ERIKA Enterprise Manual for the Altera Nios II target

the multicore RTOS on FPGAs

version: 1.2.3
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 **5**
- 1.1 Erika Enterprise on multicores . . . 5
  - 1.1.1 Systems-On-Chip and FPGAs: the future? . . . 5
- 1.2 Solving problems when moving code to multicores . . . 6
  - 1.2.1 Design and programming. . . . 7
  - 1.2.2 Code placement and resource sharing. . . . 7
  - 1.2.3 Porting of legacy code to new architectures and to multicores. . . . 7
  - 1.2.4 Resource sharing and OS-level mechanisms. . . . 8
- 1.3 Multicore API implementation . . . 8
  - 1.3.1 Erika Enterprise and multicore hiding . . . 8
  - 1.3.2 Contention for multicore shared data structures. . . . 9
- 1.4 Erika Enterprise for the Altera NIOS II platform . . . 10

2 Single CPU designs **14**
- 2.1 Single core Design Flow . . . 14
- 2.2 System libraries . . . 14
- 2.3 Interrupts . . . 16
- 2.4 The StartOS function . . . 16
- 2.5 Altera Nios II IDE tips and tricks . . . 17
- 2.6 Altera Device Drivers . . . 18

3 Multicore designs **19**
- 3.1 Multicore Design Flow . . . 19
- 3.2 Basic notions . . . 20
- 3.3 Data scopes . . . 22
- 3.4 Memory types and their allocation . . . 25
- 3.5 Handling data consistency using mutual exclusion . . . 28
- 3.6 Startup barrier . . . 29
- 3.7 Interprocessor Interrupt . . . 32
- 3.8 Data caches and multicore data sharing . . . 34
  - 3.8.1 How to define cache-disabled multicore global data structures . . . 37
  - 3.8.2 How to define a data structure that is global to a single processor 39
  - 3.8.3 How to allocate a data structure inside a particular data memory 40
  - 3.8.4 Disabling GP addressing for global resource access . . . 40
- 3.9 Code sharing . . . 41
- 3.10 On-chip memories, .hex files and multicore systems . . . 42
- 3.11 Designing multicore software . . . 43

4 OIL Extensions for the Nios II target **45**
4.1 Task Extensions . . . 45
4.1.1 Stack size . . . 45
4.2 Nios II specific extensions . . . 45
4.2.1 Configuration parameters . . . 45
4.2.2 CPU data . . . 46
4.2.3 CPU data extensions for FRSH . . . 47
4.2.4 Mutex options . . . 48
4.2.5 Startup . . . 48
4.2.6 Intertask notifications . . . 49
4.2.7 Compiler options . . . 50
4.3 OIL Extensions for Multiprocessing . . . 50
4.3.1 Interprocessor Interrupts Extensions . . . 50

5 History **51**

1 Introduction

1.1 Erika Enterprise on multicores

Today's embedded systems are continuously being extended to support additional and more complex functionality. In many domains (automotive, telecommunications, domotics), availability of powerful hardware at low prices and cost/time market pressure are pushing for integration of functionality on system-on-chip devices capable of processing large amount of data in a short time.

Multicore systems are being considered as an economically viable alternative to support this increasing computational demand. In this context, Evidence proposes OS-level solutions and tools for embedded multicore-on-a-chip.

The Evidence RT-Druid/Erika Enterprise solution, originally developed for small-scale OSEK/VDX compatible embedded systems for the automotive powertrain market has been ported to the Altera Nios II environment, easing the adoption of reconfigurable FPGA capable of supporting multiple softcores on the same FPGA.

The RT-Druid/Erika Enterprise pair has been designed to handle multicore development and programming by hiding the use of multicore synchronization primitives. With multicore hiding, it is possible to seamlessly migrate application code from a single processor to multicore without changing a single line of the source code.

Hiding helps customers preserving their code base. Retargetting an application from single to multicore architectures only requires different OIL configurations, but allows retaining the same source code in all the configurations.

1.1.1 Systems-On-Chip and FPGAs: the future?

Trends and challenges from the automotive market. Electronic components in today's cars are mandatory elements to satisfy tight safety, efficiency and regulation constraints. Current vehicle electronics systems consist of up to 100 embedded controllers connected in a real-time distributed electronic system with several network clusters and electronic control units (ECU). Setup and Management of such complex distributed configurations is now the rapidly becoming the cost, performance and reliability bottleneck of the system. To satisfy tighter cost and packaging constraints the number of ECU should be reduced compared to today cars resulting in an integration of several functions.

To cope with this issue and the related design efficiency, the AUTOSAR [2] industrial initiative is standardizing a common set of hardware and software components and an accommodating platform that allows the integration of applications provided by different sub-system and component makers. This trend and the ever increasing application

complexity (e.g. x-by-wire) demand architectures with multiple computational units.

FPGAs: the opportunity. On the silicon technology side, the continuous scaling provides hardware fabrics that allows unprecedented integration of functions in a system-on-chip with very high parallelism. This trend promises that, in the very near future, even the use of reconfigurable hardware and softcores (CPU implemented with FPGAs) will be cost effective for selected applications.

Moreover, several CPU providers (Intel,Motorola, IBM, ARM, etc) have already announced multi-processor platforms with a winning cost/performance trade-off for the consumer and multimedia markets.

The Janus system, developed by ST Microelectronics/Magneti-Marelli Powertrain and Parades, is an example of a single-chip dual-processor platform for power train applications (featuring two 32-bit ARM processors and four memory banks connected by a crossbar switch) offering better cost/performance trade-offs than traditional architectures.

Reconfigurable platforms, such as Altera FPGAs, are providing soft-core solutions like Nios II, offering the opportunity for implementing 4 or more CPUs into even the cheapest Altera Cyclone II chips.

1.2 Solving problems when moving code to multicores

The development trends for next generation embedded devices clearly states that the future lies to multicore devices on the same chip. In particular, power management, leakage power issues, frequency limits, cost reduction and other factors are driving many companies toward the implementation of multicore system-on-a-chip solutions integrating together CPUs, memories, and peripherals.

Altera Nios II is an example of these multicore platforms. Thanks to the configurability of FPGAs, developers can design multicore in minutes. Although on one side hardware design is simplified by tools like Altera SOPCBuilder, on the other side application development is made more complex by the fact that the application code have to be spread out among different processors.

In particular, partitioning the application among the different CPUs, and implementing an efficient communication mechanism between the CPUs are one of the first issues to be addressed at the first stages of the development process. Unfortunately these choices are very critical for later development phases, since changes in the partitioning scheme can heavily change application design.

RT-Druid and Erika Enterprise give support to developers to solve the partitioning issues in multicore applications based on Altera Nios II, enabling the developer to perform code partitioning and then to easily change it at later stages in the design.

1.2.1 Design and programming.

Design and Programming paradigms must exploit the parallelism of these architectures, but programmers are usually not trained for writing code executing in parallel on multiple processors, and designers need to find the best trade-offs for exploiting computing capabilities without incurring in excessive blocking over shared resources or because of synchronization.

With Erika Enterprise, each task can be thought to run on a single processor multi threaded environment. Multicore issues like data cache disabling and mutual exclusion between different CPUs accessing concurrently the same data structures are handled automatically by RT-Druid and Erika Enterprise, simplifying the application design and verification.

1.2.2 Code placement and resource sharing.

The choice of which software has to be placed on which processor is usually called code placement or software binding or partitioning.

The task of the final user is to figure out the right communication pattern(s) / architecture(s) and to implement it. Today's approach used by many development tools is non conclusive, leaving out to the designer the job of choosing the best partitioning and communication scheme for its application, with the risk of making early decisions that may impact heavily on the application code with bad application performance.

In reality, changing the code placement must not impact on the way people design and program applications. The multicore structure must be hided to the user whenever possible, and source code compatibility between mono and multicore, with automatic mapping of application facilities to the new multicore features. This is exactly what the RT-Druid code generator offers, exploiting the multicore support provided by Erika Enterprise.

Future versions of RT-Druid will also enable users to perform well-reasoned partitioning choices integrating the results of timing analysis of the application with schedulability analysis to guide design choices.

1.2.3 Porting of legacy code to new architectures and to multicores.

One of the problems that typically arises when adopting multicores is that application rarely start from scratch with a multicore approach. More often, new applications are upgrades to existing (working) systems that are adapted to particular new environment.

RT-Druid and Erika Enterprise helps porting existing legacy code to multicores because:

- The kernel API is the same for both single and multicore systems.
- A multicore system can be view as an "extension" of a single core. You can add functionality to a system on a separate core, without perturbating too much the application running on the first CPU.

- Application code does not have to be changed when changing the partitioning scheme.

1.2.4 Resource sharing and OS-level mechanisms.

When moving from single processor systems to multicore systems, standard programming paradigms used in real-time systems to access shared resources does not work anymore.

In particular, all the solutions used to avoid the Priority Inversion problem, such as the Immediate Priority Ceiling protocol implemented in the OSEK/VDX standard and in other APIs does not scale to multicores like Altera Nios II.

RT-Druid and Erika Enterprise offer an innovative way to automatically provide support for resource sharing among different processors, automatically integrating resource consistency protocols for multicores when a resource is shared among different CPUs.

1.3 Multicore API implementation

While hardware technology is offering the opportunity for greatly increased performance, conventional RTOS implementations and standards are not yet ready to support multicore architectures. The OSEK/VDX standard for Real-Time Operating Systems has gone a long way in introducing mechanisms for ensuring predictable (schedulable) real-time behavior and has provided the developers of automotive applications with a stable and sound API.

Unfortunately, OSEK/VDX was not meant to be and it is not ready to cope with the challenges posed by a multicore system. In particular, OSEK/VDX fits best on systems where different processors are physically separated and connected through a network (e.g., Flexray, CAN, LIN, etc.), but it does not best fit in systems that implement multicores on the same ECU.

In detail, the OIL configuration language supports the definition of single processor systems possibly communicating using OSEK COM, but provides no support for fundamental multicore issues such as: distribution of the computational activities (tasks) among processors; multicore communication and synchronization; and multicore details hiding for applications compilation and execution.

1.3.1 Erika Enterprise and multicore hiding

With multicore hiding - that is the possibility to seamlessly migrate application code from a single processor to multicore without changing the source code - the programmer's view of the application is not compromised, meaning that the RTOS API is supported without changes and only minimal OIL extensions. Furthermore, Erika Enterprise extends the real-time and memory saving features of automotive RTOS to multicore systems.

In detail, Erika Enterprise provides:

- Innovative mechanisms that allow exploiting the main features of Immediate Priority Ceiling in multicore systems.
- Predictable real-time behavior.
- Kernel features and OIL extensions that allow seamless execution of single processor code in multiple processor architectures.
- Tools that support task placement on processors; easy reconfiguration of kernel calls for resource sharing and communication mechanisms from intraprocessor to interprocessor usage.
- Minimal footprint in terms of ROM and RAM usage.
- Real-time performance (such as interrupt and scheduler latencies, and context switch times) in line with the best market options and the possibility of configuring the kernel for constant time O(1) scheduling.

Erika Enterprise hides the multiple processor structure by providing an automatic mapping of the RTOS API calls to the single processor or the multiple processor implementation, according to the target of the call. This mapping is obtained by exploiting an enhanced remote notification mechanism based on multicore interrupts. Remote notification is used to provide an implementation to kernel primitives acting on tasks (resources) allocated to remote processors. Moreover, the notification mechanism is hidden inside the implementation of the Erika Enterpriseprimitives, maintaining the same interface to the developer both for single and for multicore systems.

The additional overhead caused by the remote execution of calls is mainly due to the "send" of the remote notification from one side, and to the interprocessor interrupt that is raised on the receiving side. Most of the overhead of the two calls is due to the internal usage of the Altera Avalon Mutex calls. However, this overhead does not have a great impact on typical applications where remote activations are not as frequent as local activations.

1.3.2 Contention for multicore shared data structures.

In addition to multicore hiding, the mechanisms typically provided by common RTOS for sharing resources with a predictable and bounded blocking time, such as the Immediate Priority Ceiling protocol and the often related mechanisms for sharing stack space among application tasks lose some of their hard real-time properties (such as predictable blocking times).

Erika Enterprise implements an advanced protocol used when sharing resources among tasks allocated to different processors. The protocol, known as Multicore Stack Resource Policy (MSRP), makes internal use of spin-locks among different processors to ensure data consistency.

We expect that contention for multicore shared resources among different systems will be the main source of overhead in the system. To reduce them, critical section for

multicore shared data structures should be reduced as much as possible, for example making local copies of the data structures when possible.

1.4 Erika Enterprise for the Altera NIOS II platform

This document describes the details of the Erika Enterprise porting for the Altera Nios II platform. Altera Nios II is a powerful soft-core processor, which is basically a 32-bit RISC microcontroller especially designed for the Altera FPGAs which can be customized to meet high performance or low hardware (in terms of Logic Elements) requirements. The Altera Nios II core can also be highly customized adding custom peripherals, custom assembler instructions, debug features, caches, and other features.

As you will note by reading the following pages, Erika Enterprise fully supports the various features of the Nios II platform, maintaining the Altera workflow based on Quartus II, SOPCBuilder, and Nios II IDE.

Erika Enterprise runs on every typical single core Nios II design without requiring any modification to the hardware. All the Nios II HAL functions, drivers, and features are retained on a Erika Enterprise system.

On the other hand, Erika Enterprise shows its performance and its ease of use when the designer moves from simple single CPU Nios II designs to more complex multicore Nios Ii designs. In particular, Erika Enterprise has been designed to address in a simple way all the problems that can arise when dealing with multicore systems.

Designing multicore System-on-a-chip (SOC) has always been a difficult task, because it involves a deep knowledge of hardware details and application details that are needed to perform a correct co-design between the application parts that have to be implemented in hardware, and those that have to be implemented in software. Moreover, in a multicore system, the user have to understand how the various software functionality maps to the available computational resources, in a way to optimize the system performance. Choosing the right partitioning scheme for a multicore system is one of the most difficult tasks, because a wrong application partition heavily impact on application performance and response times. In fact, tuning a multicore design is usually made in various design steps. Each design step in fact refers to different stages in the application design, and often the result is a tradeoff with the design choices that were made earlier in the application definition.

The design tools provided by Altera allow designers to easily create and test multicore hardware platforms based on the Altera Nios II Processor. Basically the developer creates a multicore platform by simply inserting CPUs and connecting peripherals using the Altera SOPCBuilder Tool. Using this tool the designer can modify the hardware architecture easily, adding and removing CPUs, modifying the bus connections, and so on. Once the hardware platform has been created and generated, the developer can develop the software separately for each CPU using the Nios II IDE based on Eclipse. Altera provides a nice environment where for each CPU the user can instantiate the device drivers of the hardware components connected to the particular CPU (using the provided Altera HAL and the System Libraries Projects). After that, the user can create

a C/C++ application that is linked to a single System Library (that is, to a single CPU).

It is our believe that the common workflow during the design of complex multicore applications requires a frequent number of change in the partitioning of the software among the different CPUs. That is typically needed because application requirements often change during the development of an embedded system, and the choices that has been made earlier in the design may not be the right ones at the end of the development. In this sense, the developer must have the power to change the software partitioning among processors with little effort, as it is currently done for the hardware part in SOPCBuilder, where the user can change the hardware configurations with a few mouse clicks.

That is exactly what Evidence Srl aims to provide to Altera Nios II customers: a way to simply deal with the complexity of multicore software development, easing the partitioning of applications on different processors, giving the possibility to developers to change their partitioning without changing their source code.

Briefly, Evidence Srl provides a powerful design and configuration tool named RT-Druid, and an RTOS, named Erika Enterprise. The RT-Druid Toolset and Erika Enterprise RTOS exploit the multicore capabilities of the Altera FPGA boards and allow to fully use the power of the multicore designs.

On the host side, RT-Druid offers an integrated view of a multicore application under the Nios II IDE, allowing the user to partition the application jobs on the different available CPU. To do that, RT-Druid offers a concept of "application" project that spans different CPUs, allowing an easy move of features across the execution units. RT-Druid is a set of plugins for the Eclipse framework that are integrated with the Altera Nios II IDE, giving the possibility to the user to:

- develop a multicore application using an RT-Druid project, that is an Eclipse Project that is able to handle the software for all the CPUs in the multicore system[1];
- specify one or more partitioning schemes to be used for a multicore system;
- generate the configuration code for the Erika Enterprise RTOS;
- analyze the results of the partitioning using schedulability analysis [2];

On the target side, Erika Enterprise offers a multiprogramming environment that can be used to deploy the application architecture defined in RT-Druid on the real hardware. Erika Enterprise helps the partitioning job supporting code deployment that is independent from the particular CPU, allowing the use to move code between CPUs without changing it. Erika Enterprise is a Real-Time Operating System (RTOS) that offers a multicore partitioned multithread environment that internally takes care of multicore issues like activation and run of remote tasks, periodic alarms, and shared resources among

[1] A single RT-Druid Project can be viewed as a replacement for the set of Altera Application Projects for each CPU.

[2] The schedulability analysis plugin will be available together with the next version of RT-Druid

tasks allocated to different processors. Of course, as said before, the restriction to single processor designs is supported. Erika Enterprise currently offers support for fixed priority scheduling, with preemptive, non preemptive and mixed preemption support.

Erika Enterprise is ready for formal compliancy with the OSEK/VDX operating system standard.

The main feature in the Evidence approach for Nios II is the fact that the RT-Druid Toolset and Erika Enterprise do not change the current Altera Workflow. For that reason, users accustomed with the Nios II IDE editing and debugging features can easily reuse their knowledge.

Some of the features supported by RT-Druid and Erika Enterprise are:

- Support for the definition, and deployment of software for Nios II multicore systems with shared memory.
- Support for partitioning of the application functionalities on different processors with negligible modification to the source code base.
- Support for easy change of the partitioning scheme allowing the developer to easily try and test different partitioning schemes.
- Support for the migration of single processor source code to multicore systems.
- Support for the Altera development toolchain: RT-Druid and Erika Enterprise are perfectly integrated with Altera Nios II, reducing the learning curve of the tools.

Specific Altera integration features include:

- Integrated as a component in the Altera HAL;
- Compatible with the Altera HAL peripheral drivers and system libraries;
- Support for nested interrupts;
- Support for a RTOS configuration code generator compliant with the OSEK OIL specifications;

Moreover, the multicore support of Erika Enterprise includes:

- Advanced software partitioning support:
  - developers can decide which task goes on which processor;
  - developers can move tasks between processors without changing the source code;
  - Transparent handling of shared resource locks behavior depending on the partitioning of the application;
- Interprocessor interrupt support;

- Shared resource support using queuing spin locks on top of the Altera Mutex Peripheral;
- Automatic cache disabling technique without changes to the user source code (only changes to data definitions);
- Support for multicore scheduling algorithms with bounded blocking times on multicore;

The following chapters describe in detail the single and multicore support for the Altera Nios II platform that is offered by Erika Enterprise and by RT-Druid, highlighting the various features available and the main architecture design requirements of the system.

2 Single CPU designs

The purpose of this chapter is to describe in detail the single CPU design support that is offered by Erika Enterprise and by RT-Druid, highlighting the various features available to the Altera developers.

2.1 Single core Design Flow

The purpose of this section is to shortly describe the design flow that must be used to develop a single core system using the tools provided by Altera and by Evidence Srl. As it can be seen, the design flow is equivalent to the current Altera design flow.

The first step of a Nios II single core system is the hardware design. Every hardware design which is able to run the Altera HAL can also run Erika Enterprise. For example, the `standard`, `full_featured`, ... designs provided by Altera for the various evaluation boards can be used as a starting point by the designer.

Once the hardware design has been defined, the designer can develop the application software using Altera Nios II IDE (based on the Eclipse Framework).

Using Nios II IDE the designer have to instantiate an Altera System Library Project for each CPU in the system. The Altera System Library contains the device drivers for the peripherals that are actually connected to the CPU the library refers to. Please refer to the Altera documentation on how to create a System Library.

After creating an Altera System Library for each CPU, the designer can create an RT-Druid Project. The RT-Druid Project contains the application that will run on the Nios II processor, and it is similar to the "C/C++ Application" project provided by Altera. As it will be described later, the only difference with the Altera project is that the configuration of the system is defined inside a text file which follows the OSEK OIL specification [1]. Figure 2.1 and 2.2 graphically shows the difference between the Altera and the Evidence approach: the Altera HAL approach provides a C/C++ application project for each CPU, whereas Erika Enterprise provides a RT-Druid project. RT-Druid Projects are described in Section 3.2, and in the RT-Druid Reference Manual.

Then, the designer compiles the application to create a set of compiled files that are equivalent to the files generated by the normal Altera Projects makefiles. As a result, the developer can debug the application in the same way as all the other Altera HAL applications.

2.2 Altera System libraries

Erika Enterprise uses the same System Libraries that are provided together with the

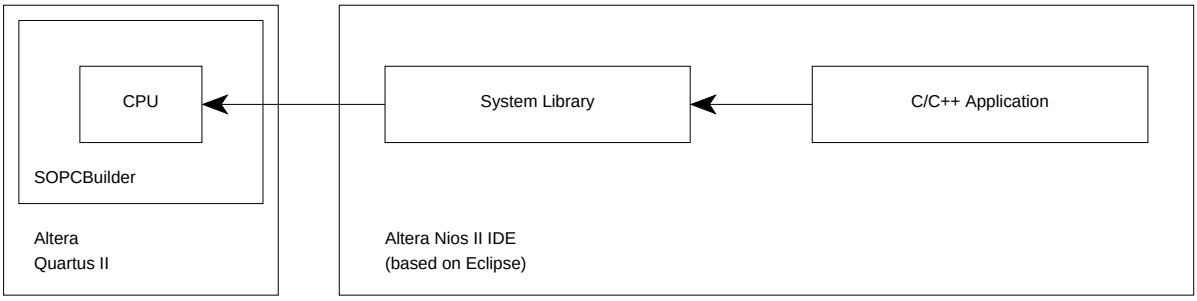

Figure 2.1: This Figure displays the Altera approach to software development for single core Nios II designs. Each Application is linked to a System Library which contains the peripherals instantiated in SOPCBuilder for the CPU.

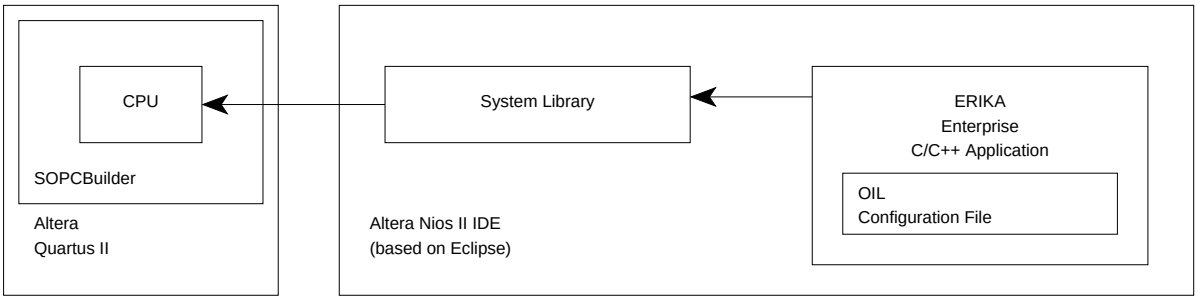

Figure 2.2: This Figure displays the Evidence Srl approach to software development for single core Nios II designs. Each RT-Druid Project contains an OIL configuration file that specifies the application and kernel configuration. The RT-Druid Project is linked to a System Library which contains the peripherals instantiated in SOPCBuilder for the CPU.

Altera HAL. When using the Altera System Libraries, developers have to take care of the following issues:

- One System Library has to be created for each CPU in the system.
- Separate stack space for exceptions should not be set inside the System Library properties. A similar feature can be obtained by setting a separate interrupt stack inside the OIL configuration file.
- Custom linker scripts are supported with Erika Enterprise. Custom linker scripts can be specified inside the System Library project properties.

2.3 Using Interrupt handlers with Erika Enterprise

Users of the Altera HAL can continue to use the Altera provided functions for interrupt handling. In particular, `alt_irq_register()` must be used to provide the registration of an interrupt handler. `alt_irq_interruptible()` and `alt_irq_noninterrtptible()` must be used to handle interrupt nesting.

The Nios II Erika Enterprise implementation considers all the interrupt handlers as interrupts of category 2. Erika Enterprise correctly handles interrupt nesting, performing appropriate checks at the end of the last interrupt to implement task preemption. There is no support for Interrupts of Category 1 in Nios II because of the internal structure of the Nios II processor.

Among the options available on the Altera System Library Properties, there is one named *Use a separate exception stack*, that can be used to allow interrupts and exceptions stack to be allocated in separate memories. This feature is typically used to move the interrupt stack in faster memories, such as on-chip tightly coupled memories. When developing applications using Erika Enterprise, the separate stack as specified in the Altera System Library cannot be used. A similar feature can be obtained with a multistack configuration of Erika Enterprise.

2.4 The StartOS function

To start the multiprogramming environment the designer have to call the `StartOS()` function inside `main()` or `alt_main()`.

The `StartOS()` primitive does the following actions:

1. It registers the interprocessor interrupt routine and initializes the Altera Mutex Peripheral.
2. It implements the startup barrier (only on a multicore system).
3. It calls StartupHook.
4. It activates autostart tasks and alarms.

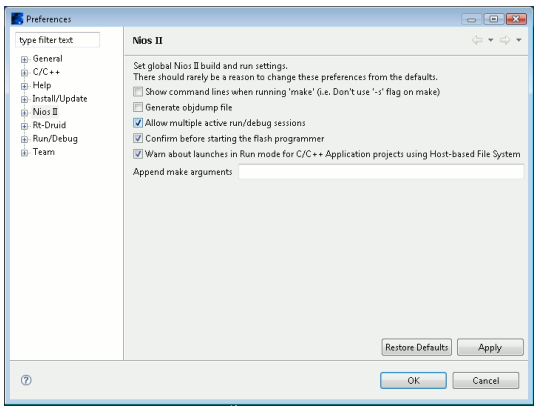

Figure 2.3: Nios 2 IDE preference window. The "Allow multiple runs" checkbox must be checked.

5. It checks for preemption.

6. It returns to the caller.

Please note that `StartOS()` returns to the caller only when the system have an idle time, that is, when all the task activations has run.

The code literally after `StartOS()` must correspond to background activities that does not use Erika Enterprise services.

If unsure of the code that have to be put after the call to `StartOS()`, use the following forever loop:

```
for (;;);
```

2.5 Altera Nios II IDE tips and tricks

This is a list of the preferences of the Altera Nios II IDE that needs a modification:

Allow multiple active runs. This option can be found in "window/preferences/nios2/ allow multiple active runs". You need to set this option to be able to debug multicore systems (see Figure 2.3)

Build if required This option can be found in "window/preferences/rundebug/launching/build if required". This option can be used disable the application automatic build (see Figure 2.4. Since building a multicore system may require a lot of time, this options waiting time when avoids automatic building when the user has to start different debug sessions without need of recompiling the system.

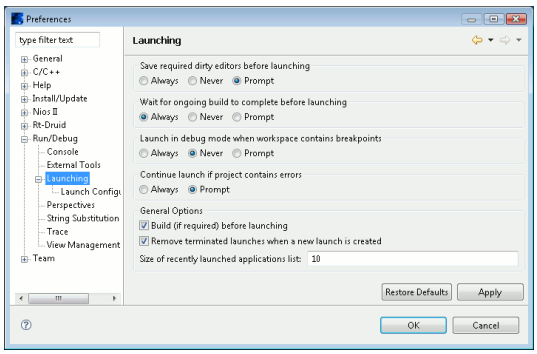

Figure 2.4: Nios 2 IDE preference Run/Debug window. Consider unchecking the "Build if Required" checkbox.

2.6 Usage of the Altera HAL device drivers with Erika Enterprise

Erika Enterprise currently does not provide any customized versions of the Altera HAL device drivers.

In general, the Altera HAL device drivers have to be used with care in multitask application developed with Erika Enterprise, because these device drivers are designed for a single task environment.

In particular, some care have to be taken avoiding that more than one task calls a device driver set of primitives concurrently. There are two possibilities:

- Allocate a dedicated task that is the only one that uses a given device driver primitive (for example, if a CPU has a JTAG UART, the user should allocate a single task that is responsible for the standard input/standard output operations on that peripheral).
- Allow different tasks to use the primitives in a mutual exclusive way. In that case, mutual exclusion can be obtained using a Resource dedicated to the purpose, and using GetResource and ReleaseResource primitives to sequentialize the accesses to the shared resource.

In general, all the file system primitives that are potentially blocking on a system are implemented with a busy wait within the HAL. For that reason, use these primitives in non-blocking mode. Future versions of Erika Enterprise will include a customization of the Altera HAL drivers to allow file system primitives to become blocking primitives.

3 Multicore designs

The purpose of this chapter is to describe in detail the multicore support that is offered by Erika Enterprise and by RT-Druid, highlighting the various features available.

3.1 Multicore Design Flow

The purpose of this section is to shortly describe the design flow that must be used to develop a multicore system using the tools provided by Altera and by Evidence Srl. As it can be seen, the design flow is equivalent to the current Altera design flow, with some additional points that are needed to create Erika Enterprise compatible hardware and software.

The first step of a Nios II multicore system is the multicore hardware. A multicore hardware can be obtained easily using Altera SOPCBuilder, as described in the document named "The Multiprocessor Nios II Systems Tutorial" [3] that describes the basics of designing a multicore Nios II system. Please refer to that document for the basic informations on how to create a generic Nios II multicore hardware.

Multicore designs compatible with Erika Enterprise can be built starting from the standard peripherals provided by Altera. For a multicore design to be compatible with Erika Enterprise, special care have to be taken when instantiating particular Altera peripherals and memories. Please refer to the following sections for the guidelines to be followed when adding the following hardware features:

- Interprocessor Interrupt (see Section 3.7);
- Altera Avalon Mutex (see Section 3.5);
- Memories (see Section 3.4);

A step-by-step description on how to create a multicore hardware is also available inside the "ERIKA Enterprise Multicore Tutorial for the Nios II hardware", available for download at the Evidence web site.

Once the hardware design has been defined, the designer can develop the application software using Altera Nios II IDE (based on the Eclipse Framework).

For each CPU in the system, the designerhave to instantiate an Altera System Library Project. Please refer to "The Multiprocessor Nios II Systems Tutorial" [3] for details.

After creating an Altera System Library for each CPU, the designer can create an RT-Druid Project. The RT-Druid Project contains the application that will run on *all* processors of the system. In a way similar to the single CPU designs for Erika Enterprise, the RT-Druid Project contains an OIL configuration file. The OIL file [1] specifies the

partitioning scheme adopted by a particular application. RT-Druid Projects are described in Section 3.2.

Next, the designer compiles the application to create a set of compiled files that are equivalent to the files generated by the normal Altera Projects makefiles. As a result, the developer can debug the multicore application as usual using an Altera Multiprocessor Collection. Please refer to "The Multiprocessor Nios II Systems Tutorial" [3] for details.

3.2 Basic notions for multicore programming

This Section is a simple list of definitions and general terms that highlight characteristics of the systems that can be designed using RT-Druid and Erika Enterprise.

A single SOPCBuilder Block. Multicore systems handled by Erika Enterprise are defined inside a single SOPCBuilder block. This is somehow similar to the approach followed by "The Multiprocessor Nios II Systems Tutorial" [3].

Partitioning of tasks and data on multicores. When developing a multicore system, the user have to think as the application as a whole. The application will be composed by a set of tasks, shared resources, that will work cooperatively to reach the application goals. The approach chosen by Erika Enterprise is a so called *partitioning* approach, that is, each concurrent task is statically linked at build time to a given CPU. Each CPU includes a separate copy of the operating system and of the device drivers that are present on the particular CPU.

Tasks partitioning into CPUs can be done at the end of the application design: the configuration system is in fact designed in a way to allow the designer to write partitioning-independent code; then, the developer can change the CPU partitions without changing the application source code, giving the possibility to really exploit the power of Nios II multicores.

The minimal partitioning item in a system is the source file. That is, the code of tasks allocated to different CPUs are typically put by designers on different files, to allow an easy partitioning in different CPU in a later stage of the development.

CPUs are not identical. There are various differences between the different CPUs in a SOPCBuilder Block. First of all, they may come from different versions of the Nios II core, with possibly different custom instructions. Moreover, Nios II peripherals are typically connected to a single CPU[1], and that means that each CPU will have a different set of interrupts, memories, and software device drivers.

The Master CPU. There is a CPU[2] which plays an important role in the system. That CPU is usually called *Master CPU* and is referred in the source code as `MASTER_CPU`.

[1] With the exception of the Altera Avalon Mutex Peripheral

[2] With Nios II 5.0 and 5.1, it was the CPU with CPUID equal to 0, that wast typically the first CPU in the SOPCBuilder CPU list.

In multicore systems supported by Erika Enterprise, the Master CPU plays an important role, because it acts as the CPU that have to initialize the shared data (see Section 3.4 and, if configured, the startup barrier (see Section 3.6).

The Master CPU is specified in the OIL configuration file as described in the following example, where `cpu0` is specified as the Master CPU:

```
CPU test_application {
  OS EE {
    ...
    MASTER_CPU = "cpu0";
    CPU_DATA = NIOSII {
      ID = "cpu0";
      ...
    };
  ...
};
```

Altera mutexes. The multicore system must include at least an Altera Avalon Mutex peripheral. That peripheral will be handled internally by Erika Enterprise to guarantee mutual exclusion between different activities residing on different CPUs (see Section 3.5). The application does not use directly the mutex peripheral. If a startup barrier is configured, a proper initialization value for the mutex have to be selected (see Section 3.6).

Multicore Interrupts. The multicore system must include a Multicore Interrupt feature. When developing a multicore application, the designer have only to think at the existence of concurrent tasks that will be activated and scheduled on the different CPUs. Erika Enterprise hides the implementation details to the developer, and the developer does not have to handle the multicore communication issues. Internally, Erika Enterprise uses a Multicore Interrupt hardware support, that is used to dispatch various information between the various CPUs. The instantiation of an Interprocessor Interrupt peripheral is done connecting together a set of Altera Avalon PIO components in SOPCBuilder (see Section 3.7 on how to instantiate a proper Multicore Interrupt controller).

Shared Memories. A multicore system typically includes some shared memory (that can be on-chip or off-chip). Shared memories will be used to exchange informations between the different CPUs. The main idea is that the designer has the option of defining data structures that are global and visible to all cores. Moreover, all the CPUs will access the data structures using an automatic cache disabling technique without changing the application source code[3]. This kind of shared data structures will be defined inside the Master CPU.

Moreover, the designer can define a shared data structure leaving RT-Druid the choice if that data structure should be local to a CPU or global to all the CPUs in the system.

[3]Please remember that Altera Nios II does not have a cache coherency mechanism.

The rationale behind this is that local shared data structures are more efficient than global data structures, and also the fact that a shared data structure must be local to a CPU or global depends on the partitioning that the designer defines for the application. Leaving the choice to RT-Druid allows designer to write partitioning independent code. See Section 3.4 for details.

RT-Druid **project and Altera System Libraries.** The Nios II IDE project layout for an Erika Enterprise Application slightly differ from the Altera counterpart. In particular, the user has to instantiate a Nios II IDE system library for each CPU. Erika Enterprise allows users to reuse all their preexisting source code developed for the Altera HAL, adding multiprogramming support to ease the partitioning job and to fully utilize the multicore power available with Altera Nios II. Then, the user application is set up inside a RT-Druid Project. An RT-Druid Project is similar to an Altera C/C++ Application Project, except that an RT-Druid Project includes the application software for all the CPUs. The partitioning of the application tasks to processors is specified using a configuration file written following the OSEK OIL Standard [1]. The result of the compilation is an ELF file for each CPU, that can be processed in the same way as the ones produced by normal Altera Nios II Projects. When compiling an RT-Druid project, Figure 3.1 and 3.2 graphically shows the differences between the various versions: the Altera HAL approach provides a C/C++ application project for each CPU, whereas Erika Enterprise has only one project (the RT-Druid project) for all the CPUs. Please refer to the RT-Druid manual on how to create a new RT-Druid Project.

3.3 Data scopes in an Erika Enterprise multicore system

Data allocation in multicore systems based on the Nios II processor are a little bit different than conventional single processor architectures. In particular, a C data definition may be classified in one of the following points.

Automatic data. These data are allocated on the current stack by a function call when the function is called. The scope of these data is the function itself. Example:

```
void foobar(void)
{
        int this_is_automatic;
        ...
}
```

Static data. Static data is a C global variable that is defined as `static`. Only the source code inside a file that define the static variable can refer to the particular symbol.

Example:

```
static int this_is_static;
```

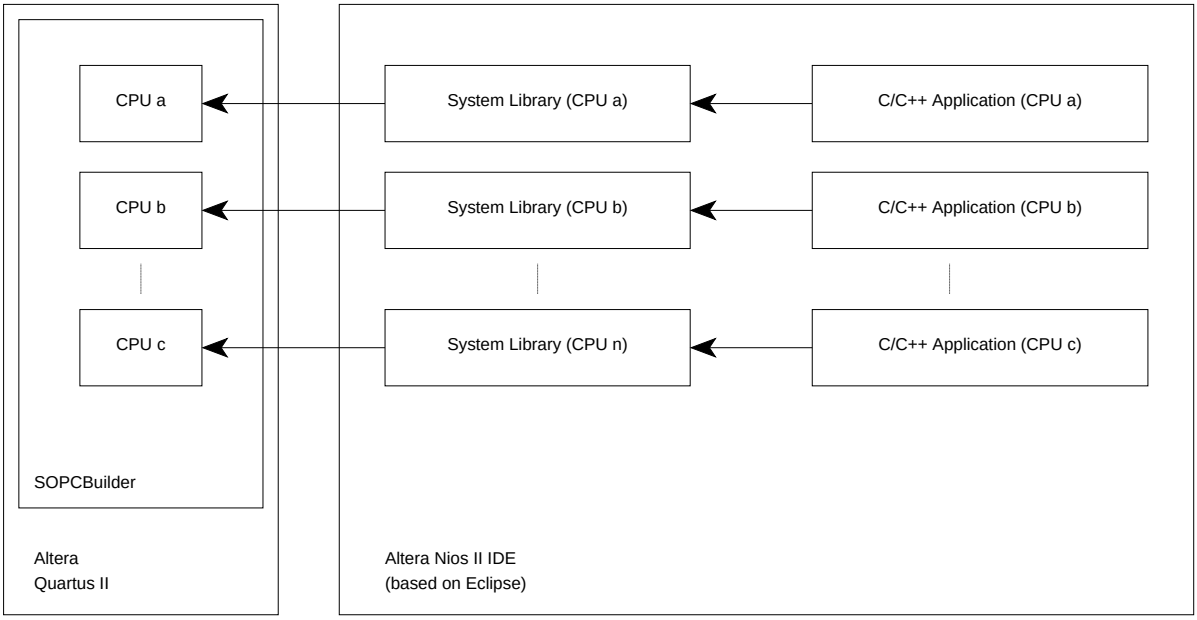

Figure 3.1: This Figure displays the Altera approach to software development with multicore Nios II designs. In particular, there is one Application for each CPU. each application is linked to a System Library project, which is linked to the CPU instantiated in SOPCBuilder.

Global data. Global data are C global symbols. The scope of these symbols is the entire source code that is linked inside an ELF file. That is, global data is **not** shared among different CPUs. Global data is also called *Local data* (to emphasize the fact that they are allocated to a single CPU only).

```
int this_is_global;
```

Heap. Heap data structure are region of memory that are allocated dynamically using library functions such as `malloc()` and `free()`. Typically, each processor has its own private heap memory pool, that each processor handles independently using a local copy of the above mentioned Standard C library functions.

Application data structures allocated in the heap typically are local to each processor and should not be shared among different processors. Erika Enterprise does not give any explicit support to share a heap variable (typically a pointer) between tasks allocated to different processors. To do that, the developer must ensure that:

- the heap memory resides on a memory zone accessible by the different processors that access it;
- proper cache disabling strategies are used to ensure that each processor uses the data consistently;

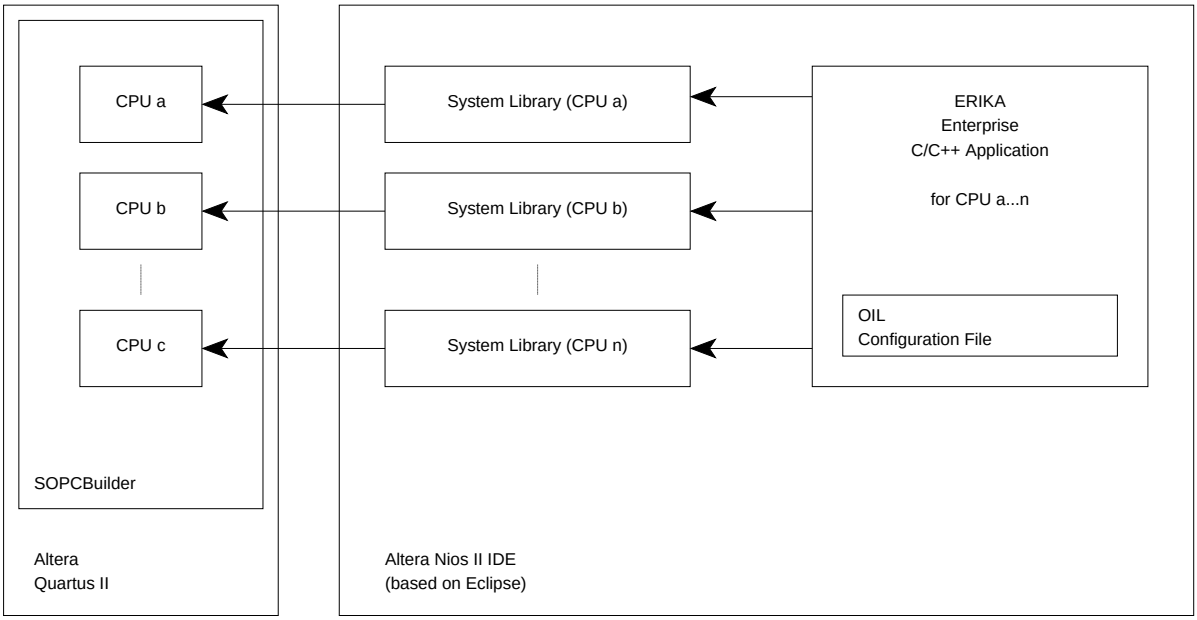

Figure 3.2: This Figure displays the Evidence Srl approach to software development with Nios II IDE. In particular, there is only a single application for every CPU, contained inside the RT-Druid Project. The RT-Druid project contains the Application partitioning and the kernel configuration inside the OIL file. In this sense, the single CPU case showed in Figure 2.2 is a special case of this Figure. Then, each CPU has its own System Library, which is linked to the CPU instantiated in SPCBuilder.

- the memory is allocated and freed on the same processor (that is, the developer cannot allocate a data structure on CPU a and free it on CPU b, because it would have called two different independent allocators, the allocator on CPU a, and the allocator on CPU b).

Multicore global data. Multicore global data are data structures that are visible to **all** CPUs in the system. This kind of data structure does not exist on a single processor system. These data structures are typically defined and initialized inside the Master CPU, and the are accessible to all the other CPU.

The Multicore Global data must be accessible to all the CPUs, that is, it must reside on a memory accessible to all the CPUs. As an alternative, the user can let the memory be shared among the CPUs that really uses the shared memory, plus the Master CPU (This may help saving some LE reducing the arbitration logic). Multicore global data not shared by the Master CPU is not supported in this release of Erika Enterprise (but will probably be supported in future releases).

When a CPU access a multicore global data, the data cache, if present, is automatically disabled[4].

Stack. The stack memory is used to store function calls, return address, and parameters. Each task in Erika Enterprise has a stack, that may be shared with other threads to reduce the overall stack space required by an application.

Basically, each CPU has stack called "shared", that is a fixed size amount that can be specified in the OIL configuration file. The shared stack has its top at the top of stack specified for each CPU inside the Altera System Library. That is, the main() function is always run on the shared stack (that is the same behavior of normal Altera HAL applications.

Additionally, each task in the system can have a private stack. A private stack is always needed if the task use blocking primitives like, for example, WaitEvent. Private stacks are allocated on the same CPU where the task is allocated.

3.4 Memory types and their allocation

Altera SOPCBuilder provides the user different kinds of memory types, ranging from onchip tightly coupled memories, onchip memories, and external SRAM, SDRAM and Flash memories.

All these different kind of memories can be used to store data and code. To let the content of a memory being shared among tasks allocated to different processors, the the memory slave ports must be connected to all the CPUs that access them.

In general, we will consider a system composed by a set of CPUs that have some "private" memory components (connected to that CPU only), and a set of "shared memory" components (connected to all the CPUs).

[4]Without the need of changing the source code, using bit-31 cache disabling.

In the following, we will consider that the shared memories are connected to all the available CPUs. Having memories connected to all the CPUs has the following drawbacks:

- If a CPU does not need any data structure allocated on a particular memory component, the connection logic used to connect that CPU to the memory component is wasted. In any case, unused arbitration logic can be removed in the final product if needed.
- Having a fully connected memory helps partitioning tasks to different processors without changing the hardware. That is especially helpful at design phase, where task partitioning is not definitive.
- Fully connected memories, allows a proper handling of the system startup. The idea is that the Master CPU have a role of "container" and "allocator" of all the shared data structures in the system. All the definitions of the shared data structures are done in the Master CPU. Having a single CPU responsible for the allocation of the shared data structures simplifies the system architecture because the linker takes care of allocating shared data structures to separate addresses.

  In this way, Erika Enterprise frees the designer the need to manually allocate shared data structures to separate memory zones, that at the end forces the developer to manually implement the linker job.

Using fully connected memories has the additional advantage of automatically adapting to partitioning changes, because inserting new shared data structures (for example because the designer splits a task in two parts allocated to different processors), or removing existing multicore shared data structures (because using proper partitioning data structures that were multicore global may become local when all the tasks accessing it are allocated on the same CPU) is handled *automatically* by RT-Druid and Erika Enterprise, without requiring any kind of manual configuration or, worse, hardware change.

Please note that whether for simple applications partitioning code and data structures is a human manageable task, when the application size grows the complexity of the configuration may be source of human errors and inefficiencies, especially if the development requires partitioning changes during system development.

The approach proposed by RT-Druid and Erika Enterprise allows a simple way to manage the partitioning problem, helping the designer concentrating on the application architecture instead of concentrating on the communication support of the application. After the application is developed, the partitioning will also influence the kind of communication pattern used to access the data structures. All that is done only at the software level, without modifying the hardware.

Memory allocation for the application data structures works as follows:

- The developer defines its data structure. See Section 3.8 on how to define a multicore shared data structure; see Section 3.8.4 about a discussion on GP-relative addressing for multicore global data.

- The linker allocates the data structures into memory sections, that are then located inside the memory components available in the hardware. The available memory sections are typically found in the Altera generated linker script in the System Library of each CPU[5].

- The only requirement that have to be fulfilled in this phase is that multicore global data structures must be allocated in memories that are connected to the various CPUs that may access the data. Since memories offsets in the address space are common to all the CPUs, pointers and addresses can be shared freely.

  There is no restriction in the allocation of local variables (except the obvious one that variables local to a CPU must be allocated in a memory space that is accessible by that CPU). Of course, local variables should be allocated in preference to private memories.

  Note: when the RT-Druid Code Generator generates the Erika Enterprise configuration code for a multicore system, some multicore shared data structures are generated. By default, these data structures are allocated inside the `.rodata`, `.sdata` and `.data` section of the Master CPU. As an alternative, a different memory region for the code generated by RT-Druid can be selected by specyfying the `MP_SHARED_RAM` and `MP_SHARED_ROM` attributes in the OIL file. The following example shows how the code generated by RT-Druid can be allocated on two memories `.myram` and `.myrom`:

```
CPU test_application {
  OS EE {
    MP_SHARED_RAM = "__attribute__ ((section (".myram")))";
    MP_SHARED_ROM = "__attribute__ ((section (".myrom")))";
    ...
  };
  ...
};
```

- Since the allocation of multicore data is done automatically by the linker, the design of a multicore system should take care of the fact that more than one CPU can access the same memory component at the same time for data that may be uncorrelated. That is, the bandwidth of the memory component is shared between different tasks that may access multicore global data structures that are not correlated, eventually creating bottlenecks. Such situations can be prevented by appropriately fragmenting memory components if possible, allowing decoupled parallel access to the different banks (for example, instead of having a big onchip memory of 64Kb, having 2 32k memories with an appropriate partition of the data structures).

[5]Note that developers may specify their own custom linker script.

3.5 Handling data consistency using mutual exclusion

Whenever a data structure can be accessed concurrently by different tasks, a mutual exclusion mechanism must be used to ensure that the shared data structure is accessed without consistency problems.

Erika Enterprise provides a mutual exclusion mechanism between concurrent tasks through the abstraction of Resource. Resources are similar to binary semaphores, and can be locked using the `GetResource` primitive and released using the `ReleaseResource` primitive.

A Resource is modeled using an Erika Enterprise Resource object, that can be thought as a binary semaphore plus a rule that says that when a task α allocated on a CPU tries to lock a resource locked by another task β on another CPU, then α actively spins waiting for β to release the lock. This protocol is also called Multiprocessor Stack Resource Policy (MSRP)

These primitives can be used by tasks independently from the CPU where the tasks are allocated to. The RT-Druid Code Generator, and the implementation of the primitives done on Erika Enterprise, guarantees that GetResource and ReleaseResource can be transparently remapped to multicore platform, hiding the complexity of the multicore platform.

Basically, the RT-Druid Code Generator handles in a different way Resources that are used only by task allocated to the same CPU (*local* resources) and Resources that are used by tasks allocated to different CPUs (*global* resources).

Locking a local resource only provoke a modification in the scheduling parameters on the local CPU, whereas locking global resources involve a coordination between tasks allocated to different CPUs that is solved by the Erika Enterprise using an Altera Avalon Mutex peripheral internally.

The kind of lock implementation used for a particular resource only depends on the partitioning defined in the OIL file. Please note that developers does not have to modify their code after a system repartitioning that changes a resource from local to global or vice versa.

The options that influence Resources and their allocation in the OIL configuration file are described in the following example:

```
CPU test_application {
  OS EE {
    NIOS2_MUTEX_BASE = "MUTEX_BASE";
    ...
  };

  TASK thread0 {
    CPU_ID = "cpu0";
    RESOURCE = myresource;
    ...
  };
```

```
  RESOURCE myresource {
    RESOURCEPROPERTY = STANDARD { APP_SRC = "resourcedata.c"; };
  };

  ...
};
```

Basically, the developer have to specify inside the property `NIOS2_MUTEX_BASE` the base address that is generated by the system library inside the `system.h` file. The name of the base address is typically the name used in SOPCBuilder (uppercase), plus the suffix `_BASE`. For example, for a mutex device called `mutex`, then the name of the base address to be used is `MUTEX_BASE`. Then each thread have to declare the resources it use. Finally, the Resource have to be declared, eventually specifying additional source files. When generating the configuration code, if the Resource will be local, these source files will be linked to the CPU where the tasks that use it are allocated; otherwise, the source files will be linked together with the Master CPU.

Please note that (starting from Nios II 6.0, the Erika Enterprise implementation does no more use the HAL mutex driver, but it uses its own mutex driver.

3.6 Startup barrier

Each processor in a multicore environment is characterized by a different set of peripherals, a different size of code and data structures. As a drawback, each CPU will have a different startup code implying that each CPU will have a different boot time (let's consider the boot time the time from the CPU reset to the execution of the first assembler instruction of the `main()` function).

Since the different CPUs will execute a set of interacting code, it is important that all the CPU execution will be synchronized before they start the application, giving in that way the time to all the CPUs to startup their peripherals, and to properly initialize their global data structures. We call that initial synchronization a "startup barrier".

Erika Enterprise provides an (optional) startup barrier service. When configured and correctly used, the CPUs will run the boot code, and will stop at a synchronization point inside the `StartOS()` primitive until all the CPUs have passed it.

The implementation of the startup barrier uses the services of the Altera Avalon Mutex peripheral. In particular, to implement the startup barrier we used the *Initial Owner* feature as shown in Figure 3.3.

The settings to be used when configuring the startup barrier are the following:

- The Altera Avalon Mutex initialization values are set according to Figure 3.3. Please note that if you are using Nios II version 6.0, the value 0 is set in any case and it has **no relationship with the CPUID register values assigned by SOPCBuilder[6].**

[6] If you are using Nios II versions 5.0 and 5.1, 0 is the CPUID register value for the first CPU.

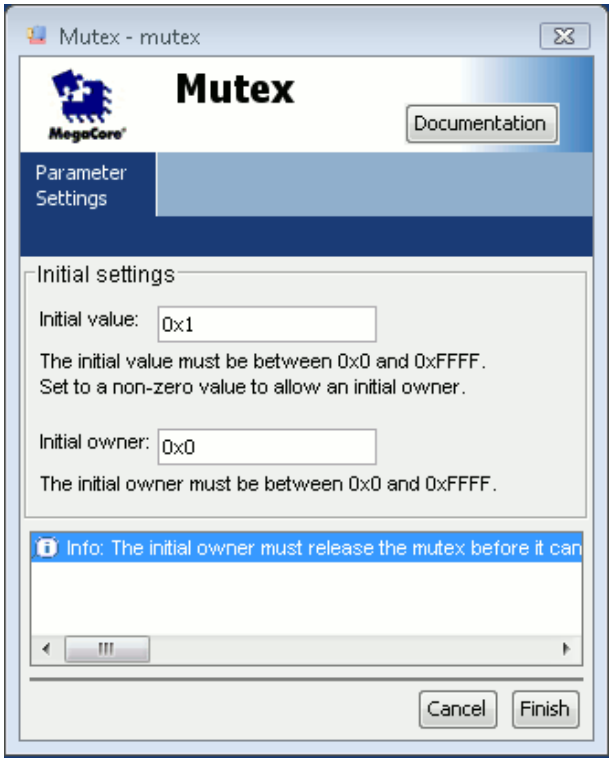

Figure 3.3: This Figure displays the *Initial Value* and the *Initial Owner* setup for the Altera Avalon Mutex peripheral. The setting must be different from 0 only when using the startup barrier feature.

- **The first cpu that is listed in the OIL file is the first CPU that is listed in SOPCBuilder (in this way, RT-Druid will assign that CPU the index 0.**
- **The Master CPU setting in the OIL file is set to the first CPU, and the `STARTUPSYNC` value is set to `TRUE` as shown below:**

```
CPU test_application {
  OS EE {
    MASTER_CPU = "cpu0";
    CPU_DATA = NIOSII {
      ID = "cpu0";
      ...
    };

    CPU_DATA = NIOSII {
      ID = "cpu1";
      ...
    };

    STARTUPSYNC = TRUE;
    ...
};
...
```

If you want not to configure the startup barrier, you have to set all the initialization values of the Altera Avalon mutex to 0, and the `STARTUPSYNC` value to `FALSE`[7].

Warning: If you are using Nios II 5.0 or 5.1, please read the following paragraphs. Please note that unfortunately Altera does not guarantee an assignment pattern to the CPUID (`ctrl5`) values. The CPUID values are assigned in an automatic way by SOPCBuilder, and the user does not have control over it. For that reason, the value that is set as initialization value for the Altera Avalon Mutex may be not what the user would like to. In our experiments, we found that an initialization value of 0 typically corresponded to the first CPU in SOPCBuilder, but unfortunately there is nothing that really specifies the assignment strategy of CPUIDs done by SOPCBuilder.

To check the correctness of the initialization value of the startup barrier, a designer can do the following procedure:

- The designer creates the multicore system, including the Altera Avalon Mutex.

[7]By default, the binary distributions of Erika Enterprise do not allow the possibility to remove the startup barrier.

- Then, the user have to check the value of the CPUID register for the first CPU. To do that, please open the PTF file containing the system description. Inside the `MODULE` section of the various CPUs, search for the `WIZARD_SCRIPT_ARGUMENTS` section. Look at the value for the attribute `cpuid_value`: that is the CPUID value for the considered CPU.
- The value found in the previous step for the Master CPU must be equal to the *Initial Owner* that is set in the Altera Avalon Mutex dialog box (see Figure 3.3).

Warning: Typical behavior that shows up when the mutex initialization is not done in the proper ways is that the either system (all the CPUs) simply blocks at the startup barrier, or simply the barrier is not considered at all. The latter case is subtle because some of the processors could start to use the shared data structures when the CPU responsible for their initialization has not initialized them yet. In this case the early arriving processor could have read uninitialized values, and the values written could be overwritten by the initialization values.

Warning: Please note that the initialization value of the Altera Avalon Mutex done with the startup barrier is only valid when the system boots up the first time. When debugging a multicore systems with a startup barrier, the user have to reboot *all* the CPUs when starting a multicore debug system. Simply stopping the debugger and starting another debug session in the Nios II IDE is not sufficient, because the Altera Mutex is not reset.

3.7 Interprocessor Interrupt

Multicore Nios II systems using Erika Enterprise must include an Interprocessor Interrupt Controller in order for a CPU to notify events to other CPUs. For that reason, Evidence SRL supports interprocessor interrupt controllers for up to 32 processors, made using the standard Altera Avalon PIO SOPCBuilder Component.

Once included into the design, the Interprocessor interrupt feature will be used internally by Erika Enterprise. The developer does not have to care anymore about it.

The interprocessor interrupt is made by two main parts, the *input* part and the *output* part. It has to be created following these guidelines:

- First of all, the designer have to instantiate the *input* part of the Interprocessor Interrupt controller, that is basically an Altera Avalon PIO component for each CPU in the system. The component must be an input PIO, capable to raise an

interrupts on the rising edge of the input pin. Figures 3.4, 3.5, and 3.6 shows the settings to be used for each component. The components have to be connected to each CPU data master. The Interrupt priority should be the lowest among the priorities of the interrupts connected to a given CPU.

- Then, the designer have to instantiate the *output* part of the Interprocessor interrupt controller, that is basically a system wide Altera Avalon output PIO component. The component must have 1 output bit *for each* CPU in the system. Figure 3.7 shows a 2 CPU setting for this component. The component have to be connected to the data master of *all* the CPUs.

- After that, the input part and the output part of the Interprocessor Interrupt have to be connected together. The simplest way to do that is from the BDF file containing the SOPCBuilder Component. After generating the SOPCBuilder Component, please connect each output pin of the output PIO to a correspondent 1-pin input PIO. Figure 3.8 and 3.9 show a simple way to connect them using Altera Quartus II named pins. The pins have to be connected *in the same order specified in the CPUID register* of each CPU. That is, the CPU with `CPUID` 0 have to be connected to the pin 0 of the output PIO, and so on.

- Next, in the OIL configuration file, the CPU have to be listed (again) in the order of the CPUID register, that is the same order they are connected to the interprocessor Interrupt output pins, starting from 0. An example for the 2 CPU snapshots showed in Figure 3.8 and 3.9 is included below. The `IPIC_GLOBAL_NAME` contains the uppercase name of the IPIC output part, as listed in the `system.h` file generated by the Altera System Libraries. The `IPIC_LOCAL_NAME` contains the uppercase name of the IPIC input part, as listed in the `system.h` file generated by the Altera System Libraries for each CPU.

```
CPU test_application {
  OS EE {
    ...
    MASTER_CPU = "cpu0";
    IPIC_GLOBAL_NAME = "IPIC_OUTPUT";
    CPU_DATA = NIOSII {
      ID = "cpu0";
      IPIC_LOCAL_NAME = "IPIC_INPUT_CPU0";
      ...
    };
    CPU_DATA = NIOSII {
      ID = "cpu1";
      IPIC_LOCAL_NAME = "IPIC_INPUT_CPU1";
      ...
    };
    ...
  };
```

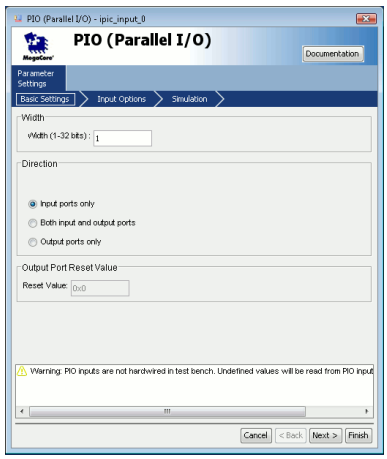

Figure 3.4: Input part of the Interprocessor Interrupt. The Figure shows the Avalon PIO basic settings.

```
    ...
};
```

- Finally, Erika Enterprise will take care of the interprocessor interrupt code. The only thing the user have to specify is the CPU where each task have to be executed; the RT-Druid code generator will configure Erika Enterprise to automatically use interprocessor interrupts when needed.

3.8 Data caches and multicore data sharing

One of the main problems when dealing with shared memories with multicores is the consistency of shared data structures, that have to be ensured in every condition.

The problem of data consistency for the Nios II architectures melts down to two topics: one is the mutual exclusion between data accesses, that is obtained hiding the usage of the Altera Avalon Mutex peripherals inside the RTOS `GetResource()` and `ReleaseResource()` primitives (see Section 3.5), and the other one is related to the usage of data caches with multicore shared data.

This section describes the usage of data cache and of data cache disabling to implement multicore data sharing. Basically, all the accesses to multicore global data structures in Nios II architectures must be done using cache disabling techniques, because the Nios II architecture does not support (yet) cache coherency protocols.

Cache disabling in the Nios II can be obtained using the following methods:

1. Using IORD/IOWR to explicitly say that a particular access must be done using cache disabling;

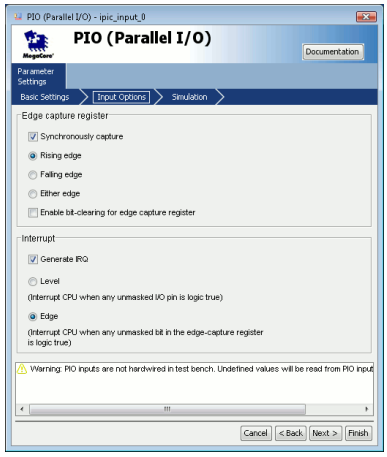

Figure 3.5: Input part of the Interprocessor Interrupt. The Figure shows the Avalon PIO input settings.

Figure 3.6: Input part of the Interprocessor Interrupt. The Figure shows the Avalon PIO simulation settings.

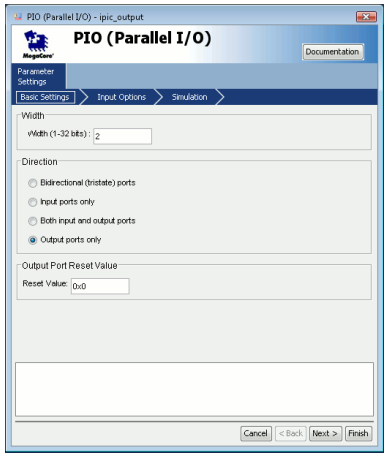

Figure 3.7: Output part of the Interprocessor Interrupt. The Figure shows the Avalon PIO output settings.

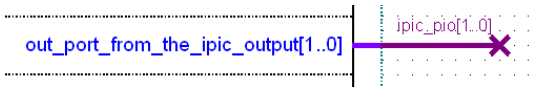

Figure 3.8: Output part of the connection from the output PIO of the SOPCBuilder Component to the input PIO in each CPU.

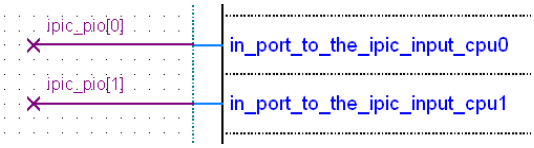

Figure 3.9: Input part of the connection from the output PIO of the SOPCBuilder Component to the input PIO in each CPU.

2. Using tightly coupled memories;
3. Using explicit data cache flushes;
4. Using bit-31 cache disabling;

In most of these cases, (except tightly coupled memories) implementing a cache disabling mechanism implies an explicit modification of the source code.

Instead of requiring an explicit code modification at each data access, Erika Enterprise provides a method for obtaining cache disabling changing *only* the definition of a data structure. That feature is obtained using the bit-31 cache disabling feature of Nios II, and allows the implicit use of bit-31 cache disabling to all the accesses of selected data structures.

The rationale behind using cache disable when dealing with multicore global data is that the designer must write the application code in a way *independent* from the fact that a particular data structure is shared or not between tasks allocated to different processors: that fact indeed depends on the partitioning specified inside the OIL file. A change in the partitioning may influence the fact that a data structure is global or not, and the designer must be as much as possible freed from modifying the code upon a repartitioning of an application.

Another reason to use implicit cache disabling techniques is the support for moving legacy single processor code to a multicore system without changing the source code implementing the application algorithms, but only changing the data definitions. Again, the idea is that the source code must be as much as possible independent from the partitioning scheme chosen by the developer.

As the final thing, since cache disabling reduces the performance of the system because it basically skips the usage of the CPU data cache (if present), cache disabling techniques must be used only for selected data structures, limiting as much as possible the performance penalty related to cache disabling.

3.8.1 How to define cache-disabled multicore global data structures

Erika Enterprise currently offers two techniques to implement cache disabling:

Cache disabling for all the accesses to a multicore global data structure. This technique can be used whenever the developer wants to disable the cache to all the accesses to a particular data structure. In particular, the user has to **define** the multicore global data structure using the `EE_SHARED_DATA` macro. The following example shows how to use the `EE_SHARED_DATA`macro to define a global structure named `mystruct`

```
struct mystruct {
  ...
};

volatile struct mystruct EE_SHARED_DATA(foo);
```

In this example, the `foo` variable is defined as *shared*, and all the accesses done to that variable will use the bit-31 cache disabling technique.

Warning: All the multicore global data structures must be defined in files that are compiled by the `MASTER_CPU` specified in the OIL File.

A typical way of doing that is to put all the multicore global data structures inside a file that is specified in the `MASTER_CPU` CPU section, as in the following example:

```
CPU test_application {
  OS EE {
    ...
    MASTER_CPU = "cpu0";
    CPU_DATA = NIOSII {
      ID = "cpu0";
      APP_SRC = "shareddata.c";
      ...
    };
    ...
  };
};
```

Cache disabling for all the access to shared data structures. This technique uses cache disabling selectively only on the data structures that are shared between different CPUs. The idea is that a multicore application is composed by a set of tasks that uses shared resources.

Shared resources are defined inside the OIL file, and for each task it is specified which resource the task is using (see the OIL example in Section 3.5). Depending on the partitioning of tasks to CPUs chosen by the designer, the resources can be either *local* (that is, all the tasks using the resources are allocated to the same CPU, and the data structure can be global to that CPU only), or *global* (that is, the tasks using the resource are allocated to different CPUs). Only the latter case really needs a cache disabling mechanism[8]

In this case, the `EE_SHARED_RES` macro must be used. `EE_SHARED_RES` takes an additional parameter, that is the OIL resource that must be considered for deciding if cache must be disabled or not. The following examples shows a typical usage of the macro:

```
volatile struct mystruct EE_SHARED_RES(myresource, foo);
```

In this case, the foo data structure will be shared using bit-31 cache disabling *only* when `myresource` is a global resource.

[8] Using cache disabling with the `EE_SHARED_DATA` technique is correct but not efficient, because cache disabling may is used also when a resource become local due to a repartitioning and thus there would have been to need to use cache disabling.

Warning: Special care must be taken in this case, because the definition of all the data structures for resource `myresource` must be put in a separate file. The filename is then listed in the `RESOURCE` section of the `myresource` resource in the OIL file (see the example below). If the resource is local, the file will be compiled together with the CPU where the tasks using it are allocated; if the resource is global, the file will be compiled together with the `MASTER_CPU`.

```
CPU test_application {
  ...
  RESOURCE mutex {
    RESOURCEPROPERTY = STANDARD { APP_SRC = "resourcedata.c"; };
  };
};
```

3.8.2 How to define a data structure that is global to a single processor

To define a data structure that is global to a single CPU, the developer must define a traditional C global variable, making sure that the file that defines the variable is compiled inside the right CPU. To obtain that, the developer have to list the file containing the global variable either in the CPU section of the OIL file (look at `mydata1.c`) or in a task allocated to the particular CPU (look at `mydata2.c`), as in the following example:

```
CPU test_application {
  OS EE {
    ...
    CPU_DATA = NIOSII {
      ID = "mycpu";
      APP_SRC = "mydata1.c";
    };
    ...
    TASK mytask {
      CPU_ID = "mycpu";
      APP_SRC = "mydata2.c";
    };
  };
};
```

Warning: The scope of the global definition is the CPU where the data structure is compiled. Other CPUs will not have access to that data structure symbol, meaning that other CPUs can define symbols with the *same* name.

3.8.3 How to allocate a data structure inside a particular data memory

In general, to allocate a data structure to a particular memory region, the user must explicitly use the section attribute of the gcc compiler, as in the example below. Please note that the memory region must be visible to the CPU defining it.

```
extern int foobar (void)
       __attribute__ ((section (".myonchipmemory")));
```

> **Warning:** If the memory region is not private to a particular CPU, but is shared among different CPUs, then the memory region should also be visible to the `MASTER_CPU`, and the variable that have to be allocated on that memory region should be defined in a file that is compiled by the `MASTER_CPU`.

3.8.4 Disabling GP addressing for global resource access

This section describes a common linking problem when developing multicore application with shared data, and describes workaround for it.

The Nios II architecture and the implementation done with the gcc compiler allows two types of addressing for the data structures: one is the global addressing, where the compiler generates code to compute an absolute address that is then used to access the data; the other method is to maintain a register value containing an absolute address of a memory region, and then use an offset to that location to access the memory. Whereas the latter method is more efficient than the former, it has the limitation that it can access only data structures that lies in a range +/- 32Kbytes from the address used in the register.

The gcc compiler uses the second method to access the small data sections (typically, small data elements like integers, chars, small structures, ...). In particular, it reserves the `gp` register for this use.

When using small multicore global data structures, the compiler continues to generate gp-relative code, but that gp-relative code cannot be used because bit-31 cache disabling generates an address that is far beyond the 32Kb offset possible with relative addressing[9].

The result is that the compiler generates gp-relative code that at the end cannot be linked. A typical linker error in this case is the following:

```
filename:line: Unable to reach this_is_shared (at 0x82012120)
from the global pointer (at 0x0201a064) because the offset
(2147451068) is out of the allowed range, -32678 to 32767.
```

To avoid this linker error the developer has to tell the compiler to disable the GP-relative code generation for that data structure. There are two possible ways that can be used, depending on the final memory where the data structure can be allocated:

[9] That because the bit 31 of the address is 1.

- if the data structure is allocated in the default memory specified in the Altera system library, the developer can use the following declaration

  ```
  extern volatile int this_is_shared EE_DISABLE_GP_ADDRESSING;
  ```

  (this method works in all the cases).

- As an alternative to the previous method, if the data structure is allocated in a particular memory component, the developer can explicitly specify the target memory in the DECLARATION using the gcc section attribute command. An example of a typical declaration is the following:

  ```
  extern volatile int goofy
         __attribute__ ((section (".mymemory")));
  ```

  Please note that **all** the declaration of the data structure must have the sections specified in this way. In general, the gcc compiler consider the attribute specified in the latest declaration.

3.9 Code sharing

Sharing code among processor can be useful if the designer has need to reduce the memory footprint used by some parts of source code.

The source code that can be shared among CPUs must have the following characteristics:

- The code must only use multicore global data or automatic data.

- The code must be linked to the `MASTER_CPU`. For example, a way to obtain that is to list the file containing the shared code inside the Master CPU in the OIL file, as in the following example:

  ```
  CPU test_application {
    OS EE {
      ...
      MASTER_CPU = "cpu0";
      CPU_DATA = NIOSII {
        ID = "cpu0";
        APP_SRC = "mysharedcode.c";
        ...
      };
    ...
  };
  ```

- The code must reside in a memory that is accessible from the other CPUs that needs access to it.

Warning: Please note that a shared code accessing global data (that is, global symbols visible from the code on the `MASTER_CPU` only) may not work if executed on other processors. NO ERRORS are raised by the compiler in these situations, so it is up to the developer to check that everything is generated correctly.

To share a C function, the user has to use the `EE_SHARED_CODE` macro when defining the function. Here is an example:

```
void EE_SHARED_CODE(foobar)(void)
{
    ...
}
```

3.10 On-chip memories, .hex files and multicore systems

Onchip memories are special memories that resides inside the FPGA. Initialization of these memories is done at FPGA reset time when the FPGA data stream is loaded initializing the hardware configuration. When an application is compiled, the initialization values for onchip memories are created inside `.hex` files, and when the hardware project is assembled the initialization values written inside the `.hex` files are assembled inside the `.sof`. In this way, onchip memories are already initialized upon FPGA reset.

The initialization of the onchip memories are stored in `.hex` files that have the name of the memory components. The `.hex` files are typically created by the Altera Nios II System library/Application makefiles. The `.hex` files are copied inside the `.sof` file at the assembly phase in Quartus II. Unfortunately, the Altera Makefile scripts put the `.hex` files inside the hardware project directory (that is, the directory that also contains the SOPCBuilder block files).

In the case of a multicore system, onchip memories will be often used to implement data exchange between processors. That means that more than one CPU will have the right of accessing a particular onchip memory, and also means that the compilation scripts for all the CPUs will produce their `.hex` files with the initialization of these memories. In general, the hex files will be the one produced by the latest compilation of the CPU that can access them.

At the end of the compilation process the Erika Enterprise scripts takes care of handling the `.hex` generation in the following way:

Onchip memories accessible from one CPU only. In this case, the `.hex` file is generated when the CPU ELF file is compiled as part of the compilation process. The `.hex` file contains all the data structures (and their initialization) that the user put on that memory using the gcc section attribute. As an example, to allocate a data structure inside an onchip memory the user has to write:

```
int foo __attribute__ ((section (".myonchipmemory")));
```

Please note that if in the Altera System Library Configuration for the particular CPU the onchip memory has been defined as the privileged memory for data and code, also the corresponding sections (.text for code, .data, .sdata, ... for the data) will be inserted inside the particular onchip memory.

Onchip memories accessible by different CPUs (Master CPU included).

In this case, the final `.hex` file will be produced when compiling the `MASTER_CPU`. This is the common case where the onchip memories are used for data exchange between CPUs: the shared data structures are defined in the Master CPU and used by all the other CPUs (other CPUs will have the data structure only declared, not defined). The Master CPU will also have the burden of initializing the onchip memory.

Onchip memories accessible by different CPUs (Master CPU not included).

This case is currently not handled, and will be probably supported in future releases of Erika Enterprise. The current behavior is that the `.hex` that is finally produced is the one produced by the latest CPU that has been compiled and that can access the memory. The current behavior is that the latest CPU is typically the latest one in the OIL file (this behavior will not be guaranteed in future releases).

In summary:

- When possible, avoid this case.
- If you really need to use it (for example in cases where there is no space on the FPGA for the arbitration logic for the Master CPU, or when using tightly coupled memories between CPUs different from the `MASTER_CPU`), try to write the code in a way that it will not depend on the initialization values of the data stored in that onchip memories.

3.11 Designing multicore software

After having described in the previous sections the basic pieces composing a multicore application, we are now able to highlight the architecture of a typical Erika Enterprise multicore application:

- The platform is defined by a multicore Nios II system composed by a set of CPUs. Each CPU has a set of peripheral connected to it, plus an Altera Avalon Mutex and an Interprocessor Interrupt component.
- The partitioning scheme used on a given configuration is specified inside an OIL configuration file. A a graphical configuration plugin for the Nios II IDE will be available in the next version.
- Tasks should be written each one on a different file. Putting more than one task on the same file is permitted, although it limits the possibilities to partition the

two tasks on different CPUs (the minimal entity that can be partitioned from one CPU to the others is a file). Activating tasks on other CPUs is possible and it is handled transparently by the kernel primitives using multicore IRQs.

- Some data structures will be typically local to a CPU, other data structures will be global to all the CPU that exists in the system. Sharing symbols with cache disabling among the various CPUs is automatically handled by Erika Enterprise.

- Shared data between tasks must be handled using the primitives `GetResource` and `ReleaseResource` primitives independently of the CPU where tasks are allocated. The usage of the Altera Avalon Mutex is hided by Erika Enterprise.

- Task Activations, Task Event Setting, and Alarm notifications that involves them are handled by Erika Enterprise in a way independent from which CPU a particular task is allocated to.

- Counters, Alarms, Application modes, and Hooks are local to a given CPU and are not shared between different CPUs.

- A typical multicore application will consider the fact that a set of tasks related to peripheral/interrupt handling should be allocated to the CPUs where these peripherals are connected to. Eventually these tasks will have to call the Altera HAL peripheral primitives. Further processing, and peripheral independent tasks will be allocated and partitioned among the various available CPUs.

- Resources will be dedicated to the exchange of informations between the various tasks. Resources will be used to protect data structures, that will be shared only if they are shared between tasks allocated to different CPUs.

- Periodic activations of tasks will be handled using Alarms. Alarms are attached to counters, and both are local to the CPU that raises the counting event (typically an IRQ source like a timer). Alarms can activate tasks allocated to other CPUs.

- Support for the usage of Altera HAL primitives is given provided that on each CPU HAL functions are not called concurrently by different tasks. In that case, HAL function calls must be protected by the usage of `GetResource()`/`ReleaseResource()`. Concurrent versions of the Altera HAL primitives will be provided in future versions of Erika Enterprise.

4 OIL Extensions for the Nios II target

This chapter contains OIL extensions specific for the Nios II target. It addresses only the extensions for Nios II.

For a detailed description of the OIL features supported by Erika Enterprise and RT-Druid, which are coomon to all the supported architectures, please refer to the RT-Druid Reference Manual.

4.1 Task Extensions

4.1.1 Stack size

Task stacks can be shared or private. If the system is using shared stack spaces, this information is ignored. In case the system is configured for private stack spaces, each task needs to declare the dimension of its stack space, expressed in bytes, with the following notation.

Definition:

```
ENUM [
  SHARED,
  PRIVATE_NIOSII {
    UINT32 SYS_SIZE;
  }
] STACK = SHARED;
```

Example of declaration:

```
STACK = PRIVATE_NIOSII {
  SYS_SIZE = 0x1000;
  /* or, alternatively
  SYS_SIZE = 4096;
  */
}
```

4.2 Nios II specific extensions

4.2.1 Configuration parameters

The `NIOS2_DO_MAKE_OBJDUMP` option has the same effect of the Nios II preference that produces an objdump of the ELF file on each compilation.

The NIOS2_APP_CONFIG and NIOS2_SYS_CONFIG options specify the configuration options for the application and for the system libraries. Typical values for these options are Debug and Release. Basically, these options specifies a set of configuration parameters used to compile the source code.

The NIOS2_JAM_FILE and NIOS2_PTF_FILE options are mandatory and are used to specify the JAM file (in case of debugging using the Lauterbach Trace32), and the PTF file used for system generation.

Definition:

```
STRING NIOS2_SYS_CONFIG;
STRING NIOS2_APP_CONFIG;
BOOLEAN NIOS2_DO_MAKE_OBJDUMP = FALSE;
STRING NIOS2_JAM_FILE;
STRING NIOS2_PTF_FILE;
```

Example:

```
NIOS2_SYS_CONFIG = "Debug";
NIOS2_APP_CONFIG = "Debug";
NIOS2_DO_MAKE_OBJDUMP = TRUE;
NIOS2_JAM_FILE = "C:/mydirectory/myjamfile.jam";
NIOS2_PTF_FILE = "C:/mydirectory/myptffile.ptf";
```

4.2.2 CPU data

Extensions to the CPU_DATA include the following options:

SYSTEM_LIBRARY_NAME The name of the Altera system library to which the cpu is linked against.

SYSTEM_LIBRARY_PATH The path where the Altera system library for the cpu is located.

IPIC_LOCAL_NAME The input PIO that is used to raise interprocessor interrupts on the CPU.

Example:

```
CPU_DATA = NIOSII {
  ID = "cpu2";
  MULTI_STACK = TRUE {
    IRQ_STACK = FALSE;
    DUMMY_STACK = SHARED;
  };

  APP_SRC = "cpu2_startup.c";
  STACK_TOP = 0x20004000;
  SHARED_SYS_SIZE = 1800;
  SYS_SIZE = 0x1000;
```

```
    SYSTEM_LIBRARY_NAME = "standard_2cpu_cpu2";
    SYSTEM_LIBRARY_PATH = "C:/altera/kits/.../mylibrary";
    IPIC_LOCAL_NAME = "IPIC_IN_2";
  };
```

4.2.3 CPU data extensions for FRSH

The following extensions are used when using the `FRSH` kernel:

```
STRING TIMER_FREERUNNING;
ENUM [
  SINGLE {
    STRING TIMER_IRQ;
  },
  MULTIPLE {
    STRING TIMER_IRQ_BUDGET;
    STRING TIMER_IRQ_RECHARGE;
    STRING TIMER_IRQ_DLCHECK;
    STRING TIMER_IRQ_SEM;
  }
] FRSH_TIMERS;
```

Example with `MULTIPLE`:

```
CPU test_application {
  OS EE {
    ...
    CPU_DATA = NIOSII {
      ...
      TIMER_FREERUNNING = "HIGH_RES_TIMER_0";
      FRSH_TIMERS = MULTIPLE {
        TIMER_IRQ_BUDGET = "TIMER_CAPACITY_0";
        TIMER_IRQ_RECHARGE = "TIMER_RECHARGING_0";
        TIMER_IRQ_DLCHECK = "TIMER_DLCHECK_0";
        TIMER_IRQ_SEM = "TIMER_SEM_0";
      };
    };
  };
  ...
};
```

Example with `SINGLE`:

```
CPU test_application {
  OS EE {
    ...
    CPU_DATA = NIOSII {
      ...
```

```
      TIMER_FREERUNNING = "HIGH_RES_TIMER_0";
      FRSH_TIMERS = SINGLE {
        TIMER_IRQ = "TIMER_CAPACITY_0";
      };
    };
  };
  ...
};
```

The setting is used to specify the timers that the FRSH implementation will use to handle timer events. In this case, there are two possible options: it is possible to specify a single timer source that is used to multiplex all the timer events, or as an alternative all timer events can be assigned a dedicated hardware timer. In the first case, additional software will be loaded for multiplexing the interrupt sources, ending up with a slightly bigger run-time overhead. In the case of a multicore design, each CPU should have its dedicated timers to handle these events.

4.2.4 Mutex options

In the HW configuration of a multiprocessor system the definition of an Altera Avalon Mutex Peripheral is required.

For Nios II versions 6.0 and above, the OIL parameter NIOS2_MUTEX_BASE is used to specify the base address of the Altera HAL mutex device as it can be found in the system.h file.

Definition:

```
STRING NIOS2_MUTEX_BASE;
```

Example:

```
NIOS2_MUTEX_BASE = "MUTEX_BASE";
```

For Nios II versions 5.0 and 5.1, the OIL parameter MUTEX_DEVICE_NAME is used to specify the name of the Altera HAL mutex device as it can be found in the system.h file.

Definition:

```
STRING MUTEX_DEVICE_NAME;
```

Example:

```
MUTEX_DEVICE_NAME = "/dev/mutex";
```

4.2.5 Startup

Erika-specific parameters for Nios II multiprocessors include a few startup options:

STARTUPSYNC This item may be declared as TRUE or FALSE. If TRUE, the startup barrier is enabled at system startup, all CPUs start executing the first line of code at the same time instant. The default value is TRUE.

USEREMOTETASK If set at IFREQUIRED, a task is declared inside the CPU declaration of the processor where it is going to be executed. If a task handles signals coming from tasks/alarms executing on a remote CPU, the task needs to be declared also in those CPUs as a remote task. If USEREMOTETASK is set to ALWAYS, then all tasks are declared in the hosting CPUs and in all the other CPUs, regardless of the fact that they are referred by local code or not. The default value is IFREQUIRED. When using the value IFREQUIRED, the user must specify intertask activations using the LINKED attributes (see section 4.2.6).

Definition:

```
BOOLEAN STARTUPSYNC = TRUE;
ENUM [ALWAYS, IFREQUIRED] USEREMOTETASK = IFREQUIRED;
```

Example:

```
STARTUPSYNC = TRUE;
USEREMOTETASK = ALWAYS;
```

4.2.6 Intertask notifications

If a task needs to activate other tasks or set Events to other extended tasks, then a corresponding declaration needs to be added to the OIL declaration part. If the tasks are mapped on remote cpus, RT-Druid provides the Erika Enterprise kernel with the necessary information so that remote signalling mechanisms (instead of local) are used. The OIL declaration requires the names/identifiers of the tasks to which the signals are directed. This option is typically used to reduce the footprint of the kernel configuration data, because the generation of the kernel data structure that supports remote activations are generated only when needed, saving space in the kernel data structures generated by RT-Druid.

> **Warning:** The specification of these options can be avoided if the option USEREMOTETASK is used with the ALWAYS value (see Section 4.2.5).

Definition:

```
TASK_TYPE LINKED[];
```

Example:

```
CPU mySystem {
  TASK myTask {
    LINKED = "task1";
    LINKED = "task2";
    ...
  };
}
```

4.2.7 Compiler options

Nios II supports an additional parameter `LDDEPS` that has the same meaning of the `LDDEPS` parameter in the Altera Makefiles.

Definition:

```
STRING LDDEPS[];
```

Example:

```
/* this example includes allthe compilation parameters
   for Nios II */
CFLAGS = "-DALT_DEBUG -G0";
CFLAGS = "-O0 -g";
/* The previous two lines are equivalent to
 * CFLAGS = "-DALT_DEBUG -G0 -O0 -g";
 */
CFLAGS = "-Wall -Wl,-Map -Wl,project.map";
ASFLAGS = "-g";
LDDEPS = "\";
LIBS = "-lm";
```

4.3 OIL Extensions for Multiprocessing

4.3.1 Interprocessor Interrupts Extensions

When using multiprocessor systems based on Altera Nios II, the user must specify the Altera HAL names used to implement interprocessor interrupts.

The user must specify inside the OIL file the name of the global Output PIO, by defining the attribute `IPIC_GLOBAL_NAME`, and the name of the Input PIO local to each CPU with the definition of the `IPIC_LOCAL_NAME` as specified in the following example:

```
CPU test_application {
  OS EE {
    ...
    IPIC_GLOBAL_NAME = "IPIC_OUTPUT_PIO";
    CPU_DATA = NIOSII {
      ...
      IPIC_LOCAL_NAME = "IPIC_INPUT_PIO_CPU1";
    };
    ...
  };
  ...
};
```

5 History

Version	Comment
1.0.0	This version is derived from the Multiprocessor Chapter of the Erika Enterprise Reference Manual version 1.2.4, with additions and corrections for Nios II 6.0.
1.2.0	Added the OIL section moved here from the RT-Druid manual; added the new versioning mechanism.
1.2.1	Added intertask notification mechanism from the RT-Druid reference manual.
1.2.2	Added PTF file specification in the OIL file. Screenshots for Nios II 8.0.
1.2.3	Added OIL changes for the FRSH implementation.

Bibliography

[1] OSEK/VDX Consortium. OSEK OIL standard. http://www.osek-vdx.org, 2005.

[2] The AUTOSAR Consortium. AUTOSAR initiative. http://www.autosar.org, 2005.

[3] Altera Corporation. Creating multiprocessor nios ii systems tutorial. Nios II literature page, http://www.altera.com/literature/lit-nio2.jsp, 2005.

Index

Altera Avalon Mutex, 28, 29
Altera Avalon Mutex peripheral, 21
Automatic data, 22

EE_SHARED_DATA, 37
EE_SHARED_RES, 38

Global data, 23
global resources, 28

Heap, 23

Interprocessor Interrupt, 32
ISR Category 1, 16
ISR Category 2, 16

Local data, 23
local resources, 28

Master CPU, 20, 25, 26, 29, 31, 32, 43
MASTER_CPU, 38
Multicore global data, 25

NIOS2_MUTEX_BASE, 29

OIL Example, 21, 27, 28, 31, 33, 38, 39, 41

private stack, 25

Resource, 28

Separate exception stack, 16
Stack, 25
Stack, shared, 25
StartOS, 16, 29
Startup Barrier, 32
Startup barrier, 29
Static data, 22