &= test->config();
        if (!ok ){ throw ("UART config failed");}

        ok = replier->setWidth(8);
        ok &= replier->setParity(NoParity);
```

```
            ok &= replier->setStopBit(TwoStopBit);
            ok &= replier->setRatio(16);
            ok &= replier->config();
            if (!ok ){ throw ("UART config failed");}

            test->dumpSetDisplayErrors(true);
            test->dumpSetFormat(DumpASCII);
            test->dumpSetDisplay(DumpSplit);
            test->dumpSetMaxLineWidth(5);
            test->openDumpFile("test_dump.log");

            replier->dumpSetDisplayErrors(true);
            replier->dumpSetFormat(DumpASCII);
            replier->dumpSetDisplay(DumpSplit);
            replier->dumpSetMaxLineWidth(5);
            replier->openDumpFile("test_dump.log");

            context.TX = test;
            context.RX = replier;

            // TB Main loop
            while (!end) {

                // Tester data sending
                if (!dataSent) {
                    if (test->send(data)) {
                        dataSent = true;
                        printf("Tester  -- Sending data 0x%02x\n",data);
                    }
                }

                // Tester data receiving
                if (test->receive(&tmpData)) {
                    if (!dataSent) {
                        printf("Tester  -- Received unexpected data
0x%02x !!!\n",tmpData); ++errors;
                    } else {
                        printf("Tester  -- Received data 0x%02x\n",tmpData);
                        dataSent = false;
                        if (tmpData != convertData(data)) {
                            printf("Wrong data '0x%02x' instead of
'0x%02x'\n",
                                    tmpData, convertData(data));
                            ++errors;
                        }

                        if (data == 0xff) {
                            end = true;
                            test->sendBreak(true);
                        } else {
                            ++data;
                        }
                    }
```

```
            }
            // Replier data receiving
            if (replier->receive(&tmpData)) {
                printf("Replier -- Received data 0x%02x\n",tmpData);
                // push processed received data in tx queue
                dQueue.push(convertData(data));
            }

            // Replier data sending
            // If TX queue not empty, try to send next data
            if (!dQueue.empty()) {
                if (replier->send(dQueue.front())) {
                    printf("Replier -- Sending data
0x%02x\n",dQueue.front());
                    // data sent, remove it from tx queue
                    dQueue.pop();
                } else {
                    printf("REPLIER - Could not send data\n");
fflush(stdout);
                }
            }
        }

        test->closeDumpFile();
        replier->closeDumpFile();

        if (errors != 0) {
            printf("Detected %d errors during test\n",errors);
            ret = 1;
        }
    } catch (const char *err) {
        ret = 1; fprintf(stderr,"Testbench error: %s\n", err);
    }
    if (test != NULL)  { delete test; }
    if (replier != NULL) { delete replier; }

    zemi3->terminate();
    zemi3->close();

    printf("Test %s\n",(ret==0)?"OK":"KO");

    return ret;
}
```

Testbench Using the Uart Class With Blocking Send and Receive

```
#include <stdexcept>
#include <exception>
#include <queue>
#include <libZebu.hh>
#include "xtor_uart_svs.hh"
#include "svt_report.hh"
#include "svt_pthread_threading.hh"
```

```
#include "svt_cr_threading.hh"
#include "svt_c_threading.hh"
#include "svt_zebu_platform.hh"
#include "TopScheduler.hh"
#include "XtorScheduler.hh"

#include "libZebuZEMI3.hh"
#include "svt_systemverilog_threading.hh"
#include "svt_simulator_platform.hh"
#define THREADING svt_systemverilog_threading
using namespace ZEBU;
using namespace ZEBU_IP;
using namespace XTOR_UART_SVS;
using namespace std;

typedef struct {
  xtor_uart_svs* rep;
  queue<uint8_t> dQueue;
} UsrCtxt_t;
uint8_t convertData ( uint8_t data )
{
  uint8_t ret = data;
  if (ret > 'a' && ret < 'Z') {
    ret = (ret < 'z')?(ret+0x20):(ret-0x20);
  }
  return ret;
}
void usercb ( void * context )
{
  UsrCtxt_t* ctxt = (UsrCtxt_t*)context;
  uint8_t data;


  // Check if data received on uart
  if (ctxt->rep->receive(&data)) {
    printf("Replier -- Received data 0x%02x\n",data);
    // push processed received data in tx queue
    ctxt->dQueue.push(convertData(data));
  }

  // If TX queue not empty, try to send next data
  if (!(ctxt->dQueue.empty())) {
    if (ctxt->rep->send(ctxt->dQueue.front())) {
      printf("Replier -- Sending data 0x%02x\n",ctxt->dQueue.front());
      // data sent, remove it from tx queue
      ctxt->dQueue.pop();
    }
  }
}
//##############################
//  main
//##############################
int main () {
```

```
int ret = 0 ;
xtor_uart_svs  *test    = NULL;
xtor_uart_svs  *replier = NULL;
Board *board   = NULL;
ZEMI3Manager *zemi3 = NULL ;
bool ok;

UsrCtxt_t usrCtxt;
uint8_t data         = 0;
uint8_t tmpData      = 0;
bool    end          = false;
unsigned int    errors       = 0;

try {
  //open ZeBu
  XtorScheduler     * xsched = XtorScheduler::get();
  XtorSchedParams_t * XtorSchedParams = xsched->getDefaultParams();
  svt_c_runtime_cfg * runtime  = new svt_c_runtime_cfg();

  char *zebuWork     = ZEBUWORK;
  char *designFeatures = DFFILE;
  svt_c_threading* threading = new svt_pthread_threading();

  XtorSchedParams->useVcsSimulation   = false;
  XtorSchedParams->useZemiManager     = true;
  XtorSchedParams->noParams           = true;
  XtorSchedParams->zebuInitCb         = NULL;
  XtorSchedParams->zebuInitCbParams   = NULL;
  xsched->configure(XtorSchedParams) ;
  zemi3 = ZEMI3Manager::open(zebuWork,designFeatures);
  board = zemi3->getBoard();
  zemi3->buildXtorList(); // manually add the xtor or use buildXtorList
  zemi3->init();

  runtime->set_threading_api(threading);
  runtime->set_platform(new svt_zebu_platform(board, false));
  svt_c_runtime_cfg::set_default(runtime);

  cerr << "#TB : Register UART Transactor..." << board << endl;
  xtor_uart_svs::Register("xtor_uart_svs");

  test =
static_cast<xtor_uart_svs*>(Xtor::getNewXtor( Board::getBoard(),
"xtor_uart_svs",  xsched, NULL,runtime)) ;
  fprintf (stderr, "Found - Drivers [%s][%s] ...\n",
test->getDriverModelName (),test->getDriverInstanceName ());
  replier =
static_cast<xtor_uart_svs*>(Xtor::getNewXtor( Board::getBoard(),
"xtor_uart_svs",  xsched, NULL,runtime)) ;
  fprintf (stderr, "Found - Drivers [%s][%s] ...\n",
replier->getDriverModelName (),replier->getDriverInstanceName ());
  test->init(board,"uart_device_wrapper.uart_driver_0");
  replier->init(board,"uart_device_wrapper.uart_driver_1");
```

```
    cerr <<"--------------------------Zemi3
Start---------------------------" << endl;
    zemi3->start();   //This will start the Zemi3 service loop


    test->setLog ("tester.log") ;
    test->setDebugLevel(4) ;

    cerr <<"-------------------------Wait for Reset
---------------------------" << endl;
    test->runUntilReset() ;

    ok = test->setWidth(8);
    ok &= test->setParity(NoParity);
    ok &= test->setStopBit(TwoStopBit);
    ok &= test->setRatio(16);
    ok &= test->config();
    if (!ok ){ throw ("UART config failed");}

    replier->setLog ("replier.log") ;
    replier->setDebugLevel(4) ;
    ok = replier->setWidth(8);
    ok &= replier->setParity(NoParity);
    ok &= replier->setStopBit(TwoStopBit);
    ok &= replier->setRatio(16);
    ok &= replier->config();
    if (!ok ){ throw ("UART config failed");}

    test->dumpSetDisplayErrors(true);
    test->dumpSetFormat(DumpASCII);
    test->dumpSetDisplay(DumpSplit);
    test->dumpSetMaxLineWidth(40);
    test->openDumpFile("test_dump.log");

    replier->dumpSetDisplayErrors(true);
    replier->dumpSetFormat(DumpASCII);
    replier->dumpSetDisplay(DumpSplit);
    replier->dumpSetMaxLineWidth(40);
    replier->openDumpFile("test_dump.log");

    usrCtxt.rep = replier;
    //test->registerUserCB(usercb, &usrCtxt);

    // TB Main loop
    while (!end) {
      // Tester sends data
      if (!test->send(data,true)) {
        throw("Tester could not send data");
      } else {
        printf("Tester  -- Sending data 0x%02x\n",data);
      }
```

```
      uint8_t rdata ;
      if (replier -> receive(&rdata, true)) {
         printf("Replier -- Received data 0x%02x\n",rdata);
          // push processed received data in tx queue
          usrCtxt.dQueue.push(convertData(rdata));
      }
      // If TX queue not empty, try to send next data
      if (!(usrCtxt.dQueue.empty())) {
        if (replier->send(usrCtxt.dQueue.front())) {
          printf("Replier -- Sending data
0x%02x\n",usrCtxt.dQueue.front());
          // data sent, remove it from tx queue
          usrCtxt.dQueue.pop();
        }
      }

      // Tester receives and check received data
      if (test->receive(&tmpData,true)) {
        printf("Tester  -- Received data 0x%02x\n",tmpData);
        if (tmpData != convertData(data)) {
          printf("Wrong data '0x%02x' instead of '0x%02x'\n",
                 tmpData, convertData(data));
          ++errors;
        }
      } else { throw("Tester could not receive data"); }
      if (data == 0xff) {
        end = true;
      } else {
        ++data;
      }
    }

    test->closeDumpFile();
    replier->closeDumpFile();

    if (errors != 0) {
      printf("Detected %d errors during test\n",errors);
      ret = 1;
    }
  } catch (const char *err) {
    ret = 1; fprintf(stderr,"Testbench error: %s\n", err);
  }
  if (test != NULL)  { delete test; }
  if (replier != NULL) { delete replier; }

  zemi3->terminate();
  zemi3->close();

  return 0;
}
```

Testbench Using the Uart Class With Callbacks

```cpp
#include <stdexcept>
#include <exception>
#include <queue>
#include <libZebu.hh>
#include <sys/resource.h>
#include <sys/time.h>
#include "xtor_uart_svs.hh"
#include "svt_report.hh"
#include "svt_pthread_threading.hh"
#include "svt_cr_threading.hh"
#include "svt_c_threading.hh"
#include "svt_zebu_platform.hh"
#include "TopScheduler.hh"
#include "XtorScheduler.hh"

#include "libZebuZEMI3.hh"
#include "svt_systemverilog_threading.hh"
#include "svt_simulator_platform.hh"
#define THREADING svt_systemverilog_threading
svt_c_threading * threading = NULL ;

using namespace ZEBU;
using namespace ZEBU_IP;
using namespace XTOR_UART_SVS;
using namespace std;

uint8_t convertData ( uint8_t data )
{
  uint8_t ret = data;
  if (ret > 'a' && ret < 'Z') {
    ret = (ret < 'z')?(ret+0x20):(ret-0x20);
  }
  return ret;
}

typedef struct {
  uint8_t sentData;
  bool    dataExpected;
  unsigned int    error;
  bool    end;
} TbCtxt;

// TX Callback
bool txCB ( uint8_t* data, void* ctxt )
{
  bool send = false;
  TbCtxt* tbCtxt = (TbCtxt*)ctxt;
  if (!tbCtxt->dataExpected) {
    if (tbCtxt->sentData == 0xFF) {
      tbCtxt->end = true; // All data have been sent and received
```

```
      //if (threading -> is_blocked()) {
      //   threading -> unblock() ;
      //}
    } else {
      *data = ++(tbCtxt->sentData);
      tbCtxt->dataExpected = true;
      send = true; // send next data
      printf("Tester  -- Sending data 0x%02x\n",*data);
    }
  }
  return send;
}


// RX Callback
void rxCB ( uint8_t data, bool valid, void* ctxt )
{
  TbCtxt* tbCtxt = (TbCtxt*)ctxt;

  printf("Tester  -- Received data 0x%02x %s\n",data,valid?"":"-- parity
 error detected !!! )");

  if (!tbCtxt->dataExpected) {
    printf("Received unexpected data '%02x'\n", data);
    ++(tbCtxt->error);
  } else {
    tbCtxt->dataExpected = false;
    if (!valid) {
      printf("Parity error detected on received data '%02x'\n", data);
      ++(tbCtxt->error);
    } else if (data != convertData(tbCtxt->sentData)) {
      printf("Wrong data '0x%02x' instead of '0x%02x'\n",
             data, convertData(tbCtxt->sentData));
      ++(tbCtxt->error);
    }
  }
}


typedef struct {
  queue<uint8_t> dataQueue;
} RpCtxt;

// TX Callback
bool rpTxCB ( uint8_t* data, void* ctxt )
{
  bool send = false;
  RpCtxt* rpCtxt = (RpCtxt*)ctxt;

  if (!((rpCtxt->dataQueue).empty())) {
    *data =  convertData((rpCtxt->dataQueue).front());
    (rpCtxt->dataQueue).pop();
    send = true; // send next data
    printf("Replier -- Sending data 0x%02x\n",*data);
```

```
    }
    return send;
  }


// RX Callback
void rpRxCB ( uint8_t data, bool valid, void* ctxt )
{
  RpCtxt* rpCtxt = (RpCtxt*)ctxt;
  printf("Replier -- Received data 0x%02x %s\n",data,valid?"":"-- parity
 error detected !!! )");
  if (valid) {
    (rpCtxt->dataQueue).push(data);
  } else {
    (rpCtxt->dataQueue).push(0x0);
  }
}

//################################
//  main
//################################
int main () {
  int ret = 0;
  xtor_uart_svs  *test    = NULL;
  xtor_uart_svs  *replier = NULL;
  Board *board   = NULL;
  ZEMI3Manager *zemi3 = NULL ;
  bool ok;
  TbCtxt tbenchCtxt;
  RpCtxt replierCtxt;

  char *zebuWork      = ZEBUWORK;
  char *designFeatures = DFFILE;

  tbenchCtxt.sentData    = 0;
  tbenchCtxt.dataExpected = false;
  tbenchCtxt.error       = 0;
  tbenchCtxt.end         = false;

  try {
    //open ZeBu
    XtorScheduler     * xsched = XtorScheduler::get();
    XtorSchedParams_t * XtorSchedParams = xsched->getDefaultParams();
    svt_c_runtime_cfg * runtime  = new svt_c_runtime_cfg();

    char *zebuWork      = ZEBUWORK;
    char *designFeatures = DFFILE;
    threading = new svt_pthread_threading();

    XtorSchedParams->useVcsSimulation  = false;
    XtorSchedParams->useZemiManager    = true;
    XtorSchedParams->noParams          = true;
    XtorSchedParams->zebuInitCb        = NULL;
    XtorSchedParams->zebuInitCbParams  = NULL;
```

```
    xsched->configure(XtorSchedParams) ;
    zemi3 = ZEMI3Manager::open(zebuWork,designFeatures);
    board = zemi3->getBoard();
    zemi3->buildXtorList(); // manually add the xtor or use buildXtorList
    zemi3->init();

    runtime->set_threading_api(threading);
    runtime->set_platform(new svt_zebu_platform(board, false));
    svt_c_runtime_cfg::set_default(runtime);

    cerr << "#TB : Register UART Transactor..." << board << endl;
    xtor_uart_svs::Register("xtor_uart_svs");

    test =
static_cast<xtor_uart_svs*>(Xtor::getNewXtor( Board::getBoard(),
"xtor_uart_svs",  xsched, NULL,runtime)) ;
    fprintf (stderr, "Found - Drivers [%s][%s] ...\n",
test->getDriverModelName (),test->getDriverInstanceName ());
    replier =
static_cast<xtor_uart_svs*>(Xtor::getNewXtor( Board::getBoard(),
"xtor_uart_svs",  xsched, NULL,runtime)) ;
    fprintf (stderr, "Found - Drivers [%s][%s] ...\n",
replier->getDriverModelName (),replier->getDriverInstanceName ());

    //open ZeBu
    printf("opening ZEBU...\n");

    test->init(board,"uart_device_wrapper.uart_driver_0");
    replier->init(board,"uart_device_wrapper.uart_driver_1");

    cerr <<"------------------------Zemi3
Start--------------------------" << endl;
    zemi3->start();   //This will start the Zemi3 service loop


    cerr <<"------------------------Wait for Reset
---------------------------" << endl;
    test->runUntilReset() ;
    cerr <<"------------------------Configuring
XTOR-------------------------" << endl;
    ok = test->setWidth(8);
    ok &= test->setParity(NoParity);
    ok &= test->setStopBit(TwoStopBit);
    ok &= test->setRatio(16);
    ok &= test->config();
    if (!ok ){ throw ("UART config failed");}

    ok = replier->setWidth(8);
    ok &= replier->setParity(NoParity);
    ok &= replier->setStopBit(TwoStopBit);
    ok &= replier->setRatio(16);
    ok &= replier->config();
    if (!ok ){ throw ("UART config failed");}
```

```cpp
      test->dumpSetDisplayErrors(true);
      test->dumpSetFormat(DumpASCII);
      test->dumpSetDisplay(DumpSplit);
      test->dumpSetMaxLineWidth(40);
      test->openDumpFile("test_dump.log");

      replier->dumpSetDisplayErrors(true);
      replier->dumpSetFormat(DumpASCII);
      replier->dumpSetDisplay(DumpSplit);
      replier->dumpSetMaxLineWidth(40);
      replier->openDumpFile("test_dump.log");

      test->setReceiveCB(rxCB,&tbenchCtxt);
      test->setSendCB(txCB,&tbenchCtxt);

      replier->setReceiveCB(rpRxCB,&replierCtxt);
      replier->setSendCB(rpTxCB,&replierCtxt);

      // TB Main loop
      while (!tbenchCtxt.end) {
        test    -> runClk (10) ;
        replier -> runClk (10) ;
      }

      test->closeDumpFile();
      replier->closeDumpFile();

      if (tbenchCtxt.error != 0) {
        printf("Detected %d errors during test\n",tbenchCtxt.error);
        ret = 1;
      }
    } catch (const char *err) {
      ret = 1; fprintf(stderr,"Testbench error: %s\n", err);
    }
    if (test != NULL)  { delete test; }
    if (replier != NULL) { delete replier; }

    zemi3->terminate();
    zemi3->close();

    printf("Test %s\n",(ret==0)?"OK":"KO");

    return ret;
  }
```

Using the Server Mode

Following are the typical testbench implementations using the *xtor_uart_svs* class in the client mode:

- The testbench starts a UART transactor server on TCP port 10000 of the test machine with an IP address in the following format: *ddd.ccc.bbb.aaa*.

- With the server started, any TCP client can connect to

  ◦ the UART transactor server.

  ◦ The testbench ends when the client disconnects.

```
#include <stdexcept>
#include <exception>
#include <queue>
#include <sys/time.h>
#include <libZebu.hh>
#include "xtor_uart_svs.hh"
#include <string.h>
using namespace ZEBU;
using namespace ZEBU_IP;
using namespace XTOR_UART_SVS;
using namespace std;

#include "svt_report.hh"
#include "svt_pthread_threading.hh"
#include "svt_cr_threading.hh"
#include "svt_c_threading.hh"
#include "svt_zebu_platform.hh"
#include "TopScheduler.hh"
#include "XtorScheduler.hh"

#include "libZebuZEMI3.hh"
#include "svt_systemverilog_threading.hh"
#include "svt_simulator_platform.hh"
#define THREADING svt_systemverilog_threading
svt_c_threading * threading = NULL ;

#define xstr(s) str(s)
#define str(s) #s

#ifndef TCP_PORT
#define TCP_PORT    10000
#endif

#ifndef TCP_SERVER
#define TCP_SERVER "localhost"
#endif

#ifndef TCP_MODE
```

```c
#define SERVER_MODE   true
#else
#define SERVER_MODE   ( strncmp( "client" , TCP_MODE ,6) != 0 )
#endif

uint8_t convertData ( uint8_t data )
{
  uint8_t ret = data;
  if (!((ret < 'A') || (ret > 'z'))) {
    ret = (ret < 'a')?(ret+0x20):(ret-0x20);
  }
  return ret;
}

typedef struct {
  queue<uint8_t> dataQueue;
} RpCtxt;

// Replier TX Callback
bool rpTxCB ( uint8_t* data, void* ctxt )
{
  bool send = false;
  RpCtxt* rpCtxt = (RpCtxt*)ctxt;

  if (!((rpCtxt->dataQueue).empty())) {
    *data =  convertData((rpCtxt->dataQueue).front());
    (rpCtxt->dataQueue).pop();
    send = true; // send next data
  }
  return send;
}

// Replier RX Callback
void rpRxCB ( uint8_t data, bool valid, void* ctxt )
{
  RpCtxt* rpCtxt = (RpCtxt*)ctxt;
  if (valid) {
    (rpCtxt->dataQueue).push(data);
  } else {
    (rpCtxt->dataQueue).push(0x0);
  }
}

//################################
//  main
//################################
int main ( ) {
  int ret = 0;
  Board       *board   = NULL;
  xtor_uart_svs*       test    = NULL;
  xtor_uart_svs*        replier = NULL;
  ZEMI3Manager *zemi3  = NULL ;
  bool ok;
```

```cpp
    RpCtxt replierCtxt;
    uint16_t tcpPort     = TCP_PORT;
    char*    tcpServer    = TCP_SERVER;
    bool     serverMode   = SERVER_MODE;

    try {
      //open ZeBu
      printf("opening ZEBU...\n");
      //open ZeBu
      XtorScheduler     * xsched = XtorScheduler::get();
      XtorSchedParams_t * XtorSchedParams =
   xsched->getDefaultParams();
      svt_c_runtime_cfg * runtime  = new svt_c_runtime_cfg();

      char *zebuWork      = ZEBUWORK;
      char *designFeatures = DFFILE;
      threading = new svt_pthread_threading();

      XtorSchedParams->useVcsSimulation  = false;
      XtorSchedParams->useZemiManager    = true;
      XtorSchedParams->noParams          = true;
      XtorSchedParams->zebuInitCb        = NULL;
      XtorSchedParams->zebuInitCbParams  = NULL;
      xsched->configure(XtorSchedParams) ;
      zemi3 = ZEMI3Manager::open(zebuWork,designFeatures);
      board = zemi3->getBoard();
      zemi3->buildXtorList(); // manually add the xtor or use
   buildXtorList
      zemi3->init();

      runtime->set_threading_api(threading);
      runtime->set_platform(new svt_zebu_platform(board, false));
      svt_c_runtime_cfg::set_default(runtime);

      cerr << "#TB : Register UART Transactor..." << board << endl;
      xtor_uart_svs::Register("xtor_uart_svs");

      test =
   static_cast<xtor_uart_svs*>(Xtor::getNewXtor( Board::getBoard(),
   "xtor_uart_svs"  ,  xsched, NULL,runtime)) ;
      fprintf (stderr, "Found - Drivers [%s][%s] ...\n",
   test->getDriverModelName (),test->getDriverInstanceName ());
      replier =
   static_cast<xtor_uart_svs*>(Xtor::getNewXtor( Board::getBoard(),"xt
   or_uart_svs" ,  xsched, NULL,runtime)) ;
      fprintf (stderr, "Found - Drivers [%s][%s] ...\n",
    replier->getDriverModelName (),replier->getDriverInstanceName ());

      printf("Going through UART drivers...\n"); fflush(stdout);

      test->setOpMode (ServerMode) ;
      test->init(board,"uart_device_wrapper.uart_driver_0");
      replier->init(board,"uart_device_wrapper.uart_driver_1");
```

```
    cerr <<"-----------------------Zemi3
Start---------------------------" << endl;
    zemi3->start();   //This will start the Zemi3 service loop
    cerr <<"----------------------Wait for Reset
--------------------------" << endl;
    test->runUntilReset() ;
    cerr <<"-----------------------Configuring
XTOR-------------------------" << endl;
    ok =  test->setWidth(8);
    ok &= test->setParity(NoParity);
    ok &= test->setStopBit(TwoStopBit);
    ok &= test->setRatio(16);
    ok &= test->config();
    if(!ok) { throw ("Could not configure UART"); }

    ok =  replier->setWidth(8);
    ok &= replier->setParity(NoParity);
    ok &= replier->setStopBit(TwoStopBit);
    ok &= replier->setRatio(16);
    ok &= replier->config();
    if(!ok) { throw ("Could not configure UART"); }

    test->dumpSetDisplayErrors(true);
    test->dumpSetFormat(DumpASCII);
    test->dumpSetDisplay(DumpSplit);
    test->dumpSetMaxLineWidth(40);
    test->openDumpFile("test_dump.log");

    replier->dumpSetDisplayErrors(true);
    replier->dumpSetFormat(DumpASCII);
    replier->dumpSetDisplay(DumpSplit);
    replier->dumpSetMaxLineWidth(40);
    replier->openDumpFile("replier_dump.log");
    replier->setReceiveCB(rpRxCB,&replierCtxt);
    replier->setSendCB(rpTxCB,&replierCtxt);

    if (serverMode) {
      char hostName[1024];
      printf("\n\n  -- Start TCP server  -- \n");
      if (!test->startServer(tcpPort)) {throw ("Could not start UART
Server."); }
      if (gethostname(hostName, 1024) == 0) {
        printf("\nTCP server is up, TCP client may now be connected
to %s:%u\n\n",hostName,tcpPort);
      }
      while (!test->isConnected()) {}
      printf("\nTCP client is now connected\n\n");
    } else {
      printf("\n\n  -- Starting TCP connection  -- \n");
      if (!test->startClient( tcpServer, tcpPort)) {throw ("Could
not start UART Server."); }
    }
```

```
    printf("\n\n  -- Starting testbench  -- \n"); fflush(stdout);
    while (test->isConnected()) {
      test    -> runClk (10) ;
      replier -> runClk (10) ;
    }

    test->closeDumpFile();
    replier->closeDumpFile();

  } catch (const char *err) {
    ret = 1; fprintf(stderr,"Testbench error: %s\n", err);
  }

  if (replier != NULL) { delete replier; replier = NULL;}
  if (test != NULL)    { delete test; test = NULL;}

  zemi3->terminate();
  zemi3->close();

  printf("Test %s\n",(ret==0)?"OK":"KO");

  return ret;
}
```

For instance, to connect from a Windows-based machine using Microsoft's HyperTerminal:

1. Start the HyperTerminal from Windows XP.

2. Create a new connection, in the **Connection** dialog box:

Set the host address to the host machine's name or to its IP address (*ddd.ccc.bbb.aaa*).

Set the port number to 10000.

Set the connection type to TCP/IP (*winsock*).

1. Click **OK** to connect the terminal.

Figure 7   HyperTerminal Connection Configuration
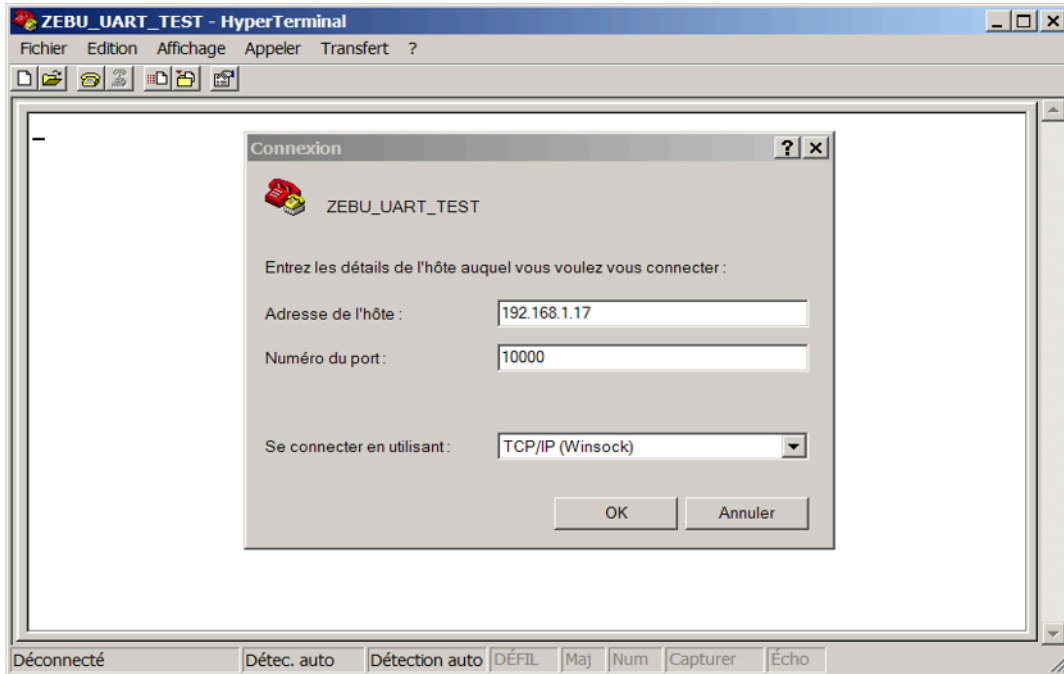
Using the XtermMode

The transactor is directly linked to a virtual console, sending commands and receiving results from the ZeBu DUT. Here is a typical implementation of a testbench using the xtor_uart_svs class operating in XtermMode :

```
#include <stdexcept>
#include <exception>
#include <queue>
#include <sys/time.h>
#include <libZebu.hh>
#include "xtor_uart_svs.hh"
#include <ctype.h>
#include "svt_report.hh"
#include "svt_pthread_threading.hh"
#include "svt_cr_threading.hh"
#include "svt_c_threading.hh"
#include "svt_zebu_platform.hh"
#include "TopScheduler.hh"
#include "XtorScheduler.hh"
#include "libZebuZEMI3.hh"
#include "svt_systemverilog_threading.hh"
#include "svt_simulator_platform.hh"
```

```
#define THREADING svt_systemverilog_threading


using namespace ZEBU;
using namespace ZEBU_IP;
using namespace XTOR_UART_SVS;
using namespace std;

uint8_t convertData ( uint8_t data )
{
  uint8_t ret = data;
  if (!((ret < 'A') || (ret > 'z'))) {
    ret = (ret < 'a')?(ret+0x20):(ret-0x20);
  }
  return ret;
}

typedef struct {
  queue<uint8_t> dataQueue;
} RpCtxt;

// Replier TX Callback
bool rpTxCB ( uint8_t* data, void* ctxt )
{
  bool send = false;
  RpCtxt* rpCtxt = (RpCtxt*)ctxt;

  if (!((rpCtxt->dataQueue).empty())) {
    *data =  convertData((rpCtxt->dataQueue).front());
    (rpCtxt->dataQueue).pop();
    send = true; // send next data
  }
  return send;
}

// Replier RX Callback
void rpRxCB ( uint8_t data, bool valid, void* ctxt )
{
  RpCtxt* rpCtxt = (RpCtxt*)ctxt;
  if (valid) {
    (rpCtxt->dataQueue).push(data);
    #if TEST_JMB==0
    if(isalnum(data)){
        (rpCtxt->dataQueue).push('[');(rpCtxt->dataQueue).push('{');
    (rpCtxt->dataQueue).push(data);
    (rpCtxt->dataQueue).push(toupper(data));
    (rpCtxt->dataQueue).push(tolower(data));
    if (isdigit(data)) {
    for (int i=(data-'0');i>=0;i--)  (rpCtxt->dataQueue).push(data);
    }
            (rpCtxt->dataQueue).push(']');
  (rpCtxt->dataQueue).push('}');
    }
```

```
      #endif
    } else {
      (rpCtxt->dataQueue).push(0x0);
    }
  }

  //##############################
  //  main
  //##############################
  int run_test (const char *testName) {
    int ret = 0;
    Board        *board = NULL;
    ZEMI3Manager *zemi3 = NULL ;
    const unsigned   nbxtor_uart_svsMax = 2;
    unsigned         nbxtor_uart_svs = 0;
    xtor_uart_svs        * uarts[nbxtor_uart_svsMax];
    const char     * uartInstNames[nbxtor_uart_svsMax];
    bool ok;
    RpCtxt replierCtxt;

    try {
      //open ZeBu
      printf("opening ZEBU...\n");
      //open ZeBu
      XtorScheduler    * xsched = XtorScheduler::get();
      XtorSchedParams_t * XtorSchedParams = xsched->getDefaultParams();
      svt_c_runtime_cfg * runtime   = new svt_c_runtime_cfg();

      char *zebuWork      = ZEBUWORK;
      char *designFeatures = DFFILE;
      svt_c_threading * threading = new svt_pthread_threading();

      XtorSchedParams->useVcsSimulation   = false;
      XtorSchedParams->useZemiManager     = true;
      XtorSchedParams->noParams           = true;
      XtorSchedParams->zebuInitCb         = NULL;
      XtorSchedParams->zebuInitCbParams   = NULL;
      xsched->configure(XtorSchedParams) ;
      zemi3 = ZEMI3Manager::open(zebuWork,designFeatures);
      board = zemi3->getBoard();
      zemi3->buildXtorList(); // manually add the xtor or use buildXtorList
      zemi3->init();

      runtime->set_threading_api(threading);
      runtime->set_platform(new svt_zebu_platform(board, false));
      svt_c_runtime_cfg::set_default(runtime);

      cerr << "#TB : Register UART Transactor..." << board << endl;
      xtor_uart_svs::Register("xtor_uart_svs");

      uarts[0] =
  static_cast<xtor_uart_svs*>(Xtor::getNewXtor( Board::getBoard(),
  "xtor_uart_svs" ,  xsched, NULL,runtime)) ;
```

```
   fprintf (stderr, "Found - Drivers [%s][%s] ...\n", uarts[0]
->getDriverModelName (),uarts[0] ->getDriverInstanceName ());
   uarts[1] =
static_cast<xtor_uart_svs*>(Xtor::getNewXtor( Board::getBoard(),
"xtor_uart_svs" ,  xsched, NULL,runtime)) ;
   fprintf (stderr, "Found - Drivers [%s][%s] ...\n",
uarts[1]->getDriverModelName (),uarts[1]->getDriverInstanceName ());

   uartInstNames[0] = "uart_device_wrapper.uart_driver_0";
   uarts[0]->setOpMode (XtermMode) ;
   uarts[0]->init(board,uartInstNames[0]);
   uarts[0]->setDebugLevel(2);
   uarts[0]->setName("UART_XTERMINAL_0");
   uarts[0]->setConvMode(Conv_DOS_BSR) ; // Conv_None , Conv_ASCII
(DOS) : Conv_DOS_BSR
   uartInstNames[1] = "uart_device_wrapper.uart_driver_1";
   uarts[1]->init(board,uartInstNames[1]);
   uarts[1]->setDebugLevel(2);uarts[1]->setName("UART_STD_1");

   nbxtor_uart_svs = 2;


   cerr <<"------------------------Zemi3
Start---------------------------" << endl;
   zemi3->start();   //This will start the Zemi3 service loop
   cerr <<"------------------------Wait for Reset
---------------------------" << endl;
   uarts[0]->runUntilReset() ;
   cerr <<"------------------------Configuring
XTOR--------------------------" << endl;
   for (unsigned int i = 0; (i < nbxtor_uart_svs); ++i) {
     // Config UART interface
     ok =  uarts[i]->setWidth(8);
     ok &= uarts[i]->setParity(NoParity);
     ok &= uarts[i]->setStopBit(TwoStopBit);
     ok &= uarts[i]->setStopBit(TwoBitStop);
     ok &= uarts[i]->setRatio(16);
     ok &= uarts[i]->config();

     if (ok) {
       char logFname[1024];
       // Configure and open dump file
       uarts[i]->dumpSetDisplayErrors(true);
       uarts[i]->dumpSetFormat(DumpASCII);
       uarts[i]->dumpSetDisplay(DumpSplit);
       uarts[i]->dumpSetMaxLineWidth(80);
       sprintf(logFname,"%s_dump.log", uartInstNames[i]);
       uarts[i]->openDumpFile(logFname);
     }

     if(!ok) { throw ("Could not configure UART"); }
   }
```

```
   if (nbxtor_uart_svs > 1) {
     uarts[1]->setReceiveCB(rpRxCB,&replierCtxt);
     uarts[1]->setSendCB(rpTxCB,&replierCtxt);
   }

   printf("\n\n  -- Starting testbench  -- \n"); fflush(stdout);
   // Testbench main loop
   while (uarts[0]->isAlive()) {
     uarts[0]->runClk (10) ;
     uarts[1]->runClk (10) ;
   }

   for (unsigned int i = 0; (i < nbxtor_uart_svs); ++i)
 { uarts[i]->closeDumpFile(); }

 } catch (const char *err) {
   ret = 1; fprintf(stderr,"Testbench error: %s\n", err);
 }
 for (unsigned int i = 0; (i < nbxtor_uart_svs); ++i) { delete
uarts[i]; }

 zemi3->terminate();
 zemi3->close();

 printf("Test %s\n",(ret==0)?"OK":"KO");

 return ret;
}
```

The testbench starts an xterm window. Every character typed in the xterm window is sent to the transactor interface without being echoed. All the data received by the UART transactor is displayed in the respective terminal.

Once the terminal is terminated (using CTRL+C or pressing the window's kill button), the testbench ends.

Figure 8   UART Terminal Window When the Testbench Ends

Using the zRci Flow

This section explains how to run the transactor using the zRci flow.

In this example, the transactor operates in the Server mode. It can be used with both Zcei clockport and RTL clock

In zrci testcase, transactor works as client and a server should be connected to do communication.

"nc -vlk 6050" is used to establish the connection, due to incompatibility in -d option of ncat command across versions. This command is also made user specific and hence user can provide their own command as: NC_CMD='nc -vlk 6050' with run command.

```
int en_dmtcp=0;

extern "C" void*  zRci_param(const std::string& key, const std::string&
 value) {

  printf
 ("==================================================================\n
");
  printf ("zRci_param %s %s\n",key.c_str(),value.c_str());
  printf
 ("==================================================================\n
");

  if (strcmp(key.c_str(),"en_dmtcp")==0) {
    en_dmtcp = std::atoi(value.c_str());
    printf("info parameter en_dmtcp received: %d\n",en_dmtcp);
  }

  return NULL;
}

extern "C" void*        zRci_pre_board_open(const ZRCI::TbOpts& tbopts) {
  printf("===============================\n");
  printf("      zRci_pre_board_open       \n");
  printf("===============================\n");

  return NULL;
}

extern "C" void*        zRci_post_board_open(const ZRCI::TbOpts& tbopts)
 {
  printf("===============================\n");
  printf("      zRci_post_board_open      \n");
  printf("===============================\n");

  board = tbopts.board;
```

```cpp
        return NULL;
}
extern "C" void*        zRci_pre_board_init(const ZRCI::TbOpts& tbopts) {
    printf("==============================\n");
    printf("       zRci_pre_board_init        \n");
    printf("==============================\n");

    uart_server_inst_name = "uart_device_wrapper.uart_driver_0";
    uart_replier_inst_name = "uart_device_wrapper.uart_driver_1";

    xsched = XtorScheduler::get();
    XtorSchedParams_t * XtorSchedParams = xsched->getDefaultParams();
    runtime  = new svt_c_runtime_cfg();
    threading = new svt_pthread_threading();

    XtorSchedParams->useVcsSimulation  = false;
    XtorSchedParams->useZemiManager    = true;
    XtorSchedParams->noParams          = true;
    XtorSchedParams->zebuInitCb        = NULL;
    XtorSchedParams->zebuInitCbParams  = NULL;
    xsched->configure(XtorSchedParams) ;

    return NULL;
}
extern "C" void*        zRci_post_board_init(const ZRCI::TbOpts& tbopts)
 {
    printf("==============================\n");
    printf("       zRci_post_board_init      \n");
    printf("==============================\n");
    printf ("\nInitializing XTORs\n");

    runtime->set_threading_api(threading);
    runtime->set_platform(new svt_zebu_platform(board, false));
    svt_c_runtime_cfg::set_default(runtime);

    xtor_uart_svs::Register("xtor_uart_svs");

    // SERVER INST
    uart_server_main = static_cast<xtor_uart_svs*>(Xtor::getNewXtor(board,
 "xtor_uart_svs",  xsched, NULL,runtime)) ;

 fprintf (stderr, "Found - Drivers [%s][%s] ...\n",
 uart_server_main->getDriverModelName (),
 uart_server_main->getDriverInstanceName ());

    // Replier INST
    uart_replier = static_cast<xtor_uart_svs*>(Xtor::getNewXtor(board,
 "xtor_uart_svs",  xsched, NULL,runtime)) ;
    fprintf (stderr, "Found - Drivers [%s][%s] ...\n",
 uart_replier->getDriverModelName (),uart_replier->getDriverInstanceName
 ());

    printf("\n\n --- Testbench Start --- \n\n");
```

```
  fflush (stdout);

  return NULL;
}
extern "C" void*        zRci_cleanup(const ZRCI::TbOpts& tbopts) {
  printf("==============================\n");
  printf("          zRci_cleanup         \n");
  printf("==============================\n");

  uart_server_main->closeDumpFile();
  uart_replier->closeDumpFile();
  delete uart_server_main;
  delete uart_replier;
  return NULL;
}

extern "C" std::string zRci_command(const std::string& key, const
 std::string& value) {
  printf("==============================\n");
  printf("          zRci_command         \n");
  printf("==============================\n");

  if (key == "preCP") {
    printf("==============================\n");
    printf("UART preCP called\n");
    printf("==============================\n");
    uart_server_main->prepareCheckpoint ();
    uart_replier->prepareCheckpoint ();
  }
  else if (key == "postCP") {
    printf("==============================\n");
    printf("UART postCP called\n");
    printf("==============================\n");
    uart_server_main->resumeCheckpoint();
    uart_replier->resumeCheckpoint();
  }

  else if (key == "initialize") { // Configure the Xtor (has to be done
 when clocks are freerunning)
    int uart_ratio = 16;
    init_uart (board, uart_server_main, "UART_SERVER__INST",
 uart_server_inst_name);
    config_uart (uart_server_main, uart_server_inst_name, uart_ratio,
 true);
    uart_server_main->enableTimestamp(false);
    init_uart (board, uart_replier, "UART_REPLIER_INST",
 uart_replier_inst_name);
    config_uart (uart_replier, uart_replier_inst_name, uart_ratio,
 false);
    uart_replier->enableTimestamp(false);
  }
  else if (key == "uart_close") {
    printf("UART TCP Connection close\n");
```

```
    //uart_server_main->closeDumpFile();
    bool close = false;
    close = uart_server_main->closeConnection();
    if (close) {
      printf("Client closed successfully\n");
      fflush(stdout);
    }
 }
 else if (key == "uart_reconnect") {
    printf("UART TCP reconnection\n");
    uart_server_main->openDumpFile
((char* )"uart_device_wrapper.uart_driver_0_dump.log");

    if (!uart_server_main->reConnect()) {
      printf("Couldn't reconnect\n");
    }
    else printf("Re-connections successful\n");
 }
 else if (key == "send_traffic") {
    // Start a new line
    uart_replier->send(0x0A, true);
    // Send all data b/w 0x10 & 0xFE
    uint8_t data = 0x10;
    while (data < 0xFF) {
      if (uart_replier->send(data, true)) {
        data++;
      }
      uart_replier->runClk(1);
    }
    // Start a new line
    uart_replier->send(0x0A, true);
 }
 else if (key == "debug") {
    uint8_t val = stoi (value) ;
    uart_server_main->setDebugLevel(val);
    uart_replier->setDebugLevel(val);
 }

 return key;
}

}
```

- For simulation -->

  ○ To compile ¨make compile SIMULATOR=1 TBENCH=<>

  ○ To run ¨ make run SIMULATOR=1 TBENCH=<>

- For zebu compile--> make compile HW_TYPE=<Platform_type>

- For example: make compile HW_TYPE=ZS4

- For more advance options please refer README

The DUT instantiates two UART controllers connected to two UART transactors as described in the following figure:

Figure 9   UART Transactor Tutorial Example
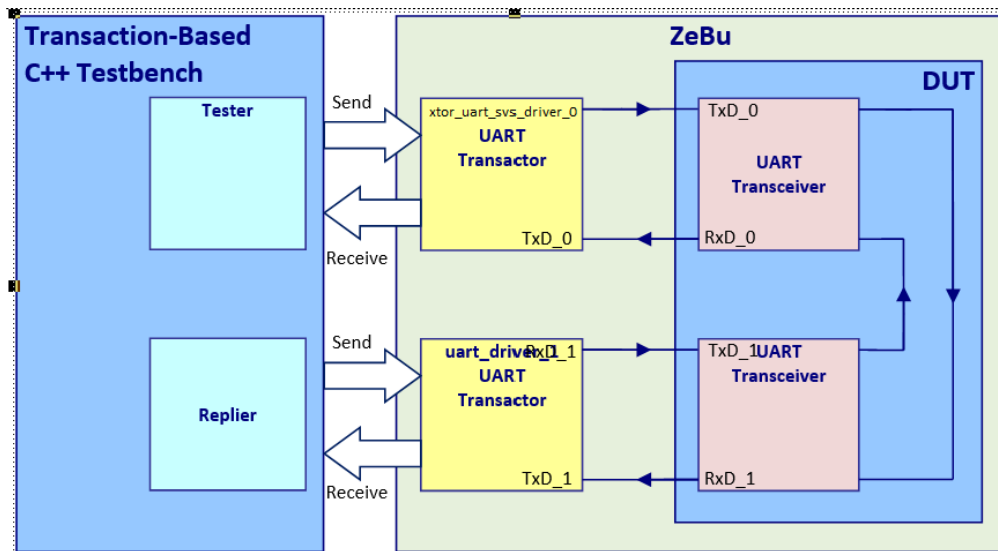
The testbench is a C++ program, which configures the UART transactors, has following two components:

- One transactor, called replier, is a xtor_uart_svs object. The replier receives data, processes it, and sends the processed data back.

- The other transactor, called tester, is a xtor_uart_svs object according to the selected testbench.

The data received by the replier is processed in the following way:

- Lowercase letters, uppercase letters.

- Uppercase letter , lowercase letters.

- No processing on other characters.

The testbenches using the xtor_uart_svs object (blocking send and receive; non-blocking send and receive; and callback testbenches), sends a set of data and check the value returned by the replier.

The testbench, when operating in XtermMode, starts a terminal. The data typed in the terminal is sent to the test interface and the data received on the test interface is displayed on the terminal.

The testbench, when operating in ServerMode, starts a TCP client or a TCP server according to the specified parameters. The user needs to launch a TCP client or server connection to connect the UART server. The data is transmitted between the TCP client or server and the UART transactor in the following way:

- When running the UART in server mode, the TCP client must be connected when the UART server starts. The testbench waits for the TCP client to be connected before it starts.

- When using the UART in client mode, the TCP server must be started before the testbench.

This example is available in the *XTOR/xtor_uart_svs.<version>/example* directory and the testbench source code is available in the example/src/bench subdirectory.

Figure 10   Tx Data Waveforms
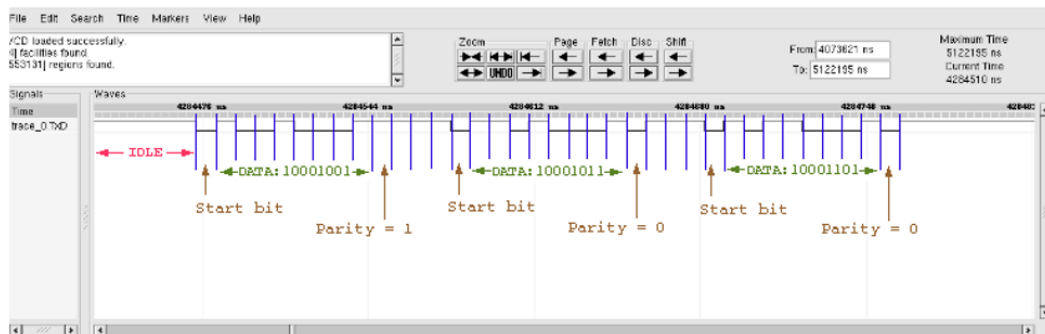
5

Baud Rate Calculation

This chapter describes how to calculate the baud rate in a ZeBu design.

This section describes the following sub-topics.

- Definition
- Calculating the Baud Rate

Definition

The baud rate in a ZeBu design is measured between the DUT and the UART transactor. As in actual designs, the UART baud rate is a division of the UART controller clock frequency, which is linked to the UART transactor controlled clock. For UART transactor with *clk_ref* input, the UART baud rate is a division of this reference clock.

The effective baud rate of the UART transactor is defined by the user using a ratio parameter. The ratio parameter of the UART matches the ratio between the UART transactor controlled clock frequency or the input *clk_ref* (DUT frequency) and the baud rate (*BaudRate*); it can be set using the *Uart::setRatio* method (see setRatio() Method):

$$BaudRate = \frac{DUT frequency}{ratio} \Leftrightarrow ratio = \frac{DUT frequency}{BaudRate}$$

Many times the same "UART controller clock/BaudRate" ratio in the ZeBu environment and the reference system environment (as specified in the UART controller DUT user setup), because all clocks and interface speeds are scaled down with the same factor:

$$\frac{DUT frequency_{ZeBu}}{BaudRate_{ZeBu}} = \frac{DUT frequency_{Ref}}{BaudRate_{Ref}}$$

However, it leads to an important slowdown of the effective UART baud rate in the verification environment.

The ratio parameter is an integer; you must therefore use the nearest value when the calculated value is a decimal number.

Reference System Environment		ZeBu Environment		
System Clock Frequency (MHz)	Reference Baud rate (bps)	DUT Clock Frequency (MHz)	Ratio Parameter	Resulting ZeBu Baud Rate (bps)
100	115,200	10	868	11,520
100	115,200	5	868	5,760
50	115,200	5	434	11,520
100	19,200	10	5,208	1,920
50	19,200	5	2,604	960

In the preceding table, notice how the resulting baud rate in the ZeBu environment is different from the system environment initial baud rate (in *blue*).

The other possible setup is to keep an identical baud rate in both the environments, regardless of the DUT clock frequency in ZeBu. The ratio parameter then varies differently according to the DUT clock frequency in ZeBu, as shown in the following table:

Table 7

Reference System Environment		ZeBu Environment	
System Clock Frequency (MHz)	Reference Baud Rate (bps)	DUT Clock Frequency (MHz)	Ratio Parameter
100	115,200	10	87
100	115,200	5	43
50	115,200	5	43
100	19,200	10	520
50	19,200	5	260

In the preceding tables, notice how the ratio parameter in the ZeBu environment has evolved (in *green*).

Calculating the Baud Rate

This section describes the methods to calculate the baud rate.

Using the UART Transactor Baud Rate Detector

The UART transactor includes a baud rate detector that can be used to estimate the ratio parameter value from the real baud rate transmitted by the DUT. To provide accurate results, the UART transactor needs to receive at least one data word from the DUT. The process is as follows:

1. Write the testbench to set the ratio to an estimated value. The getDetectedRatio() Method can then be used in the testbench to obtain the ratio detected by the UART transactor. When releasing the transactor, if the detected ratio is different from the user-specified ratio, a warning message is displayed to signal the detected ratio value to the user.

2. Run the testbench and get the ratio value detected by the UART transactor:

```
UART Xactor  Warning    : Specified and detected clock ratio differs
                              ( detected=16 / specified=64 )
```

1. Update the testbench with the UART ratio previously detected, if necessary.

2. Check again whether the data from the DUT is correctly received by the transactor.

Alternative Method

The ratio parameter description in the previous section may prove useless if you do not have all the necessary information about the UART controller setup. Here is a method based on the measurements for baud rate evaluation:

1. Determine which clock signal is your transactor controlled clock (see Connecting the Design Clocks).

2. Create an RxD transactor port waveform with the controlled clock attached to the UART transactor, using driver clock sampling.

3. Count the number of cycles from the UART clock for a single bit transfer, like the minimum delay between two consecutive edges of the RxD transactor signal (for

example, a STOP to START transition). This count is the "frequency/baud rate" Q quotient which may be equal to the ratio parameter:

Figure 11   Frequency/Baud Rate
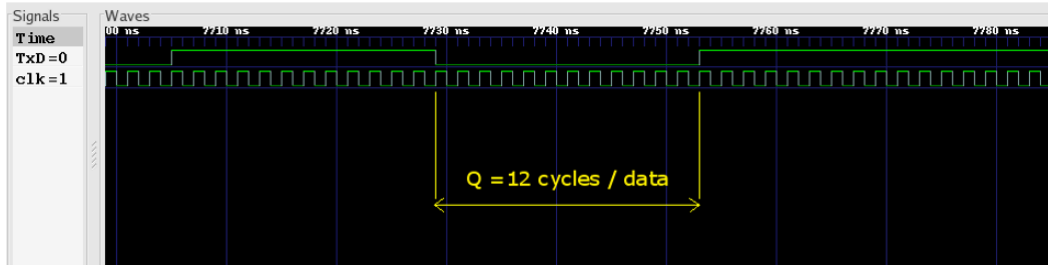
Note:

The waveform above shows the RxD port on a transactor connected to the TxD output signal from the DUT.

6

Debugging the Transactor

This section explains the following topics:

- Logs

- Debug Probe Information

Logs

Transactor generates two types of logs; one is traffic log which can be controlled by using API openDumpFile(). Another one is complete transactor functionality log which can be controlled by using *API setLog()*.

Debug Probe Information

For debugging purposes, a list of signals are available at runtime to determine the internal behavior of the UART transactor.

A diagnostic port is provided on the interface of the transactor. The diagnostic port can be accessed at runtime by any of the existing debug mechanisms such as dynamic probes. The name of this 197-bit diagnostic port is debug_probe [196:0]; the designation of the 197 bits is described by the following table. This diagnostic port encodes into a single bit-vector all the information needed to analyze the behavior of your memory models.

Table 8   Debug Probe InformationDC

*Bit*Position/Range	TypePS: Protocol SpecificIS: Implementation Specific	Description
Top Level Debug		
196	PS	Error in serial receive data
195	PS	Glitch in serial received data
194	PS	Rx data transition detected
193	IS	Clear to send filter

Table 8   Debug Probe InformationDC (Continued)

192	IS	Serial receive data filter
191	IS	Data capture
190	IS	Read Hold
189	IS	Configuration Ready
188	IS	Write Acknowledged
187	IS	Sets Ready to Send Signal
186	IS	Ratio Acknowledgement
185	IS	Design Under Test - Ready
184:182	--IS	FSM State Bits
181	IS	Write Enable Bit
180	IS	Not used
179:148	IS	Transmitted Message Information
147	IS	Control Message transmission from hardware to software is complete
146	IS	Control Message transmission from hardware to software is pending
145:114	IS	Data transmit message
113	IS	Data Message transmission from hardware to software is complete
112	IS	Data Message transmission from hardware to software is pending
107	PS	Frame stop bit configuration
106	IS	Transmitter flow enabled
105	PS	Enables Transmitters
104	IS	Tx FIFO empty
103	IS	Transmission FIFO full
58	IS	Receiver Synchronize

Feedback

Table 8   Debug Probe InformationDC (Continued)

57	IS	Data Message transmission from hardware to software is pending
56	IS	Read Enable
55	IS	Receiver FIFO full
54	IS	Receiver FIFO empty
53	PS	Parity Bit
52:51	PS	Configured data width
50	PS	Enables Receiver
Arbiter Signals----		
111	IS	Control Message transmission from hardware to software is complete (true)
110	IS	Control Message transmission from hardware to software is pending (false)
109:108	IS	FSM State Bits
Transmission Datapath		
102	PS	Enables break condition
101	IS	Read Acknowledged
100	IS	Start bit
99	IS	Hold bit
98k	IS	Read Enable
97	IS	Allows Reading
96	IS	Allows writing
95:92	IS	Write pointer
91:88	IS	Read pointer
87	IS	Write error interrupt
86	IS	Write acknowledge interrupt

Table 8   Debug Probe InformationDC (Continued)

85	IS	Read error interrupt
84	IS	Read acknowledge Interrupt
83:80	IS	Count of number of data in TX FIFO
79:72	IS	Shift register for Tx data
71	IS	Parity Value
70	IS	Resets the counter
69	IS	Shift Bit
68	IS	Capture Bit
67	IS	Serial data Transmission
66	IS	Send Ready
65	IS	Xtor is sending break condition
64:61	IS	Local counter used in Tx FSM
60:59	IS	TX FSM State bits
Receiver's Datapath		
49:42	IS	Data Out
41	IS	Write Acknowledged Bit
40	IS	Hold Bit
39	IS	Write Enable Bit
38	IS	Read Allow Bit
37	IS	Write Allow Bit
36:33	IS	Write pointer Bit
32:29	IS	Read pointer Bit
28	IS	Write error interrupt Bit
27	IS	Write acknowledge interrupt Bit

Table 8   Debug Probe InformationDC (Continued)

26	IS	Read error interrupt Bit
25	IS	Read acknowledge interrupt Bit
24:21	IS	Count of number of data in RX FIFO
20:13	IS	Local buffer while receiving the data
12:9	IS	Local count variable for the receiver FSM
8:4	PS	Configured data width
3	IS	Error detection bit
2	IS	Previous serially received data
1:0	IS	FSM State bits