ERIKA Enterprise API Tutorial

for the Altera Nios II Platform

version: 1.0.6
December 11, 2012

About Evidence S.r.l.

Evidence is a spin-off company of the ReTiS Lab of the Scuola Superiore S. Anna, Pisa, Italy. We are experts in the domain of embedded and real-time systems with a deep knowledge of the design and specification of embedded SW. We keep providing significant advances in the state of the art of real-time analysis and multiprocessor scheduling. Our methodologies and tools aim at bringing innovative solutions for next-generation embedded systems architectures and designs, such as multiprocessor-on-a-chip, reconfigurable hardware, dynamic scheduling and much more!

Contact Info

Address:
Evidence Srl,
Via Carducci 56
Località Ghezzano
56010 S.Giuliano Terme
Pisa - Italy
Tel: +39 050 991 1122, +39 050 991 1224
Fax: +39 050 991 0812, +39 050 991 0855

For more information on Evidence Products, please send an e-mail to the following address: info@evidence.eu.com. Other informations about the Evidence product line can be found at the Evidence web site at: http://www.evidence.eu.com.

Contents

1 Introduction

This short tutorial consists of a set of small application targeted to show the peculiarities of the main Erika Enterprise API primitives.

The three applications described in this tutorial are the following:

Task API Example. The example shows the main primitives about task management, including the options of preemptive and non-preemptive scheduling.

Resource and Application Mode API Example. The example shows the usage of the resource primitives and their interaction with the scheduling policies, as well as the usage of the Application Modes.

Event and Alarm API Example. The example shows how to use the primitives implementing event handling. It also shows the usage of Erika Enterprise alarms, and the configuration of separate stacks for the application tasks.

The demos have been reported working with the following evaluation board:

- Altera Stratix 1s40 evaluation Board
- Altera Stratix 2s60 RoHS evaluation Board

Other evaluation boards should work as well, because these examples only use the following set of peripherals (which are normally included in the examples provided by Altera):

- an Interval timer (the System Clock) for periodic alarms;
- a button;
- an Avalon PIO with eight LEDs;
- a JTAG UART.

Please report any problem, comment, and suggestion to the Evidence technical support, by writing a mail to `support@evidence.eu.com`.

1.1 How to compile the demos

This tutorial assumes the reader's familiarity with the Nios II IDE, and the ability to create, compile and run an application using Erika Enterprise. However, in this section we provide a brief step-by-step guide to the setup and usage of the main tools for running this tutorial. You can find more information on these topics in the Erika Enterprise tutorial available in the ERIKA Enterprise Reference Manual.

To compile and run the examples described in this document, follow the next steps:

Installation of the development environment

1. If you want to use the *Lauterbach Trace32 Debugger and Tracer* [1] to flash and run the tutorial, copy the content of the `files\demo\kernel\orti` directory of the CDROM into `C:\t32\demo\kernel\orti`.

2. Download the *Quartus II Web edition* environment from the *Altera* web site and install the software on your computer.

3. Download the *Nios II IDE Web edition* environment from the *Altera* web site and install the software on your computer.

4. Download the *Erika Enterprise and RT-Druid Demo Version for Altera Nios II* from the *Evidence* website, and install the software on your computer.

Creation of the Jam images

1. Start the *Quartus II* environment.

2. Click on *File* → *Open Project*.

3. Open the `C:\altera\80\nios2eds\examples\...\full_featured\full_featured.qpf` project.

4. Click on *Tools* → Programmer.

5. Check the *Prog/conf* checkbox.

6. Click on *File* → *Create/Update* → *Create JAM*.

7. Rename the file as `fpga.jam`[1].

8. Save the project.

9. Click on *File* → *Open Project*.

10. Open the `C:\altera\80\nios2eds\examples\...\standard\standard.qpf` project.

11. Click on *Tools* → Programmer.

12. Check the *Prog/conf* checkbox.

13. Click on *File* → *Create/Update* → *Create JAM*.

14. Rename the file as `fpga.jam`[2].

15. Save the project.

[1] The file will be created in `C:\altera\80\nios2eds\examples\...\full_featured`.
[2] The file will be created in `C:\altera\80\nios2eds\examples\...\standard`.

Compiling the examples

To run the examples, you need to create two system libraries, one called `standard_syslib` (linked to the `standard` example provided by Altera), the other called `full_featured_syslib` (linked to the `full_featured` example provided by Altera).

Compile the examples in the following way:

1. Start the *Nios2 IDE* environment.
2. Click on *Windows* → *Preferences* → *RT-Druid* → *Oil* → *OS Configurator* and choose *Source distribution*.
3. Click on *New* → *Project* → *Altera Nios II* → *System Library*.
4. As project name, specify `standard_syslib`.
5. Build the project.
6. Click on *New* → *Project* → *Altera Nios II* → *System Library*.
7. As project name, specify `full_featured_syslib`.
8. Build the project.
9. Click on *New* → *Project* → *Evidence* → *RTDruid Oil and c/c++ project*.
10. Select one of the templates, as in Figure 1.1
11. As project name, specify `nios2_task`[3].
12. Reference the two libraries already created (i.e. `standard_syslib` and `full_featured_syslib`).
13. Build the project[4].
14. If you want to use the *Lauterbach Trace32 Debugger and Tracer* [1] to flash and run the tutorial, run the `Debug.bat` script and click on `go` once the T32 environment has finished initialization.

Warning: To run the examples as provided in this package without modifications, you need to create the system libraries with the *same name and location* as specified in the `SYSTEM_LIBRARY_NAME` and `SYSTEM_LIBRARY_PATH` attributes in the OIL file provided with each demo!

[3] For the `resource` and the `events` examples, just repeat these steps by creating a different project and importing the related files.

[4] The ELF binary will be created in the directory `C:\altera\80\nios2eds\bin\eclipse\workspace\nios2_task\Debug\default_cpu`.

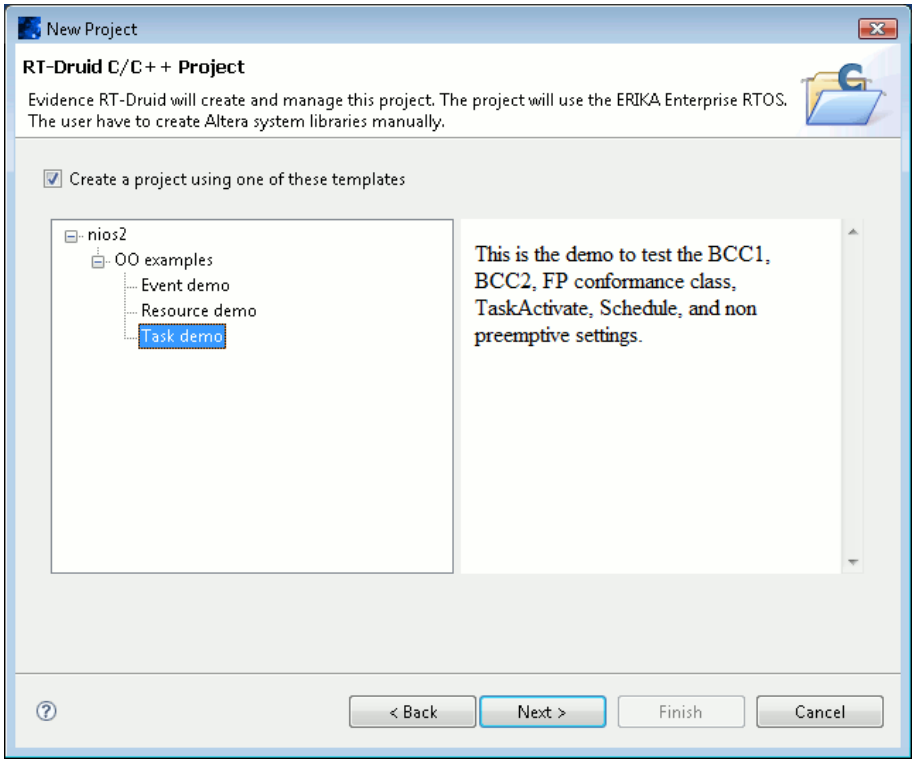

Figure 1.1: Creating a new project from a template.

2 Task API Example

The Task API Example, also called the *Christmas Tree* example, is a simple demo that shows the usage of the following primitives:

- `DeclareTask`;
- `ActivateTask`;
- `TerminateTask`;
- `Schedule`.

2.1 Demo structure

The demo consists of two tasks, `Task1` and `Task2`. `Task1` repeatedly puts on and off a sequence of LEDs, like a Christmas Tree. `Task2` simply turns on and off a LED, and is activated by the press of a button. Task2 is de facto a *disturbing* task that, depending on the configuration parameters, may preempt `Task1`.

The demo can be compiled in four different configurations, showing in each case a different scheduling behavior. To select a configuration, please look at the end of the OIL file containing the configuration of the example, and uncomment *only one* configuration at a time.

2.2 Configuration 1: Full preemptive

This configuration is characterized by the following properties:

- An Altera HAL alarm `Task1_alarm_callback` is linked to the System Clock (typically an Altera Interval Timer). The alarm simply activates `Task1`.
- Each instance of `Task1` simply shows a *Christmas Tree*, that is, `Task1` controls a sequence of LEDs blinking from LED 0 to LED 5.
- each time the first button in the board is pressed, `Task2` is activated.
- `Task2` always preempts `Task1` (it has higher priority). `Task2` does the following actions:
  - LED 6 or LED 7 blink alternatively;
  - a string is printed to the JTAG UART displaying the number of alarms fired, the number of button interrupt fired, the number of times Task2 has run.

2.2.1 Running configuration 1

The behavior of the two tasks in configuration 1 can be evaluated looking at the evaluation board LEDs together with the console output.

When the demo starts, the first string that appears on the console is the banner

```
Welcome to the ERIKA Enterprise Christmas Tree!
```

meaning the application has started correctly.

After that, the main() function calls `StartOS()`. `StartOS` is an Erika Enterprise primitive that initializes the Erika Enterprise environment. The call to `StartOS()` must precede any call to the Erika Enterprise API functions.

`StartOS` automatically activates `Task2`. This can be configured by modifying the following property of `Task2` in the OIL configuration file:

```
AUTOSTART = TRUE;
```

As a result, `Task2` runs, blinking LED 6 and printing the following string:

```
Task2 - Timer:   0 Button:   0 Task2:   1
```

- The first number is the number of times the HAL alarm has run so far. Every execution of the alarm callback activates `Task1`, that in turn blinks LED 0 to 5.
- The second number displays the number of executions of the button interrupt. Every time the button is pressed or released, an interrupt is generated. The interrupt activates `Task2`.
- The third number displays the number of times `Task2` has run. As we will see later, the number may differ from the number of button activations.

Now press the first button in the evaluation board *for MORE THAN ONE SECOND*, and release it. As a result, two lines are printed on the console. The first line corresponds to the `Task2` activation issued by the interrupt generated when you pressed the button. The second line correspond to the `Task2` activation issued by the interrupt generated when you released the button.

Moreover, looking at the LEDs on the evaluation board, every time you press or release the button, LEDs 6 or 7 blink regardless of the current status of the Christmas Tree (that is, the current code being run by `Task1`).

The behavior happens because `Task2` has an *higher priority* than `Task1`: when `Task2` is activated, it has a higher priority than `Task1`, and as a result `Task2` "preempts" `Task1`[1].

Now please press the button rapidly a few times. As a result, a few lines are printed on the console. As it can be seen looking at the number printed for button presses and `Task2` executions, the number of `Task2` Activations are less than the number of Button IRQs. The behavior happens because the OIL file for configuration 1 contains the line:

```
KERNEL_TYPE = BCC1;
```

[1]Please note that this behavior will change in Configuration 2, 3, and 4.

that is, Erika Enterprise is configured with a BCC1 conformance class, which forces every task to maintain only one activation at a time. Task2 activations issued when Task2 is in the ready queue or is running are *lost* (in that case the ActivateTask primitive returns E_OS_LIMIT). We will address this issue in Configuration 4.

You are now ready to proceed to Configuration 2.

To change configuration, go at the end of the OIL file, and leave uncommented only the Configuration you want to try. After that, please rebuild the application.

2.3 Configuration 2: Non preemptive

This configuration differs from Configuration 1 only for the fact Task1 is configured as *NON preemptive* in the OIL file by using

```
SCHEDULE = NON;
```

in the task attributes of the OIL file.

2.3.1 Running configuration 2

The effect of Task1 being non-preemptive is that Task2 only runs when Task1 (the Christmas Tree) is not running. The fact can be perceived by looking at the board LEDs: LEDs 6 or 7 only blinks when the Christmas Tree is off, that is when Task1 ended its current activation.

If compared with configuration 1, the lines printed on the console by Task2 show the correct number of alarm executions, with the correct number of alarm and button interrupts, meaning that interrupts are handled also when a non-preemptive task is running.

On the other side, the number of times Task2 has run could be less than in configuration 1 because some more activations could have been lost due to the fact Task2 cannot preempt Task1.

2.4 Configuration 3: Preemption points.

This configuration differs from configuration 2 for the fact Task1 calls Schedule() in the middle of the display of the Christmas Tree. To implement that, the OIL file specifies a global

```
EE_OPT = "MYSCHEDULE";
```

option, that translates in a

```
#define MYSCHEDULE
```

statement in the generated files when compiling the application source code (see the automatically generated file Debug/default_cpu/eecfg.h inside your project directory). Then, Task1 source code includes a conditional compiler directive to add the call to Schedule().

2.4.1 Running configuration 3

This configuration shows the usage of the `Schedule()` primitive, which can be used to implement a preemption point inside a non-preemptive task.

In particular, `Task2` can now preempt `Task1` in the middle of the Christmas tree (between the blinking of LED 2 and LED 3). The fact is visible looking at the board LEDs: if the button is pressed while LED 0, 1, or 2 are on, then in any case LED 6 (or 7) will blink between the blink of LED 2 and 3.

2.5 Configuration 4: Multiple Activations.

This configuration differs from configuration 3 because `Task2` has now the possibility of storing a few pending activations. To do that, the number of pending activations for `Task2` is set to 6 by having the line

```
ACTIVATION = 6;
```

to control the pending activations of `Task2` and the line

```
KERNEL_TYPE = BCC2;
```

to set the kernel conformance class to `BCC2`.

2.5.1 Running configuration 4

When running configuration 4, `Task2` now stores up to 6 pending activations, which means that if the task is activated more than once (by pressing and releasing the button rapidly a few times) while Task1 is running , then `Task2` is executed consecutively for a corresponding number of times up to the maximum number of pending activations specified in the OIL File.

2.6 Configuration 5: Minimal configuration.

This configuration differs from configuration 1 because the system is running with the `FP` conformance class. As a result, pending activations are stored inside an integer, and for that reason no activations are lost when clicking the button.

2.6.1 Running configuration 5

When running configuration 5, `Task2` now stores all your pending activations, which means that if the task is activated more than once (by pressing and releasing the button rapidly a few times) while Task1 is running , then `Task2` is executed consecutively for a corresponding number of times. Pleae note that the order of activations of tasks with the same priority is not maintained as it is with conformance class `BCC2` and `ECC2`.

3 Resource and Application Mode API Example

The *Resource and Application Mode API Example* is a simple demo that shows the usage of the following primitives:

- `GetActiveApplicationMode`;
- `GetResource`;
- `ReleaseResource`;

The demo also uses `ActivateTask` and other task related primitives. Please refer to Section 2 for a better explanation of these primitives.

3.1 Demo structure

The demo consists of two tasks, `LowTask` and `HighTask` that share a *resource*. `LowTask` is a periodic low priority task, activated by a timer, with a *long* execution time. Almost all its execution time is spent inside a critical section on the resource (that is, between a call to `GetResource` and `ReleaseResource`). To show this behavious, LED 0 is turned on when `LowTask` is inside the critical section.

`HighTask` is a high priority task that increments (decrements) a counter depending on the application mode being `ModeIncrement` (`ModeDecrement`). The task is *aperiodic*, and is activated by the ISR linked to the button. Most of the execution time of `HighTask` is spent inside a critical section on the shared resource. To show this behaviour, LED 1 is turned on when `HighTask` is inside the critical section.

The application uses *Application Mode*s to implement a task behavior dependent on the Application Mode. When the application starts, it checks the first pin of the Button PIO to select one out of the two different application modes (`ModeIncrement` or `ModeDecrement`).

An *Application Mode* is basically a system configuration that is selected when the application starts. Application modes are set at application startup, and are not changed after the call to `StartOS()`. Application modes can be used to model particular application configurations such as "Debug Mode", "Normal Mode", and so on. The application tasks can read the application mode using `GetActiveApplicationMode`, and adjust their behavior depending on the particular mode. Please also note that the `AUTOSTART` property in the OIL file for tasks and alarms may have different values for each application mode.

After the call to `StartOS()`, all the calls to `printf` are always done in mutual exclusion inside the `mutex` critical section, to ensure that there is no concurrency issues between the various `printf` calls.

`HighTask` and `LowTask` are configured to share the same stack by setting the following line inside the OIL task properties:

```
STACK = SHARED;
```

As a result, only a single stack will be used, with a dimension that (in this case) is the sum of the stack space required by both tasks. In general, by appropriately using priorities and preemption thresholds the stack space can be reduced significantly.

Access to shared resources is managed by using the implementation of the Immediate Priority Ceiling protocol in Erika Enterprise. In practice, whenever `LowTask` locks the resources with primitive `GetResource()`, its priority is raised to the one of `HighTask`. This fact prevents `HighTask` from preempting `LowTask` while in the critical section, ensuring data consistency. Of course, `LowTask` will return to its priority when calling `ReleaseResource()` and the scheduler will be called to check whether a preemption is necessary.

3.2 Running the example

Compile and run the application as usual. When the application starts, the Button PIO is read, and the appropriate mode is passed to `StartOS`. let's suppose that the Button is not pressed: the application will start with the application mode set to `ModeIncrement`.

Each click of the first button in the evaluation board activates `HighTask`: if `LowTask` is already executing in critical section, `HighTask` will have to wait for `LowTask` to complete its critical section before being able to execute.

Pressing the button too fast will activate `HighTask` too fast. In that case, it is likely that some activation will be lost (as it happened in the Task API Example).

Starting the application with the first Button pressed, causes `StartOS` to be called with the `ModeDecrement` application mode. In this example, the application mode simply influences the increment or decrement of a counter, as it can be seen looking at the console outputs.

4 Event and Alarm API Example

The Event and Alarm API Example is a simple demo that shows the usage of the following primitives:

- `WaitEvent`;
- `GetEvent`;
- `ClearEvent`;
- `SetEvent`;
- `ErrorHook`;
- `StartupHook`;
- `SetRelAlarm`;
- `CounterTick`.

4.1 Demo structure

The demo consists of two tasks, `Task1` and `Task2`.

`Task1` is an *extended task*. Extended tasks are tasks that:

- can call blocking primitives (in our case, wait for events using the `WaitEvent` primitive;
- *must* have a separate stack (because they may suspend upon a call to a blocking primitive).

A task is considered an Extended Task when the OIL file includes events inside the task properties. In this example, `Task1` is an extended task because its properties contain the following lines:

```
TASK Task1 {
  ...
  EVENT = "TimerEvent";
  EVENT = "ButtonEvent";
};
```

`Task1` waits for two events:

- A `TimerEvent`. The action related to this event is the blinking of LED 1. The event is set by a periodic alarm notification `AlarmTask1` defined in the OIL file.
- A `ButtonEvent`. The action related to this event is the blinking of LED 2. The button event is set explicitly using the `SetEvent` primitive inside the button interrupt handler.

The buttons available on the evaluation board are attached to an interrupt handler. This interrupt handler sets a relative one-shot alarm `AlarmTask2`. The alarm notification of `AlarmTask2` activates `Task2`. `Task2` simply turns LED 3 on.

The demo also includes an `ErrorHook`. To understand the usage of `ErrorHook`, please place a breakpoint in the `ErrorHook` function. Every time an error appears in the execution of a Erika Enterprise primitive, then `ErrorHook` will be called. In the demo, such a condition appears when the button is pressed rapidly twice. In that case, a few button interrupts will be generated, and each execution of the interrupt handler will call the `SetRelAlarm` primitive. When `SetRelAlarm` will try to activate an alarm already armed, then `ErrorHook` will be called with a parameter `E_OS_STATE`.

The alarm support in Erika Enterprise is basically a wakeup mechanism that can be attached to application or external events (such as timer interrupts) to implement an asynchronous notification. In this example, the Erika Enterprise alarm support is used to implement a replacement of the Altera HAL alarm feature. To obtain this feature, the demo programs the `HIGH_RES_TIMER` timer to periodically raise an interrupt.

> **Warning:** To run this demo, the Altera System Library project should specify `none` as Timestamp timer. In this way, we are sure that the High Res Timer will not be registered by the Altera HAL.

The timer interrupt calls the `CounterTick` function. The `CounterTick` function simply increments the counter passed as parameter, checking if any pending Alarm notification has to be executed.

Please note that `CounterTick` can be attached to any source of interrupt, and can be called from any point of your application to implement user-defined wakeup mechanisms.

The demo contains a `StartupHook`, which contains the registration of the timer and button interrupts.

Finally, please note that `Task1` uses a blocking primitive like `WaitEvent`. This implies that `Task1` needs a separate stack for its execution. Therefore, `Task1` properties in the OIL file include the following lines:

```
TASK Task1 {
  ...
  STACK = PRIVATE_NIOSII {
    SYS_SIZE = 1024;
  };
};
```

that basically reserve 1024 bytes for the private stack of `Task1`.

Another interesting feature of Erika Enterprise is the possibility of reserving a separate stack for the execution of the interrupt handlers. This feature has been used in the OIL file of this demo, reserving 512 bytes for the IRQ stack with the with the following lines:

```
CPU test_application {
  OS EE {
    ...
    CPU_DATA = NIOSII {
      ...
      MULTI_STACK = TRUE {
        IRQ_STACK = TRUE {
          SYS_SIZE=512;
        };
        DUMMY_STACK = SHARED;
      };
    };
  };
};
```

Having an IRQ stack separated from the rest of the stacks allows a better sizing of task stacks (that does not have to leave space for IRQ handlers on each separate stack).

4.2 Running the example

Compile and run the application as usual.

> **Warning:** Please remember the Altera System Library project should specify `none` as Timestamp timer!!!

The only action done by `main` is to call `StartOS()`, which registers the two sources of IRQ (the button and the timer), automatically activates `Task1`, and arms a periodic alarm `AlarmTask1`.

Every time the timer interrupt fires, the counter `Counter1` will be incremented. Every alarm expiration, Alarm `AlarmTask1` will fire, setting the event `TimerEvent` on Task `Task1`: `Task1` wakes up, get the event, and blinks LED 1. The visible result is that LED 1 periodically blinks on the board.

Now press the button on the board once. An interrupt is generated that activates an interrupt handler that performs two actions:

- It arms the alarm `AlarmTask2`, whose notification will activate `Task2`. Then `Task2` switches LED 3 on and off.
- It sets an event `ButtonEvent` on `Task1`. As a result, `Task1` wakes up switching LED 2 on and off.

The visible result is that upon a press of the button, LED 2 immediately blinks (meaning `Task1` has been woken up by the event), and after a while (around 1 second) LED 3 blinks again (meaning the alarm `AlarmTask2` fired activating `Task2`).

If the button is pressed rapidly, an error in `SetRelAlarm` is raised, executing `ErrorHook`. The visible result in this case is that *all* the LEDs blink signaling the execution of `ErrorHook`.

4.3 Lauterbach Trace32 support

This demo also shows the integration of Erika Enterprise with the Lauterbach Trace32 Debugger and Tracer [1].

The integration supported by Erika Enterprise includes the following features:

- Automatic generation of the Trace32 PRACTICE Debug scripts to program the FPGA, and to load the ELF files produced in the IDE.
- Automatic generation of multicore debug scripts for Multicore designs.
- Kernel awareness support using ORTI files automatically generated by RT-Druid.

To enable all these features, you need to specify a JAM file name[1] inside the OS section of the OIL file, as well as the specification of the ORTI sections that should be generated, as follows:

```
CPU test_application {
  OS EE {
    ...
    NIOS2_JAM_FILE = "JAM_filename.jam";
    ORTI_SECTIONS = ALL;
  }
  ...
}
```

(`ALL` means the generation of all the ORTI information).

As a result of the compilation process, a set of files are produced inside the `Debug` directory (see Table 4.1 for a detailed list).

To run the Trace32 debugger, just double click on the `Debug/debug.bat` file generated during the compilation. The debugger opens up showing a window similar to the one in Figure 4.1.

Please note that each window has a title with the name of the CPU being under debug. The menu list include a submenu named "ee_cpu_0" containing the specification of the ORTI related debug features.

By clicking on each menu item, you can get useful debug informations about Erika Enterprise. In particular:

[1] JAM is one of the file formats containing the FPGA configuration that is accepted by Lauterbach Trace32

File name	Description
`debug.bat`	This batch script loads the FPGA hardware and starts a T32 instance for each CPU. You can double click it on the Nios II IDE to directly launch the debug session.
`debug_nojam.bat`	This batch script starts a T32 instance for each CPU. You can double click it on the Nios II IDE to directly launch the debug session. You can use it if the FPGA has been already programmed with the hardware contents.
`t32.cmm`	Main PRACTICE script, responsible for loading the JAM file and starting all the T32 instances on every CPU.
`testcase_data.cmm`	Internal file used for automatic testcase generation.
`t32/*`	Internal PRACTICE scripts. They are a copy of the files inside `components/evidence_ee/ee/pkg/cpu/nios2/debug/lauterbach/t32`.
`cpuname/config.t32`	Configuration file for T32. Contains the Multicore configuration information.
`cpuname/orti.men`	Trace32 menu automatically generated using the Lauterbach ORTI menu generator.
`cpuname/system.orti`	The ORTI file, for each cpu.
`cpuname/t32.cmm`	The main script file executed by each CPU.

Table 4.1: Files generated for the Lauterbach Trace32 support (*cpuname* is the name of the CPU as specified in the OIL file).

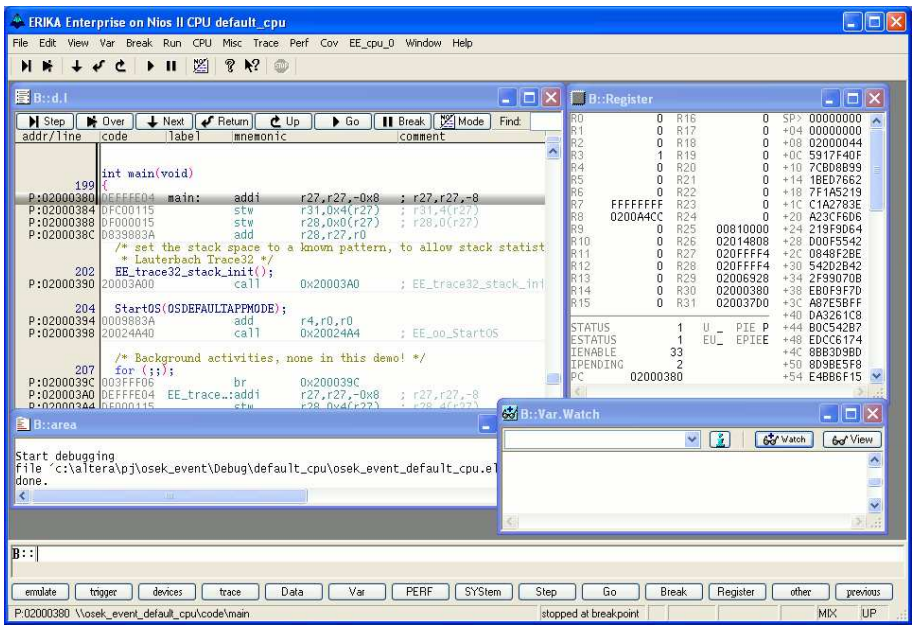

Figure 4.1: The Lauterbach Trace32 for Altera Nios II.

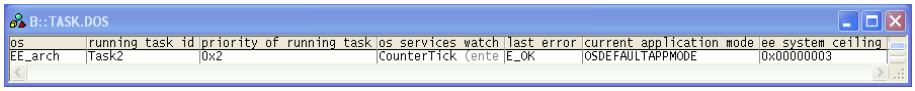

Figure 4.2: General information about the Erika Enterprise status.

- Figure 4.2 shows the general information about the kernel global variables, such as the name of the running task, the current priority of the running task, the last RTOS primitive called, the last error returned by an Erika Enterprise primitive, the current application mode and the current system ceiling.
- Figure 4.3 shows, for each task, the task name, its current priority (it may be different from the nominal priority when the task lock a resource), the task state, the task stack, and the current pending activations.
- Figure 4.4 shows, for each resource, the resource name, the resource status, the

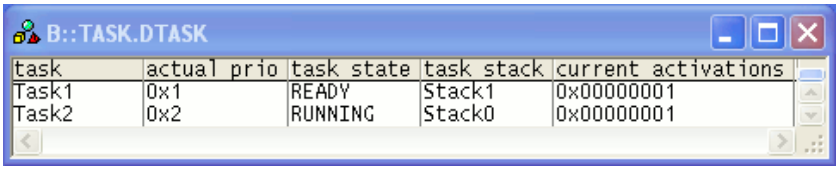

Figure 4.3: Information about the tasks in the system.

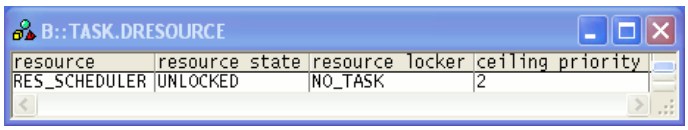

Figure 4.4: Information about the resources in the system.

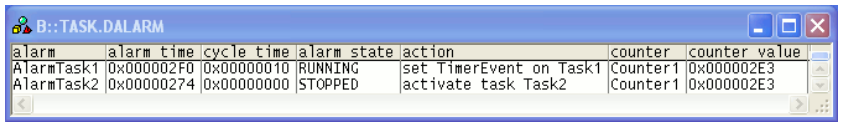

Figure 4.5: Information about the alarms in the system.

task that has locked the resource (if any), and the ceiling priority of the resource.

- Figure 4.5 shows, for each alarm in the system, the alarm name, the time to which the alarm will fire, the cycle time of the alarm (`0x0` means the alarm is not cyclic), the alarm state, the action linked to the alarm notification, the counter to which the alarm is attached, and its value.

- Finally, Figure 4.6 and Figure 4.7 show information about the stacks that has been configured in the application. In particular, the first figure shows the stack name, size, base address, direction, and fill pattern, while the second figure shows in a graphical way the current stack usage. Please remind that to obtain the graphical stack usage estimation the application has to call `EE_trace32_stack_init` at system startup. In this example, `Stack0` is the shared stack used by the background task (that is, the `main` function), and by `Task2`. `Stack1` is used by `Task1`, and `Stack2` is the interrupt stack.

The Erika Enterprise Trace32 support also includes support for the Nios II tracer module. As an example, Figure 4.8 shows the execution of an interrupt handler as recorded by the tracer module. Figure 4.9 shows an interpretation of the context changes and task status values using the ORTI information.

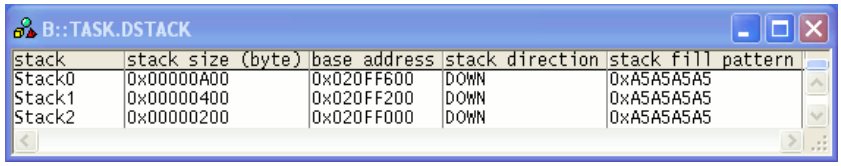

Figure 4.6: The application stack list.

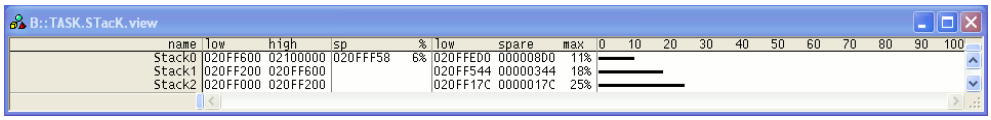

Figure 4.7: A graphical view of the application stack usage.

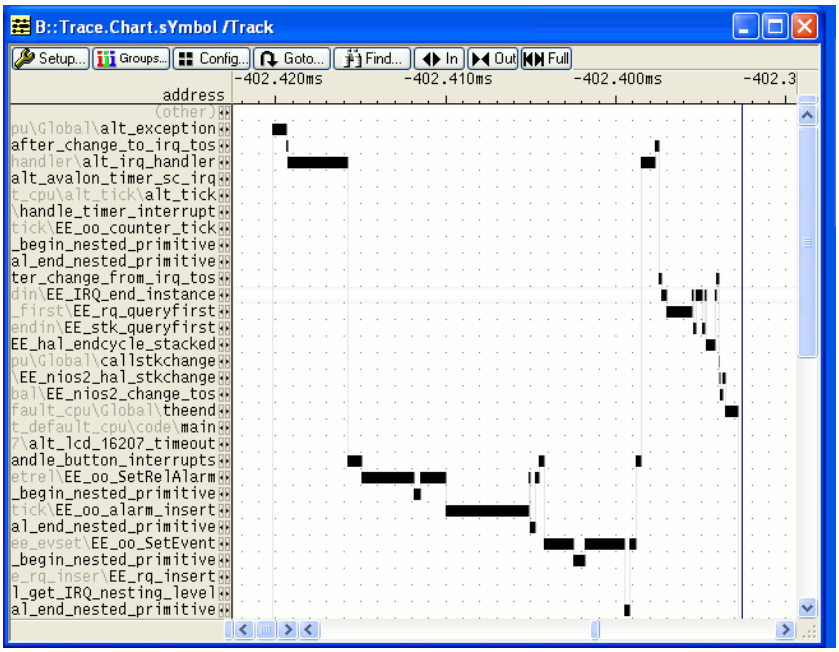

Figure 4.8: The execution of the Button IRQ as recorded by Lauterbach Trace32.

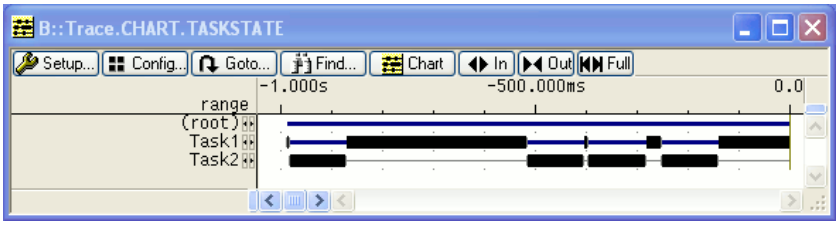

Figure 4.9: The interpretation of a trace recorded with Lauterbach Trace32 showing the context changes happened in the system.

4.3.1 Acknowledgements

We would like to thank Ing. Maurizio Menegotto from Lauterbach Italy Srl for his support in the integration of RT-Druid and Erika Enterprise with the Lauterbach Trace32 Debugger and Tracer.

5 History

Version	Comment
1.0.x	Initial revisions for Nios II 5.0 and 5.1.
1.0.3	Updated text, corrected typos.
1.0.4	Completed the introduction with some instructions about how to install and configure the environment. Corrected typos.
1.0.5	Corrected typos; added new versioning mechanism.
1.0.6	Updated for Nios II version 8.0.

Bibliography

[1] Lauterbach GMBH. The Lauterbach Trace32 Debugger for Nios II. http://www.lauterbach.com, 2005.