Universität Augsburg
Fakultät für Angewandte Informatik

parMERASA Multi-core RTOS Kernel

Florian Kluge, Christian Bradatsch

Report 2013-02 January 2013

INSTITUT FÜR INFORMATIK
D-86159 AUGSBURG

Table of Contents

Abstract

The parMERASA project is the response to demands from European avionic, automotive and automation industries for increased performance at reduced costs while maintaining safety levels. Its aim is to stimulate industrial, social and environmental changes by demonstrating the benefits of moving from embedded single core to multi-core processors which can run applications in parallel, speeding up performance and cutting costs.

This report presents requirements for a hard real-time capable multi-core Kernel Library in embedded systems and the transfer to the parMERASA simulator. It shows details of context and memory management, synchronization mechanisms and interrupt handling relating to the Power PC instruction set architecture used by the simulator.

1 Introduction

Engineers who design hard real-time embedded systems express a need for significant increases in the hardware performance over that available today, but without compromising the safety-critical nature of their software. A breakthrough in performance is expected by parallelizing hard real-time applications onto multi-core hardware. parMERASA will provide a timing analysable system of parallel hard real-time applications running on a scalable multi-core processor. Several new scientific and technical challenges will be tackled in the context of timing analysability: parallelization techniques for industrial applications, operating system virtualization and efficient synchronization mechanisms, worst-case execution times (WCET) of parallelized applications, verification and profiling tools, scalable memory hierarchies and I/O systems for multi-core processors.

Hard real-time applications, such as flight management system, automotive engine and drilling machine control, will be parallelized and executed on an embedded multicore processor. The parMERASA multi-core processor and system software is expected to scale up to 64 cores.

This document describes the parMERASA Multi-core Kernel Library, which builds together with domain specific RTEs (runtime environment) a common system architecture for a many-core processor suitable for the three application domains automotive, avionic, and construction machinery. It also comprises a short overview of the overall system architecture concept composed of simulated hardware, kernel services, RTE services and application layer.

This report is organized as follows: Section 2 gives a short overview of the requirements of a RTOS kernel influenced by requirements of the application domains. In section 3 the Kernel Library is specified and appendix A1 comprises the corresponding application interface.

2 Requirements

In this section, we state the requirements for the Kernel Library for embedded systems with many-core hardware according to the needs for the domain specific applications. It has also been taken care of the WCET analysability of the Kernel Library components for better support of verification tools.

The three application domains automotive, avionic and construction machinery require different services from the particular RTE. Hence the aim of the Kernel Library is to build a common basis for these RTEs. Table 1 highlights the requirements of the RTE services of the three application domains.

	Automotive	**Avionic**	**Const. Machinery**
Scheduling Strategy	Fixed priority pre-emptive (Earliest Deadline First)	Fixed cyclic + Fixed priority pre-emptive	Round-robin
Communication & Synchronization	Resources, Events, buffered/unbuffered Messages	Messages; Events, Buffers, Black-boards, Semaphores	Events, Semaphores, Spin-locks
I/O Requirement	Low latency	Predictability	Low latency
Protection Unit	OS-Application, (Task)	Partition	Task

Table 1: Comparison of application domain specific RTE services

Process and thread management capabilities respectively a system scheduler are essential to all RTEs. Since every RTE uses its own scheduling strategy, the Kernel Library supports the different RTE schedulers by providing elementary components organized in *context management* services. Similarly the communication and synchronization features of the application domains are supported by basic synchronization mechanisms providing a common basis for the high level synchronization services. Common to all three RTEs are the need for *memory management/protection* services in a many-core system to further guarantee the concept of freedom of interference, which is accommodated by temporal segregation (cyclic scheduling) and spatial partitioning. Likewise all application domains need access to peripheral hardware, which is done in a uniform and simple way through *interrupt handling* services. In summary the Kernel Library builds the fundamental services also used for higher level RTE services, which together form the basement for the application software.

3 Kernel Library Specification

The Kernel Library provides the common basis for the implementation of the RTE subsets required by the parMERASA applications. It incorporates the four kernel services *context management*, *memory management*, *synchronization mechanisms*, and *interrupt handling*. These are used to implement the four RTE services (see Figure 1). The Kernel Library provides basic hardware abstractions for RTE services. The provided Kernel Library functions are listed in Appendix A1.

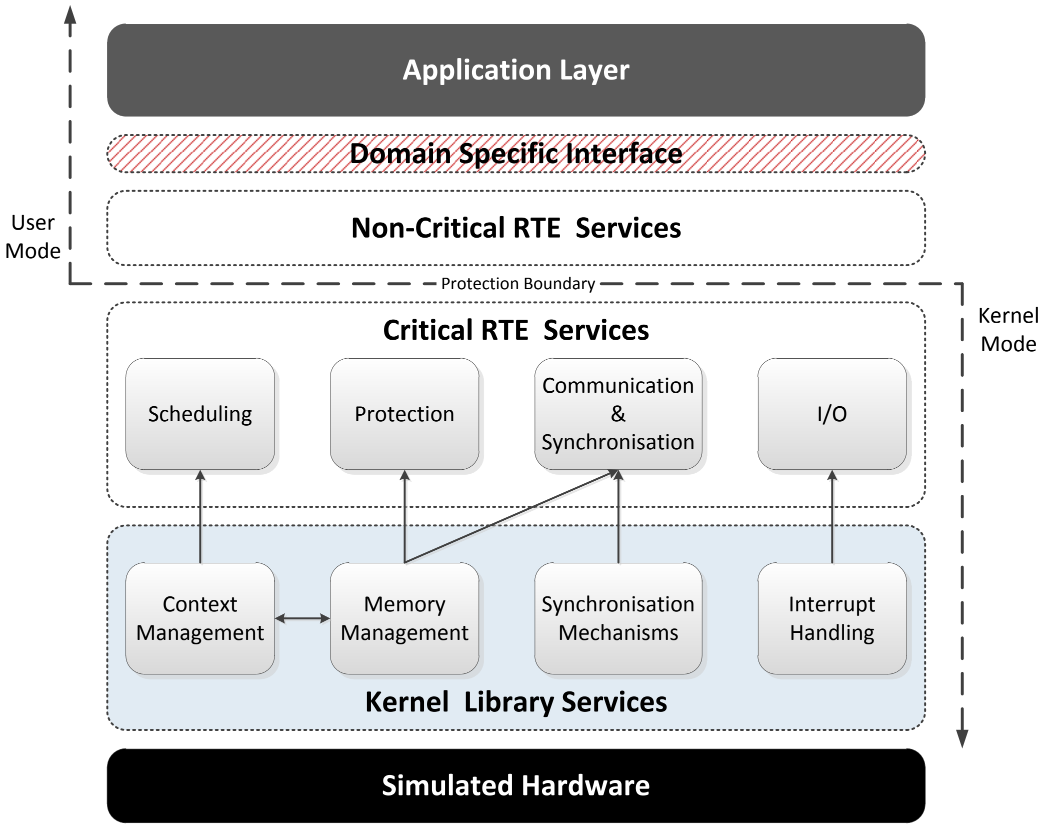

Figure 1: Entanglement of RTE and Kernel Services in parMERASA System Architecture

The four RTE services *scheduling*, *protection*, *communication & synchronization*, and *I/O* are dependent on their corresponding kernel service, but there are also cross dependencies inside kernel services and between kernel and RTE services. Figure 1 also shows the protection boundary between user level and kernel mode respectively supervisor level. According to that boundary, the RTE services are divided into non-critical/critical RTE services. A detailed description of the single Kernel Library Services and their linkage in-between is given in the following sections.

3.1 Context Management

The main task of a scheduler is to assign processing time to processes or threads. The RTE dependent scheduling algorithm decides which process is executed at any time. The common action for all schedulers is to store the active process and at the same time load the next process. Therefore, a swap of the processor context has to take place. A process' context consists of all CPU registers necessary to continue execution from a former state. These are, amongst others, mostly general purpose, link and stack pointer registers. Depending on the Instruction Set Architecture (ISA) additionally frame, floating point and special purpose registers are also implied. Thus, the **context management** must provide means to initialize a process' context and to swap the processor's execution context. Since a process context has a fixed size contingent on the ISA, a context swap has a constant execution time.

3.2 Memory Management

The **memory management** service is responsible for access privileges and mapping of local and shared memories, even beyond cluster boundaries. As the protection facility memory management ensures that a certain core can only access address ranges intended for it. Physically it can be memory only locally accessible as well as common memory shared among several cores. These memory areas can be spread over the complete physical address space. So it is also up to the memory management service to provide a continuous address space to each core by translating virtual to physical addresses. For this purpose Translation Lookaside Buffers (TLBs) are used. TLBs support the translation of addresses by specialized hardware in a fast and efficient way. The provisioning of shared memory areas is also required by the communication & synchronization service to implement for example message queues or buffers, which are accessed by at least two different processes.

The mapping of virtual to physical address ranges is statically assigned during boot-up phase via the memory management service. Therefore the address mappings are placed into the TLBs. To meet hard real-time capability all mappings have to fit into the provided number of TLBs to avoid dynamic memory reallocation during runtime.

3.3 Synchronization mechanisms

The **synchronization mechanisms** provide functionalities for simple software synchronization mechanisms and hardware primitives to implement complex mechanisms. On software mechanism side a hard real-time capable spin-lock is offered, i.e. ticket lock. Spin-locks belong to the category of busy-waiting synchronization techniques, meaning that program execution is blocked until a specific condition is reached. They are not intended to be explicitly used by the applications, because of the lack of fairness, but only to implement more intricate synchronization mechanisms such as mutex locks.

On the other side there are blocking synchronization techniques, for example barriers, which need to interact with the system scheduler to start and stop threads. Since the scheduler is located in the RTE services the whole functionality of blocking software synchronization techniques cannot be placed in the kernel services. Therefore supportive hardware synchronization primitives are provided by kernel services. The key hardware primitive has a "compare and swap" semantic, which can be used by the communication & synchronization RTE service to deliver intricate software synchronization mechanisms such as barriers or semaphores. The concrete implementation of the compare and swap function is hidden from the user and can be adapted to different instruction set specific atomic commands.

Both synchronization techniques have in common that two or more processes/threads either spin on a shared variable or read/write to a shared memory location. For this purpose the corresponding memory locations have to be accessible by the designated processes, handled by the memory management service as stated in the prior section. Publications by UAU and UPS show how these basic primitives can be used to implement timing predictable synchronization functions [1] [2].

3.4 Interrupt Handling

To react on internal and external events the **interrupt handling** service is used. It allows the implementation of a unique handler routine for any kind of event respectively interrupt service request. Events can be distinguished between software and hardware caused interrupts. Software interrupts imply exceptions, caused by unintentional faults e.g. division by zero, and volitional program interruptions e.g. system call. On the other side hardware interrupts can arise form core integrated devices like timers or from external connected I/O devices like a CAN controller.

For each supported interrupt service request a predefined standard handler routine exists, which can be replaced by a custom one. Except for system calls, which have to be registered to the interrupt handling service. There is no standard handler available as it is up to the RTE implementation to specify individual system calls.

Custom interrupt handler routines should be implemented in a way that they guarantee time predictability and low latency. Time-consuming or extensive computations are delegated to RTE I/O services. Complex operation logic for I/O devices is placed in the RTE I/O services whereas a rapid response to events from I/O devices is controlled by an interrupt handling routine. In order to access memory mapped I/O the RTE I/O service also needs to work together with the memory management service, which grants read/write operations to the desired I/O.

List of Figures/Tables

List of References

[1] M. Gerdes, F. Kluge, T. Ungerer, C. Rochange und P. Sainrat, „Time Analysable Synchronisation Techniques for Parallelised Hard Real-Time Applications," in *Proceedings of Design, Automation and Test in Europe (DATE'12)*, 2012.

[2] M. Gerdes, F. Kluge, T. Ungerer und C. Rochange, „The Split-Phase Synchronisation Technique: Reducing the Pessimism in the WCET Analysis of Parallelised Hard Real-Time Programs," in *Proc. of the 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'12)*, 2012.

List of Appendices

Appendix A1 – parMERASA Kernel Library

Appendix A1: parMERASA Kernel Library

Version 1.0

18/01/2013

Table of Contents

Kernel Library

The kernel library comprises functionalities to abstract from the underlying hardware. The aim is to supersede the usage of assembly for the programmer. For this purpose the kernel library is divided into the sections **Interrupt Handling**, **Context Management**, **Memory Management** and **Synchronization Mechanisms**.

Version:

1.0

Author:

Christian Bradatsch

Module Index

Modules

Here is a list of all modules:

Data Structure Index

Data Structures

Here are the data structures with brief descriptions:

File Index

File List

Here is a list of all documented files with brief descriptions:

Module Documentation

Interrupt Handling

Macros

- #define **PROGRAM_IRQ_ILLEGAL** 0x8000000

  Mask for exception syndrome.

- #define **PROGRAM_IRQ_PRIVILEGED** 0x4000000

  Mask for exception syndrome.

- #define **PROGRAM_IRQ_TRAP** 0x2000000

  Mask for exception syndrome.

- #define **PROGRAM_IRQ_UNIMPLEMENTED** 0x1000000

  Mask for exception syndrome.

Functions

- static void **enable_external_irq** ()

  Enable external interrupts.

- static void **disable_external_irq** ()

  Disable external interrupts.

- void **clear_external_irq** ()

  Activate external interrupts.

- static uint32_t **save_external_irq** ()

  Save external interrupt status flag.

- static void **restore_external_irq** (uint32_t msr)

  Restore external interrupt flag.

- uint64_t **get_time_base** ()

  Get the time base measured in clock cycles.

- void **set_time_base** (uint64_t time)

  Set the time base.

- void **enable_pit** (uint32_t interval)

Enable the programmable interval timer.

- void **disable_pit** ()

  Disable the programmable interval timer.

- void **clear_pit** ()

  Clear the programmable interval timer.

Custom Interrupt Handler

- void **_irq_program_handler** (uint32_t esr)

  Defines a custom program interrupt handler.

- uint32_t **_irq_sys_handler** (uint32_t arg1, uint32_t arg2, uint32_t arg3, uint32_t arg4, uint32_t arg5, uint32_t scno)

  Defines a custom system call handler.

- void **_irq_timer_handler** (void)

  Defines a custom programmable interval timer interrupt handler.

- void **isr_pre_hook** (uint32_t cause)

  Declaration of ISR pre hook routine.

- void **isr_post_hook** (uint32_t cause)

  Declaration of ISR post hook routine.

System Call Macros

- #define **_syscall0**(type, name)

  System Call without arguments.

- #define **_syscall1**(type, name, type1, arg1)

  System Call with 1 argument.

- #define **_syscall2**(type, name, type1, arg1, type2, arg2)

  System Call with 2 arguments.

- #define **_syscall3**(type, name, type1, arg1, type2, arg2, type3, arg3)

  System Call with 3 arguments.

- #define **_syscall4**(type, name, type1, arg1, type2, arg2, type3, arg3, type4, arg4)

  System Call with 4 arguments.

- #define **_syscall5**(type, name, type1, arg1, type2, arg2, type3, arg3, type4, arg4, type5, arg5)

System Call with 5 arguments.

Detailed Description

Interrupt Handling includes definitions to add a custom interrupt handler for different interrupt sources.

Macro Definition Documentation

#define _syscall0(type, name)

```
Value:type name(void)                                              \
    {                                                              \
        {                                                          \
            register unsigned long __sc_0 __asm__ ("r0");          \
            register unsigned long __sc_3 __asm__ ("r3");          \
                                                                   \
            __sc_0 = __NR_##name;                                  \
            __asm__ __volatile__                                   \
                ("sc"                                              \
                : "=&r" (__sc_3)                                   \
                : "0"   (__sc_3), "r"   (__sc_0)                   \
                : __syscall_clobbers);                             \
        }                                                          \
        return (type) __sc_3;                                      \
    }
```

System Call without arguments.

Parameters:

type	- type of the return value.
name	- name of the system call.

#define _syscall1(type, name, type1, arg1)

```
Value:type name(type1 arg1)                                        \
    {                                                              \
```

```
        {                                                    \
            register unsigned long __sc_0 __asm__ ("r0");   \
            register unsigned long __sc_3 __asm__ ("r3");   \
                                                            \
            __sc_3 = (unsigned long) (arg1);                \
            __sc_0 = __NR_##name;                           \
            __asm__ __volatile__                            \
                ("sc"                                       \
                : "=&r" (__sc_3)                            \
                : "0"   (__sc_3), "r"   (__sc_0)            \
                : __syscall_clobbers);                      \
        }                                                   \
        return (type) __sc_3;                               \
    }
```

System Call with 1 argument.

Parameters:

type	- type of the return value.
name	- name of the system call.
typeX	- type of the Xth argument.
argX	- value of the Xth argument.

#define _syscall2(type, name, type1, arg1, type2, arg2)

```
Value:type name(type1 arg1, type2 arg2)                       \
    {                                                        \
        {                                                    \
            register unsigned long __sc_0 __asm__ ("r0");   \
            register unsigned long __sc_3 __asm__ ("r3");   \
            register unsigned long __sc_4 __asm__ ("r4");   \
                                                            \
            __sc_3 = (unsigned long) (arg1);                \
            __sc_4 = (unsigned long) (arg2);                \
```

```
            __sc_0 = __NR_##name;                           \
            __asm__ __volatile__                            \
                ("sc"                                       \
                : "=&r" (__sc_3)                            \
                : "0"   (__sc_3), "r"   (__sc_0),           \
                  "r"   (__sc_4)                            \
                : __syscall_clobbers);                      \
        }                                                   \
        return (type) __sc_3;                               \
    }
```

System Call with 2 arguments.

Parameters:

type	- type of the return value.
name	- name of the system call.
typeX	- type of the Xth argument.
argX	- value of the Xth argument.

#define _syscall3(type, name, type1, arg1, type2, arg2, type3, arg3)

Value:
```
type name(type1 arg1, type2 arg2, type3, arg3)          \
    {                                                   \
        {                                               \
            register unsigned long __sc_0 __asm__ ("r0");   \
            register unsigned long __sc_3 __asm__ ("r3");   \
            register unsigned long __sc_4 __asm__ ("r4");   \
            register unsigned long __sc_5 __asm__ ("r5");   \
                                                            \
            __sc_3 = (unsigned long) (arg1);                \
            __sc_4 = (unsigned long) (arg2);                \
            __sc_5 = (unsigned long) (arg3);                \
            __sc_0 = __NR_##name;                           \
            __asm__ __volatile__                            \
```

```
                ("sc"                                       \
                : "=&r" (__sc_3)                            \
                : "0"   (__sc_3), "r"   (__sc_0),           \
                  "r"   (__sc_4),                           \
                  "r"   (__sc_5)                            \
                : __syscall_clobbers);                      \
        }                                                   \
        return (type) __sc_3;                               \
    }
```

System Call with 3 arguments.

Parameters:

type	- type of the return value.
name	- name of the system call.
typeX	- type of the Xth argument.
argX	- value of the Xth argument.

#define _syscall4(type, name, type1, arg1, type2, arg2, type3, arg3, type4, arg4)

```
Value:type name(type1 arg1, type2 arg2, type3 arg3, type4 arg4)   \
    {                                                       \
        {                                                   \
            register unsigned long __sc_0 __asm__ ("r0");   \
            register unsigned long __sc_3 __asm__ ("r3");   \
            register unsigned long __sc_4 __asm__ ("r4");   \
            register unsigned long __sc_5 __asm__ ("r5");   \
            register unsigned long __sc_6 __asm__ ("r6");   \
                                                            \
            __sc_3 = (unsigned long) (arg1);                \
            __sc_4 = (unsigned long) (arg2);                \
            __sc_5 = (unsigned long) (arg3);                \
            __sc_6 = (unsigned long) (arg4);                \
            __sc_0 = __NR_##name;                           \
```

```
            __asm__ __volatile__                            \
                ("sc"                                       \
                : "=&r" (__sc_3)                            \
                : "0"   (__sc_3), "r"   (__sc_0),           \
                  "r"   (__sc_4),                           \
                  "r"   (__sc_5),                           \
                  "r"   (__sc_6)                            \
                : __syscall_clobbers);                      \
        }                                                   \
        return (type) __sc_3;                               \
    }
```

System Call with 4 arguments.

Parameters:

type	- type of the return value.
name	- name of the system call.
typeX	- type of the Xth argument.
argX	- value of the Xth argument.

#define _syscall5(type, name, type1, arg1, type2, arg2, type3, arg3, type4, arg4, type5, arg5)

```
Value:type name(type1 arg1, type2 arg2, type3 arg3, type4 arg4, type5 arg5)   \
    {                                                       \
        {                                                   \
            register unsigned long __sc_0 __asm__ ("r0");   \
            register unsigned long __sc_3 __asm__ ("r3");   \
            register unsigned long __sc_4 __asm__ ("r4");   \
            register unsigned long __sc_5 __asm__ ("r5");   \
            register unsigned long __sc_6 __asm__ ("r6");   \
            register unsigned long __sc_7 __asm__ ("r7");   \
                                                            \
            __sc_3 = (unsigned long) (arg1);                \
```

```
            __sc_4 = (unsigned long) (arg2);                    \
            __sc_5 = (unsigned long) (arg3);                    \
            __sc_6 = (unsigned long) (arg4);                    \
            __sc_7 = (unsigned long) (arg5);                    \
            __asm__ __volatile__                                \
                ("sc"                                           \
                : "=&r" (__sc_3)                                \
                : "0"   (__sc_3), "r"   (__sc_0),               \
                  "r"   (__sc_4),                               \
                  "r"   (__sc_5),                               \
                  "r"   (__sc_6),                               \
                  "r"   (__sc_7)                                \
                : __syscall_clobbers);                          \
        }                                                       \
        return (type) __sc_3;                                   \
    }
```

System Call with 5 arguments.

Parameters:

type	- type of the return value.
name	- name of the system call.
typeX	- type of the Xth argument.
argX	- value of the Xth argument.

#define PROGRAM_IRQ_ILLEGAL 0x8000000

Mask for exception syndrome.

If

```
(esr & PROGRAM_IRQ_ILLEGAL) != 0
```

an illegal instruction caused the program interrupt.

See Also:

_irq_program_handler

#define PROGRAM_IRQ_PRIVILEGED 0x4000000

Mask for exception syndrome.

If

```
(esr & PROGRAM_IRQ_PRIVILEGED) != 0
```

a privileged instruction caused the program interrupt. This happens while an instruction limited to *privileged mode* is executed in *user mode* .

See Also:

_irq_program_handler

#define PROGRAM_IRQ_TRAP 0x2000000

Mask for exception syndrome.

If

```
(esr & PROGRAM_IRQ_TRAP) != 0
```

an trap instruction caused the program interrupt.

See Also:

_irq_program_handler

#define PROGRAM_IRQ_UNIMPLEMENTED 0x1000000

Mask for exception syndrome.

If

```
(esr & PROGRAM_IRQ_UNIMPLEMENTED) != 0
```

an APU or FPU instruction, which is not implemented, caused the program interrupt.

See Also:

_irq_program_handler

Function Documentation

void _irq_program_handler (uint32_t esr)

Defines a custom program interrupt handler.

By default a standard handler is executed upon an program interrupt event. To define a custom handler the function *_irq_program_handler* has to be implemented and the USER_PROGRAM_IRQ define in file *kernel_lib.mk* has to be set to (y)es.

To distinguish between different causes of the program interrupt the value of *esr* has to be examined.

Parameters:

esr	- status of the exact exception syndrome of the program interrupt.

See Also:

PROGRAM_IRQ_ILLEGAL, **PROGRAM_IRQ_PRIVILEGED**, **PROGRAM_IRQ_TRAP**, **PROGRAM_IRQ_UNIMPLEMENTED**

uint32_t _irq_sys_handler (uint32_t **arg1**, *uint32_t* **arg2**, *uint32_t* **arg3**, *uint32_t* **arg4**, *uint32_t* **arg5**, *uint32_t* **scno***)*

Defines a custom system call handler.

By default a standard handler is executed upon an syscall interrupt event. To define a custom handler the function *_irq_sys_handler* has to be implemented and the USER_SYSCALL_IRQ define in file *kernel_lib.mk* has to be set to (y)es.

To distinguish between different system calls a unique system call number has to be assigned to every system call, for example:

```
#define __NR_sys_read        1

#define __NR_sys_write       2
```

To distinguish between several system calls, the paramter *scno* can be used. It contains the number of the system call. For correct usage of the arguments *arg1* - *arg5* they have to be explicitly casted to the types stated in *_syscallX* .

Parameters:

argX	- Xth argument to the system call handler.
scno	- number of the system call.

See Also:

_syscall0, **_syscall1**, **_syscall2**, **_syscall3**, **_syscall4**, **_syscall5**

void _irq_timer_handler (void)

Defines a custom programmable interval timer interrupt handler.

By default a standard handler is executed upon an timer interrupt event. To define a custom handler the function *_irq_timer_handler* has to be implemented and the USER_TIMER_IRQ define in file *kernel_lib.mk* has to be set to (y)es.

void clear_external_irq ()

Activate external interrupts.

Activates external interrupts respectively clears the internal status register. This should be done in the external interrupt service routine after an external interrupt was generated to reactivate the interrupt.

See Also:

disable_external_irq, **enable_external_irq**

void clear_pit ()

Clear the programmable interval timer.

Clears the programmable interval timer (PIT) after a PIT interrupt occurred. This should be done in the PIT interrupt service routine.

See Also:

disable_pit, **enable_pit**

static void disable_external_irq () `[inline]`, `[static]`

Disable external interrupts.

Disables and deactivates external interrupts in order that no interrupt is generated from its source.

See Also:

enable_external_irq, **clear_external_irq**

void disable_pit ()

Disable the programmable interval timer.

Disables and deactivates the programmable interval timer (PIT) in order that no interrupt is generated from its source.

See Also:

enable_pit, **clear_pit**

static void enable_external_irq () `[inline], [static]`

Enable external interrupts.

Enables and activates external interrupts. If an external IRQ is signaled of an external interrupt the event is saved in an internal status register. An interrupt is only generated, if external interrupts are enabled, the internal status register is cleared (activated) and an external interrupt event occurs.

External interrupts are automatically deactivated (internal status register value remains) after an interrupt was generated. To activate external interrupts again *activateExternalIrq* has to be called.

See Also:

disable_external_irq, **clear_external_irq**

void enable_pit (uint32_t **interval***)*

Enable the programmable interval timer.

Enables and activates the programmable interval timer (PIT) and sets the time for generating a PIT interrupt to *interval* .

The internal counter is initialized with *interval* and is decremented every clock cycle. A timer event occurs when the counter value is 1 and gets decremented (counter overflow 1-0). The event is stored in an internal status register. A PIT interrupt is only generated, if the PIT is enabled, the internal status register is cleared (activated) and a timer event occurs.

The PIT is automatically deactivated (internal status register value remains) after a PIT interrupt was generated. To activate the PIT again *activatePIT* has to be called. Nonetheless the counter is automatically loaded with the last *interval* value after every counter overflow. The PIT re-activation only concerns the interrupt generation and not the internal counter facility.

Parameters:

interval	- a value greater than 0 representing the time in clock cycles after a PIT interrupt is generated.

Note:

To fully disable the PIT call *disable_pit* .

See Also:

disable_pit, **clear_pit**

uint64_t get_time_base ()

Get the time base measured in clock cycles.

The time base is incremented on every source clock cycle. The measuring begins at value 0 after resetting the processor or at a certain value set by the user with *setTimeBase* .

Returns:

the time in clock cycles counted from the last value set by *set_time_base* or processor reset.

See Also:

set_time_base

void isr_post_hook (uint32_t cause*)*

Declaration of ISR post hook routine.

The hook routine has to be called at the end of interrupt service routine but before switching to normal program execution.

Parameters:

cause	- value representing the interrupt cause

See Also:

isr_pre_hook, **ctx_switch_hook**

void isr_pre_hook (uint32_t cause*)*

Declaration of ISR pre hook routine.

The hook routine has to be called after an interrupt event at the beginning of interrupt service routine. The hook routine can be used for profiling support for example.

Parameters:

cause	- value representing the interrupt cause

See Also:

isr_post_hook, **ctx_switch_hook**

static void restore_external_irq (uint32_t msr*)* `[inline]`, `[static]`

Restore external interrupt flag.

Restores the status of the external interrupt flag, which was saved with *saveExternalIrq* before.

Parameters:

msr	- value containing the status flag.

See Also:

save_external_irq

static uint32_t save_external_irq () [inline], [static]

Save external interrupt status flag.

Saves the external interrupt status flag. This should be done before disabling external interrupts.

Returns:

value containing status flag.

See Also:

restore_external_irq

void set_time_base (uint64_t **time)**

Set the time base.

Sets the time base to the value specified by *time* . After setting the time base, it is incremented starting from the specified value.

Parameters:

time	- value to be set in clock cycles.

See Also:

get_time_base

Context Management

Data Structures

- struct **regs**
- *Structure containing the registers involved in context switching.* struct **context_t**

Structure used for context switching. Typedefs

- typedef struct **regs regs_t**

Structure containing the registers involved in context switching.

- typedef
- RTE_SPECIFIC_TASK_IDENTIFIER **task_identifier_t**

  Type of task identifier.

Functions

- **context_t** * **init_context** (void *sp, uint32_t size, void(*func)(void *))

  Initialize a context for use with context switching.
- void **switch_context** (**context_t** **oldctx, **context_t** *newctx)

  Switch from the actual context to a new context.
- void **ctx_switch_hook** (**task_identifier_t** *task_id)

  Declaration of context switch hook routine.

Detailed Description

The context management includes features to maintain in the first instance process and thread scheduling.

Typedef Documentation

typedef RTE_SPECIFIC_TASK_IDENTIFIER task_identifier_t

Type of task identifier.

Defines the task identifier type of the ***ctx_switch_hook()*** . To support different RTE implementations the *task_identifier_t* type has to be set in file *kernel_lib.mk* via *RTE_SPECIFIC_TASK_IDENTIFIER* to the RTE specific task identifier.

Function Documentation

*void ctx_switch_hook (task_identifier_t * task_id)*

Declaration of context switch hook routine.

The hook routine is executed before the call of ***switch_context()*** . Therefore ***ctx_switch_hook()*** must be called directly before ***switch_context()*** .

Parameters:

task_id	- task id of the new context.

See Also:

isr_pre_hook, **isr_post_hook**

context_t init_context (void * sp, uint32_t size, void(*)(void *) func)*

Initialize a context for use with context switching.

Allocates memory for a context of type ***context_t*** and initializes its values. The function pointer *func* points to the entry function of the context, which is called after first switch to this context.

A short programming example is provided under ***switch_context***

Parameters:

sp	- pointer to the top of the stack of a context. The initContext function reserves a frame for the context on this stack.
size	- size of the stack.
func	- function pointer to the entry function.

Returns:

a pointer to the initialized context.

Note:

Context initialization should be done before first usage of *switch_context* , otherwise the behavior is undefined.

See Also:

switch_context

*void switch_context (context_t ** oldctx, context_t * newctx)*

Switch from the actual context to a new context.

The current context is saved and its location is stored in pointer *oldctx* . Then the context indicated by pointer *newctx* is restored and program execution of the restored context is resumed.

Short example for using *initContext* and *switchContext:*

```
context_t *procA = initContext(topOfStackA, &funcA);

context_t *procB = initContext(topOfStackB, &funcB);
```

```
// start process A (procA)
// process A begins execution at entry point of function A (funcA)

switchContext(&procA, procB);

// process A is stopped
// process B begins execution at entry point of function B (funcB)
// process B (procB) is running

switchContext(&procB, procA);

// process B is stopped
// process A is continued
```

Parameters:

oldctx	- address of the pointer to the old context.
newctx	- pointer to the new context.

Note:

New context must be initialized before first usage of *switch_context* . Otherwise behavior is undefined.

See Also:

init_context

Memory Management

Data Structures

- struct **TLBentry**

Structure of a TLB entry. Macros

- #define **NTLB** 64

Defines the number of TLB entries.

- #define **RO_DATA** 0

  Constant for TLB entry read only access for data

- #define **RW_DATA** 1

  Constant for TLB entry read/write access for data

- #define **RO_DATA_INS** 2

  Constant for TLB entry read only access for data and instructions

- #define **RW_DATA_INS** 3

  Constant for TLB entry read/write access for data and read access for instructions

- #define **ENTRY**(_epn, _size, _rpn, _ap) {.epn = _epn, .size = _size, .rpn = _rpn, .ap = _ap}

  Define a new TLB entry.

- #define **CORETLB**(CID, args...) **TLBentry** entries_##CID[] = {args, **ENTRY**(0, 0, 0, 0)}

  Define a new TLB entry table for a specific core.

Typedefs

- typedef struct **TLBentry TLBentry_t**

  Structure of a TLB entry.

Detailed Description

Memory management permits protection and mapping of memory areas, which are divided in so called memory pages. Each memory page has its own access privileges for reading/writing data and fetching instructions for execution. It also facilitates to map virtual address ranges to certain physical address ranges.

Macro Definition Documentation

#define CORETLB(CID, args...) TLBentry entries_##CID[] = {args, ENTRY(0, 0, 0, 0)}

Define a new TLB entry table for a specific core.

Defines all TLB entries for a specific core. For each entry the **ENTRY** Macro shall be used. The MMU of the corresponding core is configured during boot-up phase.

Example:

```
#include <stdio.h>
#include "kernel_lib.h"

// The following lines are provided by the developer/system integrator.

// Adds an entry for core 0: virtual address range 0x10000000 - 0x10FFFFFF is
// mapped to physical address range 0x80000000 - 0x80FFFFFF for data load and
// instruction fetch access.
CORETLB(0, ENTRY(0x10, 7, 0x80, 2));

// Adds two entries for core 1:
// 0x0       - 0xFFF     to 0x40000    - 0x40FFF    full access
// 0x6000000 - 0x6000FFF to 0x10000000 - 0x10000FFF read only access
CORETLB(1, ENTRY(0x0, 1, 0x40, 3), ENTRY(0x6000, 1, 0x10000, 0));

// This line is provided by the OS, so the developer has to ensure that all
// referenced entries exist!
TLBentry_t *mappings[] = {entries_0, entries_1, NULL};

int main(void) {
    int i, j;
    i=0;
    while (mappings[i] != NULL) {
        printf("Evaluating mapping table %d\n", i);
        j=0;
        while (mappings[i][j].a != 0) {
            printf("\tEvaluating mapping entry @(%d,%d): {%d,%d}\n",
                i, j, mappings[i][j].a, mappings[i][j].b);
            j++;
        }
        i++;
    }
```

```
    return 0;
}
```

Parameters:

CID	- core ID specifies a certain core.
args	- argument list of TLB entries for the specified core. The *ENTRY* Macro is used to add an single entry into the table.

#define ENTRY(_epn, _size, _rpn, _ap) {.epn = _epn, .size = _size, .rpn = _rpn, .ap = _ap}

Define a new TLB entry.

The Macro is used inside the **CORETLB** Macro and can not be used stand alone. A TLB entry supports 8 different page sizes ranging from 1 KB to 16 MB.

Parameters:

_epn	- effective page number together with the page size results in the start address of a virtual address range.
_size	- page size is the memory size which is mapped from virtual to physical addresses. Legal values are: • 0 - 1 KB • 1 - 4 KB • 2 - 16 KB • 3 - 64 KB • 4 - 256 KB • 5 - 1 MB • 6 - 4 MB • 7 - 16 MB
_rpn	- real page number together with the page size results in the start address of a physical address range.

_ap	- access privileges define how the specified address range can be accessed. By default every address is readable by a data load. Access for data store and instruction fetch can be granted via access privileges. The four legal values are: • 0 - read only access for data load • 1 - read/write access for data load/store • 2 - read access for data load and instruction fetch • 3 - full access - read/write access for data load/store and read access for instruction fetch

#define NTLB 64

Defines the number of TLB entries.

By default, there are 64 TLB entries supported. Regarding to the TLB hardware implementation this value has to be adapted.

Synchronization Mechanisms

Typedefs

- typedef ticketlock_t **spinlock_t**

  Type for spin-lock variable.

- typedef uint32_t **barrier_t**

  Type for barrier variable.

Functions

- static uint32_t **fetch_and_add** (uint32_t *addr, int32_t val)

  Atomic Fetch-and-Add operation.

- static void **spin_init** (**spinlock_t** *lock)

  Initialization for spin-lock.

- static uint8_t **spin_lock** (**spinlock_t** *lock)

  Spin-lock function.

- static uint8_t **spin_unlock** (**spinlock_t** *lock)

  Spin-unlock function.

- static void **barrier_wait** (volatile **barrier_t** *barrier, uint32_t nr_of_threads)

  Barrier for process synchronization.

Detailed Description

The synchronization mechanisms provide basic techniques for synchronization. They can be used to implement more complex synchronization mechanisms such as mutex locks. The spin lock synchronization is mapped to a ticket lock implementation, which uses an atomic fetch and add instruction and thus is fair in the sense of access order to the critical section.

Typedef Documentation

typedef uint32_t barrier_t

Type for barrier variable.

Type definition for declaring a barrier variable.

typedef ticketlock_t spinlock_t

Type for spin-lock variable.

Type definition for declaring a spin-lock variable.

Function Documentation

*static void barrier_wait (volatile barrier_t ** **barrier,** *uint32_t* **nr_of_threads***)* `[inline]`, `[static]`

Barrier for process synchronization.

Parameters:

barrier	- specifies the barrier variable to which processes synchronize.

nr_of_threads	- number of threads synchronizing to the barrier.

*static uint32_t fetch_and_add (uint32_t ** **addr,** *int32_t* **val)** *[inline], [static]*

Atomic Fetch-and-Add operation.

Loads the value from the specified memory location *addr* into a register, adds the value *val* and stores the modified value back to same memory location *addr* . The three steps are indivisible and executed atomically.

Parameters:

addr	- specifies the memory address of the variable to be modified.
val	- specifies the value to be added to the variable to be modified.

Returns:

the unmodified value loaded from memory location *addr* .

*static void spin_init (spinlock_t ** **lock)** *[inline], [static]*

Initialization for spin-lock.

Initializes the lock variable for a critical section.

Parameters:

lock	- specifies the lock variable for exclusive access.

*static uint8_t spin_lock (spinlock_t ** **lock)** *[inline], [static]*

Spin-*lock* function.

The spin-lock function provides a busy waiting software synchronization technique. It gains access to a critical section in a fair manner (i.e. FIFO order) to all participants.

Parameters:

lock	- specifies the lock variable to which exclusive access is requested.

Returns:

zero on success, if the *lock* is acquired.

*static uint8_t spin_unlock (spinlock_t ** **lock)** *[inline], [static]*

Spin-*unlock* function.

Releases the *lock* to a critical section gained by *spinLock* .

Parameters:

lock	- specifies the lock variable which is released.

Returns:

zero on success, if the *lock* is released.

Data Structure Documentation

context_t Struct Reference

Structure used for context switching.

```
#include <kernel_lib.h>
```

Detailed Description

Structure used for context switching.

The documentation for this struct was generated from the following file:

- **kernel_lib.h**

regs Struct Reference

Structure containing the registers involved in context switching.

```
#include <kernel_lib.h>
```

Detailed Description

Structure containing the registers involved in context switching.

The documentation for this struct was generated from the following file:

- **kernel_lib.h**

TLBentry Struct Reference

Structure of a TLB entry.

```
#include <kernel_lib.h>
```

Detailed Description

Structure of a TLB entry.

The documentation for this struct was generated from the following file:

- **kernel_lib.h**

File Documentation

kernel_lib.h File Reference

Contains type definitions, macros and function declarations for accessing hardware dependent functionalities.

```
#include <stdint.h>
```

Data Structures

- struct **regs**
- *Structure containing the registers involved in context switching.* struct **context_t**
- *Structure used for context switching.* struct **TLBentry**

Structure of a TLB entry. Macros

- #define **PROGRAM_IRQ_ILLEGAL** 0x8000000

  Mask for exception syndrome.
- #define **PROGRAM_IRQ_PRIVILEGED** 0x4000000

  Mask for exception syndrome.
- #define **PROGRAM_IRQ_TRAP** 0x2000000

  Mask for exception syndrome.
- #define **PROGRAM_IRQ_UNIMPLEMENTED** 0x1000000

  Mask for exception syndrome.
- #define **NTLB** 64

  Defines the number of TLB entries.
- #define **RO_DATA** 0

  Constant for TLB entry read only access for data
- #define **RW_DATA** 1

  Constant for TLB entry read/write access for data
- #define **RO_DATA_INS** 2

  Constant for TLB entry read only access for data and instructions

- #define **RW_DATA_INS** 3

  Constant for TLB entry read/write access for data and read access for instructions
- #define **ENTRY**(_epn, _size, _rpn, _ap) {.epn = _epn, .size = _size, .rpn = _rpn, .ap = _ap}

  Define a new TLB entry.
- #define **CORETLB**(CID, args...) **TLBentry** entries_##CID[] = {args, **ENTRY**(0, 0, 0, 0)}

  Define a new TLB entry table for a specific core.
  - System Call Macros#define **_syscall0**(type, name)

    System Call without arguments.
  - #define **_syscall1**(type, name, type1, arg1)

    System Call with 1 argument.
  - #define **_syscall2**(type, name, type1, arg1, type2, arg2)

    System Call with 2 arguments.
  - #define **_syscall3**(type, name, type1, arg1, type2, arg2, type3, arg3)

    System Call with 3 arguments.
  - #define **_syscall4**(type, name, type1, arg1, type2, arg2, type3, arg3, type4, arg4)

    System Call with 4 arguments.
  - #define **_syscall5**(type, name, type1, arg1, type2, arg2, type3, arg3, type4, arg4, type5, arg5)

    System Call with 5 arguments.

Typedefs

- typedef struct **regs regs_t**

  Structure containing the registers involved in context switching.
- typedef
- RTE_SPECIFIC_TASK_IDENTIFIER **task_identifier_t**

  Type of task identifier.
- typedef struct **TLBentry TLBentry_t**

  Structure of a TLB entry.
- typedef ticketlock_t **spinlock_t**

  Type for spin-lock variable.

- typedef uint32_t **barrier_t**

  Type for barrier variable.

Functions

- static void **enable_external_irq** ()

  Enable external interrupts.

- static void **disable_external_irq** ()

  Disable external interrupts.

- void **clear_external_irq** ()

  Activate external interrupts.

- static uint32_t **save_external_irq** ()

  Save external interrupt status flag.

- static void **restore_external_irq** (uint32_t msr)

  Restore external interrupt flag.

- uint64_t **get_time_base** ()

  Get the time base measured in clock cycles.

- void **set_time_base** (uint64_t time)

  Set the time base.

- void **enable_pit** (uint32_t interval)

  Enable the programmable interval timer.

- void **disable_pit** ()

  Disable the programmable interval timer.

- void **clear_pit** ()

  Clear the programmable interval timer.

- **context_t** * **init_context** (void *sp, uint32_t size, void(*func)(void *))

  Initialize a context for use with context switching.

- void **switch_context** (**context_t** **oldctx, **context_t** *newctx)

  Switch from the actual context to a new context.

- void **ctx_switch_hook** (**task_identifier_t** *task_id)

Declaration of context switch hook routine.

- static uint32_t **fetch_and_add** (uint32_t *addr, int32_t val)

  Atomic Fetch-and-Add operation.

- static void **spin_init** (**spinlock_t** *lock)

  Initialization for spin-lock.

- static uint8_t **spin_lock** (**spinlock_t** *lock)

  Spin-lock function.

- static uint8_t **spin_unlock** (**spinlock_t** *lock)

  Spin-unlock function.

- static void **barrier_wait** (volatile **barrier_t** *barrier, uint32_t nr_of_threads)

  Barrier for process synchronization.

  - Custom Interrupt Handlervoid **_irq_program_handler** (uint32_t esr)

    Defines a custom program interrupt handler.

  - uint32_t **_irq_sys_handler** (uint32_t arg1, uint32_t arg2, uint32_t arg3, uint32_t arg4, uint32_t arg5, uint32_t scno)

    Defines a custom system call handler.

  - void **_irq_timer_handler** (void)

    Defines a custom programmable interval timer interrupt handler.

  - void **isr_pre_hook** (uint32_t cause)

    Declaration of ISR pre hook routine.

  - void **isr_post_hook** (uint32_t cause)

    Declaration of ISR post hook routine.

Detailed Description

Contains type definitions, macros and function declarations for accessing hardware dependent functionalities.

Author:

Christian Bradatsch

Index