Integrating with ZeBu thread-safe library

AN025
Revision b

ZeBu Application Note

September 2008

Purpose:

This document presents the advantages of multithreading for ZeBu. It introduces the specific features of the ZeBu thread-safe library (`libZebuThreadSafe`) and presents its functional differences with `libZebu`.
This document details the C++ interface and its usage and lists the equivalent functions for the C interface.

Applicability:

This document is applicable for ZeBu software version 4.2_0 and upper.

History:

This table gives information about the content of each revision of this manual, with indication of specific applicable version:

Doc Revision	Date	Evolution
b	Sept 08	Review for customer distribution.
a	Aug 08	Preliminary Information.

Copyright Notice
Proprietary Information

Right to Copy Documentation

The license agreement with EVE permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

"*This document is duplicated with the permission of EVE, for the exclusive use of* _____ *and its employees. This is copy number* _____."

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

EVE AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Table of Contents

1 Introduction

1.1 Multithreading and Thread-Safe Advantages with ZeBu

In a multi-transactors environment, transactors are often interdependent and have to run concurrently.

If several transactors control synchronous clocks or are synchronized with the same clock domains, their software parts have to run in parallel and the concurrency between them has to be controlled. For example, if they do not run concurrently, any stalled transactor can stop clocks indefinitely and lead to a deadlock by starving all other transactors which can no longer advance clocks or exchange messages.

In the ZeBu environment, the parallelism is controlled in the testbench and two different programming choices are available:

- In a multi-process environment, each process loads its own memory resources, in particular the ZeBu runtime database. Moreover, the ZeBu environment supports a maximum of 2 simultaneous processes (only 1 in the zTIDE environment)
- In a multi-thread environment, the memory resources, in particular the ZeBu runtime database, are shared. Moreover, the maximum number of threads is not a fixed number but depends on the amount of memory and on the operating system of your workstation (in most cases, the operating system supports up to 256 threads).

In a multi-thread environment, the ZeBu thread-safe library provides the appropriate security: all the threads share the same memory resources but only one thread can access the shared data, in particular the ZeBu database, at a given time.

For that purpose, the ZeBu thread-safe library implements serialized accesses to critical resources with minimum impact on the behavior which is very similar to a multi-process environment. This library has also been designed to minimize the re-writing of the testbench and transactors when migrating from a single- or multi-process environment to a multi-thread environment.

General information about multi-process and multi-thread programming can be found for example in the Wikipedia on-line encyclopedia (http://en.wikipedia.org/) with “multiprocessing”, “multithreading” or “thread-safe” keywords for example.

1.2 Thread-Safe Libraries

To use ZeBu thread-safe library, you have to link your testbench and transactors with the `libZebuThreadSafe.so` library in replacement of `libZebu.so`. Typically any testbench program linked to the `libZebu.so` library can be linked to the `libZebuThreadSafe.so` library without modification or recompilation.

Note that a thread-safe library also exists for zTIDE environment. The corresponding thread-safe library is `libZebuThreadSafeDebug.so` which has to be used in replacement of `libZebuDebug.so` library.

1.3 Performance

The ZeBu thread-safe library automatically switches the context between threads in a similar way to multi-process so that it has similar performance and behavior, without major modification of the testbench.

However, the performance of a multithreaded environment can be impacted by different situations:

- In a multi-CPU environment, the ZeBu thread-safe library introduces overhead and serialization latencies.
- If the time balance between threads in the CPU is not appropriate. This can be avoided by using port waiting and by modifying the port priority which are described later in this document.

Note that the ZeBu thread-safe library can also be used for a single-thread testbench but it impacts the overall performance because of the overhead of the thread-safe security mechanisms.

2 Specific features of the thread-safe ZeBu library

2.1 Modified Interface for Thread-safe Environment

Most methods of the ZeBu API are not impacted by the thread-safe implementation of the ZeBu library: they are automatically serialized so that only one thread executes them at a given time. Their behavior is the same with both `libZebu.so` and `libZebuThreadSafe.so` libraries.

The methods which are impacted by the thread-safe environment are the methods which control the transactional interface:

C++ Method	C Function
`ZEBU::TxPort::isPossibleToSend`	`ZEBU_Port_isPossibleToSend`
`ZEBU::RxPort::isPossibleToReceive`	`ZEBU_Port_isPossibleToReceive`
`ZEBU::TxPort::waitToSend`	`ZEBU_Port_waitToSend`
`ZEBU::RxPort::waitToReceive`	`ZEBU_Port_waitToReceive`
`ZEBU::Port:setGroup`	`ZEBU_Port_setGroup`
`ZEBU::RxPort::WaitGroup` (*)	`ZEBU_Port_WaitGroup` (*)
`ZEBU::TxPort::write`	`ZEBU_Port_write`
`ZEBU::RxPort::read`	`ZEBU_Port_read`
`ZEBU::TxPort::sendMessage`	`ZEBU_Port_sendMessage`
`ZEBU::RxPort::receiveMessage`	`ZEBU_Port_receiveMessage`
`ZEBU::TxPort::flush`	`ZEBU_Port_flush`
`ZEBU::Board::serviceLoop`	`ZEBU_Board_serviceLoop`
`ZEBU::CDriver::run`	`ZEBU_Driver_run`
`ZEBU::CDriver::wait`	`ZEBU_Driver_wait`

Note that the `ZEBU::RxPort::WaitGroup` method exists only in the thread-safe environment.

2.2 Ports and Threads in the Thread-safe Environment

The user must make sure that several threads do not access a given port at the same time, which means that the software part of a transactor linked to a port has to run in a single thread and cannot be parallelized on several threads.

The prerequisite for a transactor to switch to a different thread, for instance after initialization or at closing, is that only one thread initializes and closes all transactors.

When the testbench program contains several threads, a port must always be associated to and accessed by a single thread.

2.3 Thread-context switch issue

The ZeBu thread-safe library provokes voluntary context switches between threads in order to synchronize thread scheduling with message flow and to optimize performance of multithreaded testbenches.

In parallel, some shared resources are accessed by using atomic operations so that they are not interrupted by other threads. The parallel usage of atomic operations creates involuntary context switches between threads but too many context switches result in poor performance.

The ZeBu thread-safe library has been designed so that the transactional interface minimizes involuntary context switches: accesses to shared resources of ports are grouped and globalized to minimize usage of atomic operations and the number of involuntary context switches. This introduces some usage constraints and some behavioral changes described in the following sections.

2.4 Sending Messages from Software to Hardware

Since each port can only be accessed by one thread, some delay appears when transmitting messages from the software side to the hardware side. In other words, the messages sent by means of the `ZEBU::TxPort::sendMessage` method are buffered. Messages are actually sent to the hardware side when a full or empty message buffer is polled through the following methods:

- `ZEBU::TxPort::isPossibleToSend`
- `ZEBU::RxPort::isPossibleToReceive`
- `ZEBU::Board::serviceLoop`

In most cases, the testbench fills the message buffer with streaming messages or tries to receive some reply messages after a call to the `ZEBU::TxPort::sendMessage` method. In such cases a full or empty buffer is polled and messages are sent at this time.

However, you can force the transmission of messages by adding the `ZEBU::TxPort::flush` method in your testbench. However, this operation often decreases performance, so it must be used only when it proves necessary.

2.5 Port waiting

When the `TxPort::isPossibleToSend`, `RxPort::isPossibleToReceive` and `Board::ServiceLoop` methods are used for polling the shared resources and ports, it is recommended to factorize the polling to significantly improve the performance.

Factorizing can be obtained by replacing or preceding the standard port polling by Port Waiting which consists in waiting for a port or at least one port of a port group to be ready to send or to receive a new message. All waiting threads are suspended then awakened by a global thread that polls all waiting ports.

This factorization minimizes the usage of atomic operations and the number of involuntary context switches and improves CPU time balance between threads according to their message flow. Threads that rarely exchange messages are suspended for a long time and use very little CPU time.

When the number of threads is greater than the number of CPUs this factorization also significantly optimizes (x2 to x50) the global frequency of multithread testbenches

The waiting mechanism is different when it concerns a single port or a group of ports.

2.5.1 Waiting for a single port

The `ZEBU::TxPort::waitToSend` and `ZEBU::RxPort::waitToReceive` methods allow waiting for a port to be ready to send or receive a new message.

Note: It is useless to call the methods `ZEBU::TxPort::isPossibleToSend` and `ZEBU::RxPort::isPossibleToReceive` after return of the `ZEBU::TxPort::waitToSend` and `ZEBU::RxPort::waitToReceive` methods.

2.5.2 Waiting for a port group

A port group is identified by an integer. All the ports with the same identifier belong to the same group. The `ZEBU::Port:setGroup` method attaches a port to a given group.

The `ZEBU::Port::WaitGroup` method allows waiting for at least one port of a group to be ready to send or receive a new message. Typically this method should be executed just before the `ZEBU::Board::serviceLoop` method, which can call back ready ports after waiting.

Note that cancelling the `ZEBU::RxPort::WaitGroup` method with `pthread_cancel` function causes a thread error which throws an exception in the thread-safe environment. As a possible workaround, developers should use a specific loop with a specified timeout, as in the following example:

```
int timeoutExpired = 0;
do {
     timeoutExpired = Port::WaitGroup(
        board,
        group identifier,
        1000 /* 1 ms timeout*/
     );
} while(timeoutExpired == 0 || isFinished != 0);
```

Here `isFinished` is a predicate to abort waiting. This predicate is assigned by another thread that wants to abort waiting in order to suspend or finish the task of the current thread. For instance this can be necessary to interrupt transactors, to save the hardware state, or to close the testbench. There is no service loop in the example because port group waiting can be used without service loop.

2.5.3 Recommendations for attaching ports to port groups

It is inefficient to attach input ports (SW → HW) in a group if it is not necessary to wait for them to be ready to send new messages. If some input ports are polled uselessly, that might interrupt waiting prematurely on bad events and produce counter-productive effects.

It is inefficient to attach input and output ports of a transactor to the same group because it is rarely necessary to wait to simultaneously send and receive a new message in a transactor.

In a hardware-event-driver transactor it is necessary to wait to receive a new message and it may also be necessary to wait before replying with a new message. These are distinct waiting steps for which it is best to use two distinct groups. Else it is only necessary to wait to receive and therefore unnecessary to attach input ports.

It is inefficient to attach input ports that do not need to be taken into account as control ports that are used only in specific steps such as initialization for instance. It may also be inefficient to attach input ports that are not used to send streaming messages. Most of the time, it is possible through design to send a new message when the software side needs it. In this case port group waiting just adds useless overhead which reduces performance.

2.5.4 Servicing port groups

The `ZEBU::Board::serviceLoop` method services the ports belonging to the same group and it executes the callbacks of ports attached to a thread with respect of the port/thread constraints specified previously. The following example shows how to attach all the ports of a thread to the same group, how to wait for the group, and then how to callback ports of the group.

```
#include <pthread.h>
port1->setGroup(pthread_yield());
// pthread_yield return the identifier of the current thread
port2->setGroup(pthread_yield());
...
Port::WaitGroup(pthread_yield());
board->serviceLoop(NULL, NULL, pthread_yield());
```

The `ZEBU::Board::serviceLoop` method allows specifying a `ServiceLoopHandler`. It is a callback executed at the end of the loop. It stops or resumes the loop in order to minimize the number of returns from the loop as well as the usage of atomic operations and the number of thread context switches. The following sequence shows how to define and use a `ServiceLoopHandler`:

```
static int AServiceLoopHandler(void * /*unused*/, int pending)
{
return (isFinished == false) & pending;
// return from the service loop if there is no pending message
}
board>
serviceLoop(AServiceLoopHandler, NULL, ...);
```

2.6 Port priority modes

The ZeBu thread-safe library defines pseudo-priorities between threads according to their message flow. It has an effect on the number of context switches in each thread according to their success in polling the ports. It extends or reduces non-preemptive sequences to minimize the usage of atomic operations causing involuntary context switches and it controls voluntary context switches according to the priority mode or the round priority of the polled ports.
The priority mode can be changed by setting the `ZEBU_PortPriorityMode` C variable to a value of the `ZEBU_PortPriotityType` enumerated type defined in the `$ZEBU_ROOT/include/Types.h` file. Available mode values are the following:

```
typedef enum _ZEBU_PortPriorityType
{
Z_NO_PREENTIVE,
Z_ROUND_PRIORITY,
Z_NO_PRIORITY
} ZEBU_PortPriotityType;
```

The priority mode should be set before opening a connection, as in the following example:

```
ZEBU_PortPriorityMode = Z_ROUND_PRIORITY;
Board *board = Board::open(...);
```

The ZeBu thread-safe library uses by default "no port" priority which corresponds to the `Z_NO_PRIORITY` mode. Each mode is described in the following sections.

Note that the misspelling in `Z_NO_PREENTIVE` will be corrected in a future release of the ZeBu thread-safe library: `Z_NO_PREEMPTIVE` will be the new value and `Z_NO_PREENTIVE` will still exist for compatibility purpose.

2.6.1 Z_NO_PRIORITY

Each port access may lead to a thread-context switch. A voluntary thread-context switch is done whenever busy ports are polled, knowing that a busy port is a port on which it is not possible to send or receive new messages.

2.6.2 Z_NO_PREENTIVE

Port accesses of a thread are gathered in the same non-preemptive sequence until no busy ports are polled. During this non pre-emptive sequence, other threads may be suspended as soon as they start a port access. Suspended threads are awakened alternatively after the end of a non-preemptive sequence.

No thread-context switch is provoked by the thread-safe environment until the polled ports are ready. A thread can keep the CPU for a longer time until it succeeds in exchanging new messages. This gives maximum CPU time to threads that exchange messages continuously.

If the ZeBu throughput is sufficient, ports polled by a thread are always ready and the non-preemptive sequence of the thread is never interrupted; it starves all other

threads and leads to a deadlock. This deadlock may particularly occur when asynchronous transactors run on different threads, or when transactors have no clock relation and do not depend on the progress of other transactors. In this case the `Z_ROUND_PRIORITY` mode must be used.

2.6.3 Z_ROUND_PRIORITY

Each port has a priority, which is computed dynamically according to its polling successes. As soon as a busy port is polled the priority of the port is reduced to the lowest priority and the current thread provokes a context switch. Else the current thread continues a non-preemptive sequence until the priority of the last accessed port is equal to or higher than the priority of all other ports.

The priority of a port decreases at each access according to a round-robin policy which gives an equal chance to each port to be accessed when it is not busy.

Time for non-preemptive sequences is extended while balancing CPU's time equitably between threads.

This mode also has the advantage of removing some voluntary and inefficient thread-context switches when the ZeBu hardware-side latency is too high in comparison with software-side polling frequency. This mode can give better CPU time balance for interactive transactors, which send messages and then wait for reply messages or which receive messages and then send a reply message.

2.6.4 Priority mode constraints

In `Z_NON_PREEMPTIVE` and `Z_ROUND_PRIORIRTY`, a non-preemptive sequence can never be interrupted if no more port accesses are done by the thread which has started the sequence. That leads to a deadlock while other threads are waiting for access to shared resources of ports.

For instance, this deadlock might occur when closing the testbench if a thread exits without interrupting a non-preemptive sequence.

The `ZEBU::Board::closeThread` method prevents such deadlocks by releasing any shared resource held by the current thread. It must be called before termination of each thread.

2.7 Co-Simulation driver

The `ZEBU::CDriver::run` and `ZEBU::CDriver::wait` methods of the C++ co-simulation driver have been modified in the ZeBu thread-safe library because they are synchronous and they could hang when the specified number of cycles has not been reached.

2.7.1 Run

The `ZEBU::CDriver::run` method runs a controlled clock for a deterministic number of cycles.

The thread-safe layer spins polling if run length is short. Else it estimates the minimal time necessary to perform the run and sleeps the current thread to yield the CPU to other threads for part of this time.

The running time is computed according to the number of cycles to be run and the rated frequency (or maximal frequency) of the clock controlled by the driver. The thread is asleep for a first time slice. Then the real frequency of the previous time slice is measured by means of the clock counter. The running time is adjusted according to the real frequency of the controlled clock. The thread is asleep for a new time slice and so on until the end of the run.

2.7.2 Wait for trigger

The `ZEBU::CDriver::wait` method waits for triggers or for timeout expiry.

The thread-safe layer sends a run command and waits for a reply message. During the wait the co-simulation thread is suspended and then awakened by another thread when the reply message has been received.

The wait feature can be used instead of the run feature if there is no trigger to wait for and the number of cycles to wait is given as a timeout.

2.7.3 Difference between run and wait

In the ZeBu thread-safe library, both run and wait do not have the same synchronization mechanism and one or the other should be chosen to match the testbench needs.

In most cases, the run feature gives the best result for short run lengths because spin polling of the run mechanism is faster than the wait mechanism.

On the other hand, the wait feature gives the best results for long run lengths: the wait mechanism is better than the estimated sleeps of the run mechanism because the spin polling cannot be used for long run lengths with impacting the CPU's time balance between threads.

The number of cycles after which it is preferable to use the wait feature is difficult to estimate and must be measured empirically with the testbench environment.

3 EVE Contacts

For product support, contact: support@eve-team.com.

For general information, visit our company web-site: http://www.eve-team.com

Europe Headquarters	EVE SA 2bis, Voie La Cardon Parc Gutenberg, Batiment B 91120 PALAISEAU France Tel: +33-1-64 53 27 30
US Headquarters	EVE-USA, Inc. 2290 N. First Street, Suite 304 San Jose, CA 95131 USA Tel: 1-888-7EveUSA (+1-888-738-3872)
Japan Headquarters	Nihon EVE KK KAKiYA Building 4F 2-7-17, Shin-Yokohama Kohoku-ku, Yokohama-shi, Kanagawa 222-0033 JAPAN Tel: +81-45-470-7811
Korea Headquarters	EVE Korea, Inc. 804 Kofomo Tower, 16-3, Sunae-Dong, Bundang-Gu, Sungnam City, Kyunggi-Do, 463-825, KOREA Tel: +82-31-719-8115
India Headquarters	EVE DESIGN AUTOMATION Pvt. Ltd. #B-15, Raheja Arcade, 80 Ft. Road, 5th Block, Koramangala, Bangalore - 560 095 Karnataka, INDIA Tel: +91-80-41460680/ 30202343
Taiwan Headquarters	14F1, No 371, Sec. 1 Guangfu Rd., East District Hsinchu City 300 Taiwan (R.O.C.) Tel: +886 (3) 564 7900