Universität Augsburg

System-Level Software for a Multi-Core MERASA Processor

Florian Kluge, Julian Wolf

Report 2009-17 Oktober 2009

Institut für Informatik

D-86135 Augsburg

Contents

Abstract

In the EC FP-7 MERASA project a hard real-time capable multi-core processor is developed. The system-level software represents an abstraction layer between application software and embedded hardware. It has to provide basic functionalities of a real-time operating system.

This report presents requirements for a multi-threaded and multi-core hard real-time capable system software in embedded systems and the transfer to the implemented MERASA multi-core processor showing details of the thread management, the dynamic memory management and the resource management. It summarises the full MERASA system-level software developed for a multi-core processor running on the MERASA SystemC Simulator.

1. Introduction

The main objective of the MERASA[1] project is the development of a multi-core processor for hard real-time embedded systems. Simultaneously there is a need for timing analysis techniques and tools to guarantee the analyzability and predictability of the features provided by the processor. The MERASA system-level software provides a fundament for application software running on such a processor. A verification of the contained features will be achieved by an integration into pilot studies.

The challenge in this software field is to guarantee an isolation of memory and I/O resource accesses of various hard real-time threads running on different cores to avoid mutual and possibly unpredictable interferences between hard real-time threads. The intent of this isolation is also to enable an effective WCET analysis of application code. The resulting system software should execute hard real-time threads in parallel on different cores of a multi-core MERASA processor or within different thread slots of simultaneously multithreaded MERASA cores. These hard real-time threads will potentially run in concert with additional non real-time threads of mixed application workload. Currently, the development of the first version of the MERASA multi-core processor is finished. It was adapted from a simultaneous multi-threaded (SMT) MERASA core processor developed based on the SMT CarCore processor [7] which is binary compatible to the Infineon TriCore.

This report describes the full system-level software which provides functionalities for a MERASA multi-core processor with a hardware-based two-level scheduler. During the next stage of the MERASA project, it will be enhanced by the integration of drivers to support pilot studies on the FPGA prototype.

This report is organised as follows: Section 2 gives an overview of requirements arising for a multi-core real-time operating system. In section 3 we present an architectural overview, followed by a short description of the user interface in

[1]**M**ulti-Core **E**xecution of Hard **R**eal-Time **A**pplications **S**upporting **A**nalysability, a STREP project within the Seventh Framework Programme of the European Union

section 4. Section 5 shows the implementation of the single parts developed to accomplish these requirements. Section 6 concludes this paper. The annex finally shows the detailed information on the user interface.

2. Requirements

In this section, we state the minimum requirements for a *Real-Time Operating System* (RTOS) for embedded systems with simultaneous multithreaded (SMT) and multi-core hardware, and show the basic properties that have to be fulfilled.

2.1. Functional Requirements

In general, an operating system (OS) makes the usage of computer hardware possible. It provides an interface to access system resources like memory, I/O devices and to manage the execution of tasks. So we can summarise the common requirements:

- The OS has to manage processes and schedule processor time.
- Memory for the applications must be allocated and controlled.
- The OS must control and manage the connected devices.
- In case of errors and interrupts, they must be handled by the OS.

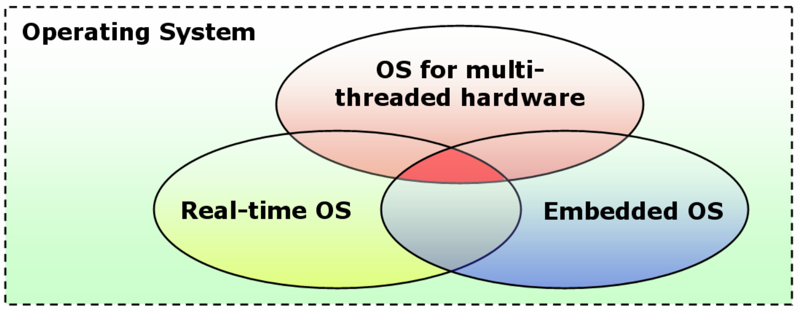


Figure 1: Combination of requirements for different OS types

Operating systems can be categorised into different classifications. So we can distinguish between single- and multi-user as well as single-threaded and multi-threaded systems. Depending on response time or execution mode, we can find real-time and non-real-time, embedded- or general-purpose computing operating systems. As our objective is the development of an RTOS for embedded systems with multithreaded and multi-core hardware, we need a combination of different fields. Figure 1 shows a symbolic intersection of the different fields of requirements and how they must be mixed together. So we first take a look at the concept of an RTOS and then add aspects regarding both embedded and SMT systems.

Concept of RTOSs

The key difference between general-purpose operating systems and real-time operating systems is the need for a deterministic timing behaviour. All operating system services have to consume only known and expected amounts of time. It is not allowed that a task causes random delays and makes an application miss real-time deadlines. So most RTOSs do their task scheduling using a scheme called *priority based preemptive scheduling*. Each task is assigned a priority, with higher values representing a need for quicker execution. The *preemptive* nature of the task scheduling enables a fast responsiveness. The scheduler is allowed to stop a task's execution, if another task needs to run immediately.

Regarding [1] we can summarise the general facets making an OS an RTOS:

- The RTOS has to be multi-threaded and preemptible.
- Either the notion of thread priority exists, or the RTOS provides a deadline driven scheduler.
- The RTOS supports predictable thread synchronization mechanisms, especially a system of priority inheritance.
- The timing behaviour of the RTOS should be known and predictable.

OSs in embedded environments

Embedded systems are mostly not recognizable as computers, instead they are hidden inside cars, aeroplanes or everyday objects surrounding and helping us in our live. A high-level connectivity to the environment through sensoric interfaces providing context data is typical.

The characteristical operation of embedded systems is limited by computer memory and processing power. The services they provide to their users are usually constrained by strict time deadlines. When using an OS in embedded environments, we also have to regard these restrictions on memory and performance.

So, we can outline also the requirements implicated from the field of embedded computing:

- The embedded OS must be very time and memory efficient.
- The OS has to be compact and concentrate on the most necessary functions.

OSs on SMT and multi-core hardware

Simultaneous multithreading (SMT) is the ability to concurrently run programs divided into subcomponents or threads on a single processor or within a processor core. While the SMT execution is only apparently parallel, multi-core hardware offers real parallelism. However, both mechanisms promise better utilization of processors and other system resources. As a result they provide a scalable, modular environment upon which it is appropriate to write application software. Working with several tasks in parallel, a multi-threaded or multi-core hardware can also cause a lot of new potential bugs to be introduced into an application. So we can add as specific requirement (see [4]) that the OS has to avoid race conditions or deadlocks caused by timing problems.

2.2. Requirements for the user interface

To facilitate the development of applications, the interface of the operating system should be geared towards a common standard. As POSIX [5] is widely used also in the fields of embedded real-time systems (e.g. QNX [6]), it is useful to support functionalities following this standard. Thus, one can provide a familiar handling of parameters, return values and function names. It will be easy to port applications developed for other systems using the POSIX interface as well (see sect. 4.5).

On basis of these requirements concerning both functionalities and the user interface, we are now able to propose an architecture for the MERASA system level software that fulfills a combination of concepts concerning a RTOS for embedded environments with multithreaded and multi-core hardware.

3. Architectural overview

The design of the MERASA system-level software joins several well-known OS techniques. The basic kernel comprises the most important management functionalities following the microkernel principle. Additional functions may run outside this kernel as seperate components.

Figure 2 gives an overview of the proposed architecture. The core of the MERASA system-level software contains three components: the Thread Management, the Dynamic Memory Management and the Resource Management. These three parts run on top of the proposed MERASA *Embedded Control Unit* (ECU). To give consideration to the real-time requirements the system-level software uses pre-allocation techniques. In an initial phase, all resources, which may potentially cause non real-time behaviour, are allocated. So when the application starts running for all resource accesses real-time behaviour can be guaranteed. The next section will show in more detail how the application programming interface can be accessed. Section 5 describes the working of the kernel parts and especially how the real-time execution is ensured.

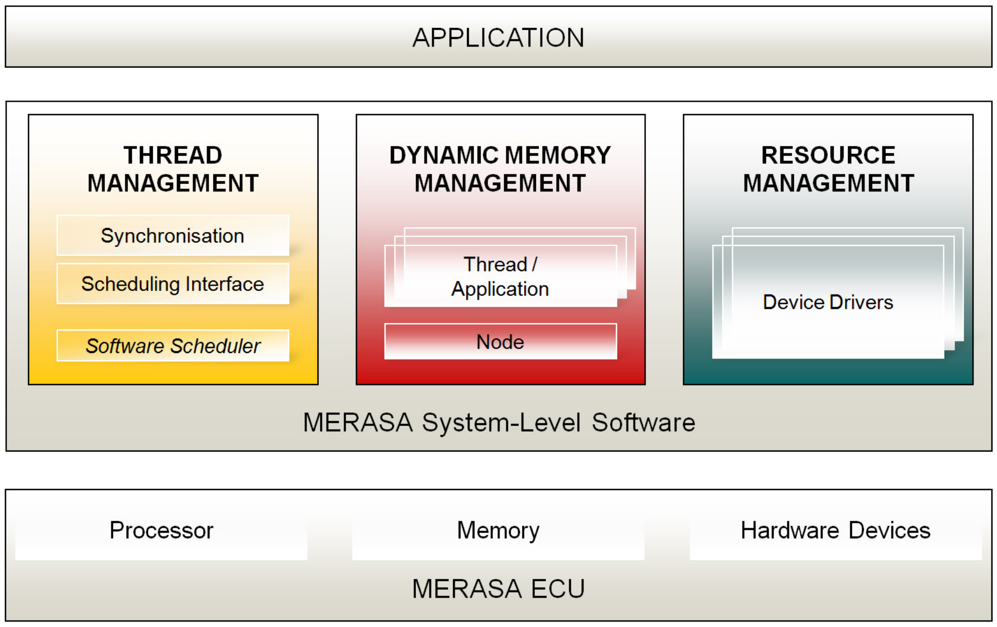


Figure 2: Architecture of the MERASA system-level software

4. Application Programming Interface

Here we describe the programming interface to access the MERASA system-level software (directory `include/`). This section is a short guide for application programmers to know where to find necessary functionalities, whereas the full specification is confined to the annex. In the first three parts of this section we specify the functions of the basic architectural components. The last part explains some common header files.

4.1. Thread Management

The Thread Management is implemented as a subset of POSIX functions:

pthread.h This header provides essential functions for the Thread Management. It allows the creation of threads and the management of the different hardware thread types (see sect. 5.1) and their scheduling parameters. Also, thread privileges are defined here as a base for a security manager. Moreover, it contains an interface to access the thread synchronization mechanisms providing functions for mutex, conditional and barrier variables.

4.2. Dynamic Memory Management

For the usage of the Dynamic Memory Management it is necessary to utilise the definitions from the following header files:

memory.h This header represents the basic file of the memory management. It provides methods to allocate a specified amount of memory or to free previously allocated blocks of the current thread. As well, one can perform an copy of non-overlapping memory sections to another address.

memory-desc.h The Dynamic Memory Management of the MERASA system-level software supports the usage of different types of memory. This file provides functionalities to notify the Dynamic Memory Management about the availability of memory hardware.

4.3. Resource Management

One basic task of the MERASA system-level software is to enable the usage of computer hardware. To get a high level of flexibility, the hardware resources are accessed through device drivers. The interface to these drivers and the resource management are defined in the following files:

driver.h This header provides macros and data types to write an individual device driver for the MERASA system-level software. So it can be ensured that the compiled drivers have the correct file format and layout structure.

drivermanager.h This file enables the management of the device drivers. It contains functions to install and remove drivers from the system and for an access of the drivers' functions.

fcntl.h This header provides a function to open a device in the MERASA resource manager.

stropts.h Here, a function is included to control operations on a specific device.

unistd.h This file contains functions to read and write from devices.

A special resource currently included in the MERASA system-level software is the *Virtual Output*. It implements functions very similar to `stdio.h` with different format modifiers for various data types. In the MERASA simulator the output is written to `STDOUT`, prefaced with some special characters (%#). This enables an easy way to debug applications but will influence the timing behaviour of the application. For this reason, the virtual output also provides *fast-logging facilities* to output just few bytes of data. These functions come with less and predictable timing overhead, requiring less than ten processor instructions.

log.h Especially for debugging purpose using the the virtual output this header gives several easy logging facilities. The output is thread safe and can be written to a log file or to `STDOUT`. There are five predefined loglevel-stages making it easy to distinguish between negligible debugging output, interesting warnings and important fatal errors. The level is set in the makefile (`config/defines.mk`) for compilation.

vo.h This header defines the memory regions for the virtual output.

4.4. Common header files

Besides the interface to the three main parts of the MERASA system-level software, there are several common header files:

config.h This file includes the general configuration of the MERASA system-level software. In many global definitions you can set the details, e.g. the number of used cores, the basic configuration of scheduling, the adresses of used thread memory etc.

error.h This header contains definitions of error numbers and error types. These are equal to the error number definitions defined by the POSIX standard.

sysmonitor.h Here, monitoring functionalities of system parameters are provided. It is possible to get detailed statistics on the usage of global and each thread's memory.

sys/types.h In this header platform-specific definitions of basic data types are defined.

merasa-ssw.h This file includes all MERASA system-level software headers.

4.5. POSIX interface

In compliance with the MERASA partners a subset of the POSIX interface was implemented, including basic functionalities for thread creation (with different attributes for scheduling) and thread join. Listing 1 shows how to use the interface from application side. Here, one hard real-time (HRT) thread executing function `f1` and one non real-time (NRT) thread executing `f2` is initialised. The procedure `finish_thread_init()` signals the end of the initialisation phase to the system software, and the execution of all threads will start just then. Finally, the `join` methods are called to suspend the calling thread and wait for both other threads to finish execution.

```
pthread_t my_hard_rt_thread;
pthread_t my_non_rt_thread;
pthread_attr_t my_hard_rt_attr;
pthread_attr_t my_non_rt_attr;

pthread_attr_setschedpolicy(&my_hard_rt_attr, SCHED_HRT);
pthread_create(&my_hard_rt_thread, &my_hard_rt_attr, &f1, NULL);

pthread_attr_setschedpolicy(&my_non_rt_attr, SCHED_NRT);
pthread_create(&my_non_rt_thread, &my_non_rt_attr, &f2, NULL);

finish_thread_init();

pthread_join(&my_hard_rt_thread);
pthread_join(&my_non_rt_thread);
```

Listing 1: Example for using the POSIX interface from application side

Moreover, also the synchronisation mechanisms are implemented following the POSIX standard. By this, it is easy for application programmers to use the commonly known functionalities for mutex, conditional and barrier variables as we provide a familiar handling of parameters, return values and function names.

Besides the thread management, the resource management was enhanced covering a subset of the commonly used POSIX interface. We implemented basic operations like the generic `read` / `write` functions as well as driver-specific access by `open` / `close` and configuration by `ioctl` in a POSIX-compliant way.

5. Implementation

This section describes the implementation of the architectural parts of the full MERASA system-level software in more detail.

5.1. Thread Management

Based on the previously delivered MERASA SMT single-core, the main task in the second project year was the extension of the thread management to support the creation and maintenance of hard real-time (HRT) and non real-time (NRT) threads on different cores.

A two-level hardware scheduler distributes threads on the one hand over the different cores and on the other hand over the different available thread slots per core. So it is a basic goal of the system-level software to provide a solid scheduling interface, which enables an easy creation, suspension and un-suspension of specific threads from application side. This is especially reached by preserving compatibility to the commonly used POSIX interface. On the other side, the scheduling interface internally builds a correct and consistent baseline for the hardware by managing the lists of HRT and NRT threads.

Moreover, it is necessary to fulfil the requirements on synchronisation. For this reason, the system software provides commonly used mechanisms like mutex, conditional and barrier variables. All these mechanisms are also implemented following the POSIX standard.

Scheduling Interface

In order to support the full MERASA multi-core SystemC simulator containing a two-level scheduler, the scheduling interface had to be adapted and extended.

First it is necessary to take a deeper look into the organisation of threads within system-software and simulator. Every HRT and NRT thread has its own specific "Thread Control Block" (TCB), which is an array of 256 bytes. In this array, the whole thread context, like used data and address register values, synchronisation and scheduling information is stored. All TCBs are located in segment `0x90000000` and aligned to 256 byte boundaries, i.e. TCB 1 starts at `0x90000100`, TCB 2 at `0x90000200`, etc.

TCB 0 is reserved for special data used by the hardware scheduler, it contains the heads of two lists, one for the HRT threads (a 32 bit pointer at `0x90000030`) and one for the NRT threads (a 32 bit pointer at `0x90000014`). On thread creation these two lists are updated and extended by the system-level software by setting the pointers `sched_next` and `sched_prev` of every TCB to the next respectively previous to schedule TCB. As a result we get two double linked lists of TCBs as shown in figure 3.

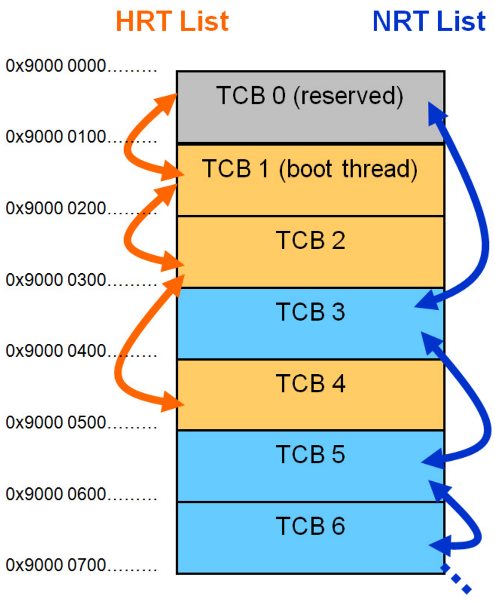


Figure 3: Two linked lists of TCBs, ready for the scheduler

As the progress of thread creation cannot guarantee timing constraints, it was decided to introduce an initial phase, where only one boot thread is running on the first core. It is possible to create new HRT and NRT threads only during this phase,. The system software then fulfils all preparation jobs which are bad for real-time behaviour, like memory allocation to guarantee timing correctness during the following execution phase. From application side, it is very important to signal the end of the initialisation phase by calling exactly once the function `finish_thread_init()`. After this call no further creation of threads is possible, all needed threads must have been created during the initialisation phase.

The call to finish the initial phase and to start thread execution internally writes the heads of the HRT and NRT to the specified addresses of the reserved TCB 0. This write access is snooped by the hardware scheduler, which in turn now stats operation. It begins traversing first the HRT list and puts each TCB onto another core, each in thread slot 0. This continues until the list is terminated (by 0) or the maximum number of cores is reached. There is one special case: as slot 0 of core 0 is the boot thread (the only one active after a reset, the head of the HRT list must always point to TCB 1 (the boot thread) and then to the next HRT thread.

A write to the head of the NRT list, which is also triggered by finishing the initial phase, also invokes the scheduler: it starts traversing the NRT list, in detail it reads the first NRT thread's TCB from the specified memory location and writes it to core 0, thread slot 1. Following `sched_next` it puts the next threads into thread slot 1 of core 1 to `MAX_CORES`. If the maximum number of cores is reached, it continues with core 0 and thread slot 2, and so on. In figure 4, the distribution of the different TCBs over the slots is shown in more detail.

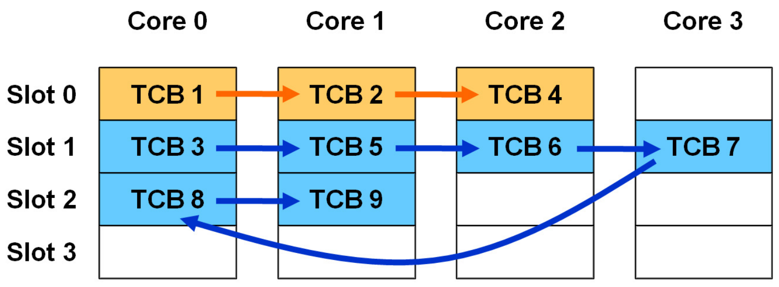


Figure 4: Distribution of TCBs over available thread slots

As already mentioned above, the developed hardware scheduler consists of two levels: the fixed priority (FP) scheduler in the issue stage of the pipeline and the initialisation logic to distribute threads over the slots in all cores. The FP scheduler is quite simple: there are 4 slots per core and the issue logic chooses the thread with the highest priority that is ready. The HRT thread always has the highest priority within one core. The second level of the hardware scheduler is the initialisation logic, which provides an interface to tell the cores where to begin the program execution. It sets the program counters and register values for each slot in every core. In order to optimise the scheduling, it is possible to additionally use a software scheduler within the thread slots.

Synchronization

The second basic part of the thread management of the MERASA system-level software besides the scheduling interface is the provision of synchronisation mechanisms. Due to the enhancement of the simulator to the multi-core version, but also due to requirements from application side, it was necessary to adapt these mechanisms.

As a simple base, already in the basic system-level software a spinlock mechanism was implemented, which can be used to give only one thread access to a critical region. If another thread tries to access the same region, it performs "busy waiting" until the lock is free. Internally, the implementation of spinlocks is based on the atomic swap instruction. But as the spinlock mechanism occupies full processor power while waiting, it should be avoided from application side or utilised only for extremely short critical regions.

To lock a critical region, the usage of mutex variables is preferable. Although the implementation is internally based on spinlocks, they cover only a very short critical region. Whenever a thread has to wait for a lock to become free, it is suspended from execution and inserted to a waiting list connected to the specific mutex variable. As soon as the mutex is unlocked, the suspended threads are reactivated following the waiting list. To ensure some degree of fairness, the waiting list is managed following a first-in-first-out (FIFO) mechanism with preferred HRT threads (if multiple HRT threads are in the list, they are also handled according to FIFO). The waiting lists are implemented considering constant access time, on insertion as well as on deletion of elements.

From a hard real-time point of view, it does not make sense to use shared locks for HRT and NRT threads, because a HRT thead may have to wait for a NRT thread holding a lock. As NRT threads do not have any timing guarantees, the HRT thread might be delayed unpredictably and therefore loses his property of being a HRT thread. However, although an important point in the coding guidelines for application developers is the avoidance of common mutex variables for HRT and NRT threads, it appears to be useful especially for debugging purpose. As we implemented a “virtual output” in the MERASA SystemC simulator which can display debugging messages from different threads on command line, it is necessary to use at least temporary one common mutex for the screen output.

In order to give a good solution for thread communication, the thread management of the MERASA system software provides condition variables. By calling a wait on a conditional variable, a thread is suspended and registered to a waiting list. This progress is very similar to the waiting on a locked mutex variable, as described above. As soon as the signal function is called, the first thread of the waiting list is unsuspended, on broadcast all waiting threads are unsuspended and continue execution.

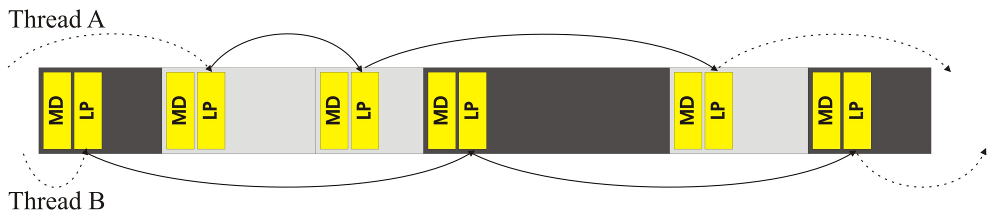
Finally, in order to guarantee a synchronised start of a specific section of different threads, the thread management provides barriers. In an initialisation function, the application programmer can set the number of threads on which the barrier should wait. Every time a thread calls the wait function, a counter, connected to the initialised barrier is incremented and the thread is suspended from execution. The suspension itself is based on condition variables. As soon as the incremented counter reaches the value which was set on initialisation, all threads in the waiting list get a signal, to synchronously continue execution.

5.2. Dynamic Memory Management

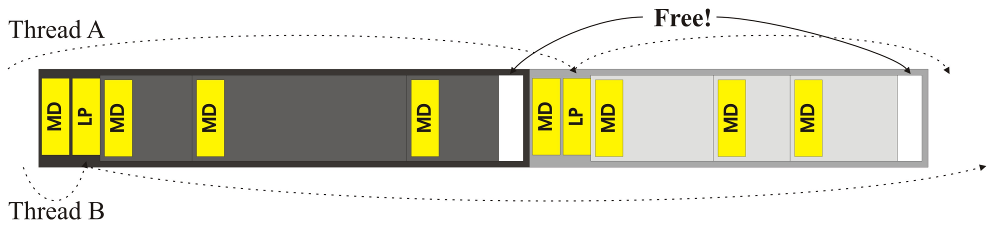
In contrast to most traditional management systems, it is necessary for our memory management to provide timing guarantees. So we introduce a two-layered memory management and the use of memory pre-allocation. By this means our objective is to minimise interferences of several threads among each other and provide higher flexibility for applications at the same time.

On the first layer – the *node* level – large blocks of memory are allocated. This allocation is performed in a mutually exlusive way to keep the state of memory consistent, so here a blocking of threads can occur. But as this is only done in the initialisation phase before threads are started, influences on the real-time behaviour of the system can be neglected. On the *thread* layer the memory management allocates memory to the executed program in the specific thread. This can be done in real-time without locking, because the memory is taken from the blocks pre-allocated in the node level – exclusively for the thread.

Alongside we can see another advantage of such a two-layered architecture in figure 5. As it is always necessary to keep the information, which memory block belongs to which thread, a lot of management data is needed by putting them into a linked list including list pointers (LP). In contrast to the conventional (one-layered) allocation scheme, our list pointers need only be added to the large blocks on node level, as shown in 5(b). Apparently, even in this simple example some memory can be saved and the management data needed to keep track of the owners is reduced.



(a) Conventional memory management; the blocks of the threads are highly mixed



(b) Two-stage memory management, the threads' memory is kept separated

Figure 5: Example layout of used memory with two threads; MD: Management data of the memory allocator, LP: List Pointers to keep track of thread's memory

When a thread finishes operation and its resources need to be cleaned up, the two-layered architecture also has its advantages. Only few large blocks must be deallocated by the node management. The internal structure of these blocks can be ignored. Besides, external memory fragmentation is reduced at least on the node level.

Regarding the implementation, dynamic memory management on the node level is currently performed by an allocator based on Lea's allocator [2] (DLAlloc). On the thread level, the user can choose between various implementations of memory allocators. For non-real-time applications, efficiency of memory usage can be improved by a best-fit allocator like DLAlloc. This variant is fast and space-conserving but hardly real-time capable. If a real-time application requires the flexibility of dynamic storage allocation, an allocator with bounded execution time can be used. The Two-Level Segregate Fit (TLSF) allocator, as introduced by [3], is a general purpose dynamic memory allocator specifically designed to meet real-time requirements. Using this alternative, the computation of worst-case execution time (WCET) is simplified. Whereas in DLAlloc, the execution time depends on the current state of the allocator and on the previous de- / allocations, TLSF provides a bounded execution time regardless of its former operation at the cost of higher internal fragmentation.

In general, the memory management supports different types of memory. It provides functionalities to define the configuration, i.e. beginning, end, length and the cost for the use. By this means, a high flexibility of memory structures will be achieved.

5.3. Resource Management

It is a main task of the system-level software to enable the usage of computer hardware. As already mentioned, the MERASA system-level software follows the microkernel principles, i.e. manages only the most essential system resources like processing time and memory. To gain maximum flexibility, hardware devices are accessed through device drivers. These resources are managed by a dedicated *Resource Management* unit.

The implementation of the resource management, containing a driver manager, is a subset of the POSIX standard [5]. It contains the generic `open` / `close` operations as well as functions to access the device (`read` / `write` operations) and for configuration (`ioctl`). Valid configuration values and further parameters depend on the specific device and driver.

In the Resource Management the problem of concurrent use of devices can be reduced to thread synchronization. For this, the Thread Manager already provides solutions. In general, there is no limitation on the number of drivers supported by the resource manager. However, the timing behaviour depends on this quantity. For the use in real-time applications, the number of drivers must be limited to guarantee device accesses in bounded time. As the number of devices an application uses is known in advance, constant-access-time handlers can be arranged during the preparation of the application's execution environment. Thus, device accesses can be performed in constant time.

6. Conclusion

In this report we presented the full system-level software for a multi-core MERASA processor. It represents an abstraction layer between application software and hard real-time capable multi-core hardware. The architecture consists of three main parts: The *Thread Manager* provides an interface to the hardware scheduler of the MERASA core processor, mechanisms for thread synchronization and support for a software scheduler. The *Dynamic Memory Management* minimises interferences of different threads by providing a flexible two-layered memory management with memory pre-allocation. Finally, the *Resource Management* enables the use of peripheral device drivers.

Regarding the different classes of requirements stated in section 2 we can summarise how they are fulfilled in detail: The MERASA system-level software is multi-threaded and preemptible. For thread synchronization it provides lock (mutex), conditional and barrier variables. As a real-time capable scheduler is already implemented in the hardware, the system-level software only guarantees a correct management of scheduling parameters. The timing behaviour is known and predictable. All time-critical operations are executed in an initialisation phase which is independent from the real execution with timing guarantees. The system-level software is very compact because it concentrates on necessary functionalities and provides a fast and efficient way of execution. As the two-level dynamic memory management of the MERASA system-level software keeps the threads' memory separated on node level, the management effort of the chunks is reduced. Concerning the multithreaded hardware, the software provides mechanisms for synchronization. So it is easy for an application programmer to avoid race conditions between different threads running in parallel.

The support of pilot studies with the FPGA marks one central task of the third year of the MERASA project. The requirements for data communication will be investigated. Then it will be necessary to define a detailed interface to the FPGA and to integrate drivers for the peripheral components. Finally, the system-level software will allow using the FPGA prototype of the MERASA multi-core in evaluation environments of interested industrial partners.

Finally, the pilot studies will be continuously supported and debugged. If experiences make it necessary, adaptations can be added and some fine tuning of the whole MERASA system-level software will be done, in order to get the best possible solution as project outcome.

A. Data Structure Documentation

A.1. driver Struct Reference

`#include <driver.h>`

Data Fields

- void ∗ **pi**
- const char ∗ **name**
- **version_t version**
- **drv_init_fn_t init**
- **drv_cleanup_fn_t cleanup**
- **drvif_t** ∗ **ops**
- **pthread_mutex_t lock**
- **thread_handler owner**
- struct **driver** ∗ **sp_next**

A.1.1. Detailed Description

This struct describes a device **driver** (p. 19). Do not use it directly, instead publish your **driver** (p. 19) using the **SDRIVER** (p. 40) macro!

Definition at line 89 of file driver.h.

A.1.2. Field Documentation

A.1.2.1. void∗ driver::pi

Process image, only for internal use.

Definition at line 90 of file driver.h.

A.1.2.2. const char∗ driver::name

Name of the **driver** (p. 19).

Definition at line 91 of file driver.h.

A.1.2.3. version_t driver::version

Version of this **driver** (p. 19). This field is currently not used, it should be set to 0x10.

Definition at line 92 of file driver.h.

A.1.2.4. drv_init_fn_t driver::init

Initialisation function.

Definition at line 93 of file driver.h.

A.1.2.5. drv_cleanup_fn_t driver::cleanup

Cleanup function; currently unused.

Definition at line 94 of file driver.h.

A.1.2.6. drvif_t∗ driver::ops

Operations struct.

Definition at line 95 of file driver.h.

A.1.2.7. pthread_mutex_t driver::lock

Lock for this **driver** (p. 19).

Definition at line 96 of file driver.h.

A.1.2.8. thread_handler driver::owner

Current owner of this **driver** (p. 19), -1 if unused.

Definition at line 97 of file driver.h.

A.1.2.9. struct driver∗ driver::sp_next `[read]`

Pointer for a thread's **driver** (p. 19) stack.

Definition at line 98 of file driver.h.

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

- **driver.h**

A.2. driver_interface Struct Reference

`#include <driver.h>`

Data Fields

- **drv_open_fn open**
- **drv_close_fn close**
- **drv_read_fn read**
- **drv_write_fn write**
- **drv_ioctl_fn ioctl**

A.2.1. Detailed Description

This struct holds the functions a **driver** (p. 19) must implement, i.e. open, close, read, write and ioctl.

Definition at line 80 of file driver.h.

A.2.2. Field Documentation

A.2.2.1. drv_open_fn driver_interface::open

Definition at line 81 of file driver.h.

A.2.2.2. drv_close_fn driver_interface::close

Definition at line 82 of file driver.h.

A.2.2.3. drv_read_fn driver_interface::read

Definition at line 83 of file driver.h.

A.2.2.4. drv_write_fn driver_interface::write

Definition at line 84 of file driver.h.

A.2.2.5. drv_ioctl_fn driver_interface::ioctl

Definition at line 85 of file driver.h.

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

- **driver.h**

A.3. mem_cfg_data Struct Reference

`#include <types.h>`

Data Fields

- char * **begin**
- char * **end**
- size_t **length**
- **uint32_t access_cycles**
- **int32_t cost**
- **uint32_t flags**
- char * **brk**
- void * **binlist**
- **pthread_mutex_t mutex**

A.3.1. Detailed Description

This struct is for the definition of a memory area.

Definition at line 160 of file types.h.

A.3.2. Field Documentation

A.3.2.1. char* mem_cfg_data::begin

first byte of memory area

Definition at line 161 of file types.h.

A.3.2.2. char* mem_cfg_data::end

first byte after the memory area

Definition at line 162 of file types.h.

A.3.2.3. size_t mem_cfg_data::length

length of memory area

Definition at line 163 of file types.h.

A.3.2.4. uint32_t mem_cfg_data::access_cycles

Average number of cycles to accesss one word of this memory

Definition at line 164 of file types.h.

A.3.2.5. int32_t mem_cfg_data::cost

cost for use of this memory

Definition at line 165 of file types.h.

A.3.2.6. uint32_t mem_cfg_data::flags

flags for further use

Definition at line 166 of file types.h.

A.3.2.7. char* mem_cfg_data::brk

current break (allocated up to this address)

Definition at line 167 of file types.h.

A.3.2.8. void* mem_cfg_data::binlist

the gmalloc bins for this memory; usually located at the start of the memory

Definition at line 168 of file types.h.

A.3.2.9. pthread_mutex_t mem_cfg_data::mutex

mutex for gmalloc

Definition at line 169 of file types.h.

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

- **types.h**

A.4. memorystatistics Struct Reference

`#include <sysmonitor.h>`

Data Fields

- size_t **used**
- **uint32_t freepages_count**
- size_t **freepages_size**
- size_t **min_freepagesize**
- size_t **max_freepagesize**
- size_t **max_ext**

A.4.1. Detailed Description

describes the state of global memory

Definition at line 58 of file sysmonitor.h.

A.4.2. Field Documentation

A.4.2.1. size_t memorystatistics::used

total inuse

Definition at line 59 of file sysmonitor.h.

A.4.2.2. uint32_t memorystatistics::freepages_count

the count of free pages

Definition at line 60 of file sysmonitor.h.

A.4.2.3. size_t memorystatistics::freepages_size

size of all free pages

Definition at line 61 of file sysmonitor.h.

A.4.2.4. size_t memorystatistics::min_freepagesize

size of greatest free page

Definition at line 62 of file sysmonitor.h.

A.4.2.5. size_t memorystatistics::max_freepagesize

size of smallest free page

Definition at line 63 of file sysmonitor.h.

A.4.2.6. size_t memorystatistics::max_ext

maximum amount of memory that can be allocated using sbrk()

Definition at line 64 of file sysmonitor.h.

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

- **sysmonitor.h**

A.5. pthread_attr_t Struct Reference

`#include <types.h>`

Data Fields

- **sched_param param**
- **memory_t * mem**
- **uint32_t flags**
- **uint32_t heapsize**
- enum **sched_policy policy**

A.5.1. Detailed Description

POSIX thread attributes

Definition at line 178 of file types.h.

A.5.2. Field Documentation

A.5.2.1. sched_param pthread_attr_t::param

Definition at line 179 of file types.h.

A.5.2.2. memory_t* pthread_attr_t::mem

Definition at line 180 of file types.h.

A.5.2.3. uint32_t pthread_attr_t::flags

Definition at line 181 of file types.h.

A.5.2.4. uint32_t pthread_attr_t::heapsize

Definition at line 182 of file types.h.

A.5.2.5. enum sched_policy pthread_attr_t::policy

Definition at line 183 of file types.h.

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

- **types.h**

A.6. pthread_barrier_t Struct Reference

```
#include <types.h>
```

Data Fields

- int **needed**
- int **called**
- **pthread_mutex_t mutex**
- **pthread_cond_t cond**

A.6.1. Detailed Description

Barrier - *not fully tested yet*

Definition at line 145 of file types.h.

A.6.2. Field Documentation

A.6.2.1. int pthread_barrier_t::needed

Definition at line 146 of file types.h.

A.6.2.2. int pthread_barrier_t::called

Definition at line 147 of file types.h.

A.6.2.3. pthread_mutex_t pthread_barrier_t::mutex

Definition at line 148 of file types.h.

A.6.2.4. pthread_cond_t pthread_barrier_t::cond

Definition at line 149 of file types.h.

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

- **types.h**

A.7. pthread_barrierattr_t Struct Reference

`#include <types.h>`

Data Fields

- **uint32_t test**

A.7.1. Detailed Description

Barrier attributes

Definition at line 152 of file types.h.

A.7.2. Field Documentation

A.7.2.1. uint32_t pthread_barrierattr_t::test

Definition at line 153 of file types.h.

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

- **types.h**

A.8. pthread_cond Struct Reference

`#include <types.h>`

Data Fields

- **pthread_mutex_t * mutex**
- **tcb_t * waitlist_first_out**
- **tcb_t * waitlist_last_out_hrt**
- **tcb_t * waitlist_last_out**

A.8.1. Detailed Description

Conditional variable

Definition at line 134 of file types.h.

A.8.2. Field Documentation

A.8.2.1. pthread_mutex_t* pthread_cond::mutex

Definition at line 135 of file types.h.

A.8.2.2. tcb_t* pthread_cond::waitlist_first_out

Definition at line 136 of file types.h.

A.8.2.3. tcb_t* pthread_cond::waitlist_last_out_hrt

Definition at line 137 of file types.h.

A.8.2.4. tcb_t* pthread_cond::waitlist_last_out

Definition at line 138 of file types.h.

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

- **types.h**

A.9. pthread_condattr_t Struct Reference

`#include <types.h>`

Data Fields

- **uint32_t test**

A.9.1. Detailed Description

Conditional attributes

Definition at line 141 of file types.h.

A.9.2. Field Documentation

A.9.2.1. uint32_t pthread_condattr_t::test

Definition at line 142 of file types.h.

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

- **types.h**

A.10. pthread_mutex Struct Reference

`#include <types.h>`

Data Fields

- **cc_spinlock_t the_lock**
- **cc_spinlock_t guard**
- **thread_handler owner**
- **sched_t prev_sched**
- **tcb_t ∗ waitlist_first_out**
- **tcb_t ∗ waitlist_last_out_hrt**
- **tcb_t ∗ waitlist_last_out**

A.10.1. Detailed Description

Mutex variable

Definition at line 120 of file types.h.

A.10.2. Field Documentation

A.10.2.1. cc_spinlock_t pthread_mutex::the_lock

Definition at line 121 of file types.h.

A.10.2.2. cc_spinlock_t pthread_mutex::guard

Definition at line 122 of file types.h.

A.10.2.3. thread_handler pthread_mutex::owner

Definition at line 123 of file types.h.

A.10.2.4. sched_t pthread_mutex::prev_sched

Scheduling parameters before the guarded critical block

Definition at line 124 of file types.h.

A.10.2.5. tcb_t∗ pthread_mutex::waitlist_first_out

Definition at line 125 of file types.h.

A.10.2.6. tcb_t* pthread_mutex::waitlist_last_out_hrt

Definition at line 126 of file types.h.

A.10.2.7. tcb_t* pthread_mutex::waitlist_last_out

Definition at line 127 of file types.h.

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

- **types.h**

A.11. pthread_mutexattr_t Struct Reference

`#include <types.h>`

Data Fields

- **uint32_t test**

A.11.1. Detailed Description

Mutex attributes

Definition at line 130 of file types.h.

A.11.2. Field Documentation

A.11.2.1. uint32_t pthread_mutexattr_t::test

Definition at line 131 of file types.h.

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

- **types.h**

A.12. sched_param Struct Reference

`#include <types.h>`

Data Fields

- **sched_t sched_priority**

A.12.1. Detailed Description

Scheduling parameter

Definition at line 110 of file types.h.

A.12.2. Field Documentation

A.12.2.1. sched_t sched_param::sched_priority

Definition at line 111 of file types.h.

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

- **types.h**

A.13. thread_memorystatistics Struct Reference

`#include <sysmonitor.h>`

Data Fields

- size_t **reserved**
- size_t **used**
- **uint32_t freechunks_count**
- size_t **min_freechunksize**
- size_t **max_freechunksize**

A.13.1. Detailed Description

describes the state of a thread's memory

Definition at line 75 of file sysmonitor.h.

A.13.2. Field Documentation

A.13.2.1. size_t thread_memorystatistics::reserved

Definition at line 76 of file sysmonitor.h.

A.13.2.2. size_t thread_memorystatistics::used

Definition at line 77 of file sysmonitor.h.

A.13.2.3. uint32_t thread_memorystatistics::freechunks_count

Definition at line 78 of file sysmonitor.h.

A.13.2.4. size_t thread_memorystatistics::min_freechunksize

Definition at line 79 of file sysmonitor.h.

A.13.2.5. size_t thread_memorystatistics::max_freechunksize

Definition at line 80 of file sysmonitor.h.

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

- **sysmonitor.h**

B. File Documentation

B.1. config.h File Reference

Global definitions for general configuration.

`#include <sys/types.h>`

Defines

- #define **NO_CORES** 4
- #define **NRT_SLOTS_PER_CORE** 3
- #define **HRT_THREADS** NO_CORES
- #define **NRT_THREADS** (NO_CORES ∗ NRT_SLOTS_PER_-CORE)
- #define **__USTACK_SIZE** 20480
- #define **HRT_STACK_BASE** ((**address**) 0xb0000000)
- #define **__CSA_SIZE** 8192
- #define **CSA_BASE** ((**address**) 0xd0000000)
- #define **HRT_CSA_CNT** 128
- #define **NRT_CSA_CNT** 128
- #define **LOGBUFFLEN** 512
- #define **TCB_COUNT** 128
- #define **TCB_BEGIN** 0x90000100
- #define **TCB_END** 0x90001000
- #define **MSS_HRT** (∗((**uint32_t** volatile ∗) 0x90000014))
- #define **MSS_NRT** (∗((**uint32_t** volatile ∗) 0x90000018))
- #define **MSS_MY_CORE** (∗((**uint32_t** volatile ∗) 0x90000000))
- #define **MSS_MY_SLOT** (∗((**uint32_t** volatile ∗) 0x90000004))
- #define **MSS_MY_TCB** (∗((**uint32_t** volatile ∗) 0x90000008))
- #define **STM_TIM0** (∗((**uint32_t** volatile ∗) 0xF0000210))

B.1.1. Detailed Description

This header file includes the general configuration of the MERASA System-Level Software. In many global definitions you can set the details, e.g. the number of used cores, the basic configuration of scheduling, the adresses of used thread memory etc.

Definition in file **config.h**.

B.1.2. Define Documentation

B.1.2.1. #define NO_CORES 4

The number of processor cores used in the System Software. Of course, the distribution of threads over the cores is always done by the scheduler, but this definition makes sure from software side, that there will not be more threads created than the processor can handle.

Definition at line 54 of file config.h.

B.1.2.2. #define NRT_SLOTS_PER_CORE 3

The number of non real-time slots per core. Usually, one MERASA core provides four slots - the first is for hard real-time and the three others for non real-time threads.

Definition at line 57 of file config.h.

B.1.2.3. #define HRT_THREADS NO_CORES

The number of hard real-time threads. Usually, there is one hard real-time thread per core.

Definition at line 63 of file config.h.

B.1.2.4. #define NRT_THREADS (NO_CORES * NRT_SLOTS_-PER_CORE)

The number of non real-time threads. Usually, there is one hard-real time thread per core, all others are non real-time threads.

Definition at line 66 of file config.h.

B.1.2.5. #define __USTACK_SIZE 20480

The size of the reserved user stack.

Definition at line 69 of file config.h.

B.1.2.6. #define HRT_STACK_BASE ((address) 0xb0000000)

The base address of the common stack for HRT threads (optimised for the use of a scratchpad).

Definition at line 72 of file config.h.

B.1.2.7. #define __CSA_SIZE 8192

The size of one reserved context save area (CSA). This is necessary to save the context (i.e. register values ...) of a function before performing another call.

Definition at line 75 of file config.h.

B.1.2.8. #define CSA_BASE ((address) 0xd0000000)

The base address of the reserved context save areas (CSA).

Definition at line 78 of file config.h.

B.1.2.9. #define HRT_CSA_CNT 128

The number of context save areas reserved for hard real-time threads.

Definition at line 81 of file config.h.

B.1.2.10. #define NRT_CSA_CNT 128

The number of context save areas reserved for non real-time threads.

Definition at line 84 of file config.h.

B.1.2.11. #define LOGBUFFLEN 512

The buffer length for log information in the virtual output.

Definition at line 90 of file config.h.

B.1.2.12. #define TCB_COUNT 128

The maximum number of threads. This is only a theoretical value, because the real number of threads is determined by the core and the available slots per core.

Definition at line 99 of file config.h.

B.1.2.13. #define TCB_BEGIN 0x90000100

The base address of Thread Control Blocks (TCBs). The range before TCB_-BEGIN is reserved for some specific scheduling information (see the MSS_* definitions for details). The begin address for the first Thread Control Block (TCB). This adress is usually used for the boot thread.

Definition at line 105 of file config.h.

B.1.2.14. #define TCB_END 0x90001000

The end address of the Thread Control Blocks (first byte after).

Definition at line 108 of file config.h.

B.1.2.15. #define MSS_HRT (∗((uint32_t volatile ∗) 0x90000014))

A pointer to the address of the first TCB in the list of hard real-time threads.

Definition at line 111 of file config.h.

B.1.2.16. #define MSS_NRT (∗((uint32_t volatile ∗) 0x90000018))

A pointer to the address of the first TCB in the list of non real-time threads.

Definition at line 114 of file config.h.

B.1.2.17. #define MSS_MY_CORE (∗((uint32_t volatile ∗) 0x90000000))

Contains the core number of the actual thread.

Definition at line 117 of file config.h.

Referenced by get_current_core().

B.1.2.18. #define MSS_MY_SLOT (∗((uint32_t volatile ∗) 0x90000004))

Contains the slot number of the actual thread

Definition at line 120 of file config.h.

Referenced by get_current_slot().

B.1.2.19. #define MSS_MY_TCB (∗((uint32_t volatile ∗) 0x90000008))

Contains the TCB address of the actual thread

Definition at line 123 of file config.h.

B.1.2.20. #define STM_TIM0 (∗((uint32_t volatile ∗) 0xF0000210))

Contains the actual system time.

Definition at line 126 of file config.h.

B.2. driver.h File Reference

Macros for writing MERASA device drivers.

`#include <pthread.h>`

`#include <stdarg.h>`

`#include <sys/types.h>`

Data Structures

- struct **driver_interface**
- struct **driver**

Defines

- #define **SDRIVER**(n, v, i, c, o)

Typedefs

- typedef **int32_t**(∗ **drv_init_fn_t**)(const void ∗)
- typedef **int32_t**(∗ **drv_cleanup_fn_t**)(void)
- typedef **int32_t**(∗ **drv_open_fn**)(void)
- typedef **int32_t**(∗ **drv_close_fn**)(void)
- typedef size_t(∗ **drv_read_fn**)(void ∗, size_t)
- typedef size_t(∗ **drv_write_fn**)(const void ∗, size_t)
- typedef size_t(∗ **drv_ioctl_fn**)(**uint32_t**, va_list)
- typedef struct **driver_interface drvif_t**
- typedef struct **driver driver_t**

B.2.1. Detailed Description

This file provides macros and data types that are necessary to write a hardware device **driver** (p. 19) for the MERASA operating system. For examples how to use them, see existing **driver** (p. 19) files. Please make sure to include all necessary functions within the **driver** (p. 19) program. The DriverManager will only resolve dependencies to OS API calls provided in the include directory!

Definition in file **driver.h**.

B.2.2. Define Documentation

B.2.2.1. #define SDRIVER(n, v, i, c, o)

Value:

```
struct driver sdriver_ ##n = { \
    .name = #n, \
    .version = v, \
    .init = i, \
    .cleanup = c, \
    .ops = o, \
    .pi = NULL, \
    .owner = NO_THREAD, \
    .sp_next = NULL }
```

Macro for a static **driver** (p. 19).

Use this macro to publish your own drivers. For examples how to use them, see existing **driver** (p. 19) files.

Parameters:

- ***n*** Name of the **driver** (p. 19).
- ***v*** Version of the **driver** (p. 19) (shoud be set to 0x10).
- ***i*** Initialisation function.
- ***c*** Cleanup function.
- ***o*** Operations struct (**drvif_t** (p. 42)).

Definition at line 115 of file driver.h.

B.2.3. Typedef Documentation

B.2.3.1. typedef int32_t(∗ drv_init_fn_t)(const void ∗)

The initialisation function of a **driver** (p. 19). The passed pointer points to the device's start address in memory. Currently, this function must not fail!

Definition at line 58 of file driver.h.

B.2.3.2. typedef int32_t(∗ drv_cleanup_fn_t)(void)

The cleanup function of a **driver** (p. 19).

Definition at line 61 of file driver.h.

B.2.3.3. typedef int32_t(∗ drv_open_fn)(void)

The open function of a **driver** (p. 19).

Definition at line 64 of file driver.h.

B.2.3.4. typedef int32_t(∗ drv_close_fn)(void)

The close function of a **driver** (p. 19).

Definition at line 67 of file driver.h.

B.2.3.5. typedef size_t(∗ drv_read_fn)(void ∗, size_t)

The read function of a **driver** (p. 19).

Definition at line 70 of file driver.h.

B.2.3.6. typedef size_t(∗ drv_write_fn)(const void ∗, size_t)

The write function of a **driver** (p. 19).

Definition at line 73 of file driver.h.

B.2.3.7. typedef size_t(∗ drv_ioctl_fn)(uint32_t, va_list)

The ioctl function of a **driver** (p. 19).

Definition at line 76 of file driver.h.

B.2.3.8. typedef struct driver_interface drvif_t

B.2.3.9. typedef struct driver driver_t

B.3. drivermanager.h File Reference

Macros to declare and register a **driver** (p. 19).

Defines

- #define **EXTERN_DRIVER**(name) extern **driver_t** sdriver_-##name;
- #define **REGISTER_DRIVER**(name) &sdriver_##name

B.3.1. Detailed Description

This file provides macros necessary to declare and register a **driver** (p. 19) in the user application.

Definition in file **drivermanager.h**.

B.3.2. Define Documentation

B.3.2.1. #define EXTERN_DRIVER(name) extern driver_t sdriver_-##name;

Simple macro for the declaration of a **driver** (p. 19).

Definition at line 59 of file drivermanager.h.

B.3.2.2. #define REGISTER_DRIVER(name) &sdriver_##name

Simple macro to register a **driver** (p. 19) in the user application.

Definition at line 65 of file drivermanager.h.

B.4. error.h File Reference

Definitions of error numbers.

`#include <sys/types.h>`

Defines

- #define **E_OK** 0
- #define **EPERM** 1
- #define **ENOENT** 2
- #define **ESRCH** 3
- #define **EINTR** 4
- #define **EIO** 5
- #define **ENXIO** 6
- #define **E2BIG** 7
- #define **ENOEXEC** 8
- #define **EBADF** 9
- #define **ECHILD** 10
- #define **EAGAIN** 11
- #define **ENOMEM** 12
- #define **EACCES** 13
- #define **EFAULT** 14
- #define **ENOTBLK** 15
- #define **EBUSY** 16
- #define **EEXIST** 17
- #define **EXDEV** 18
- #define **ENODEV** 19
- #define **ENOTDIR** 20
- #define **EISDIR** 21
- #define **EINVAL** 22
- #define **ENFILE** 23
- #define **EMFILE** 24
- #define **ENOTTY** 25
- #define **ETXTBSY** 26
- #define **EFBIG** 27
- #define **ENOSPC** 28
- #define **ESPIPE** 29
- #define **EROFS** 30
- #define **EMLINK** 31

- #define **EPIPE** 32
- #define **EDOM** 33
- #define **ERANGE** 34

Functions

- **error_t get_errno** (void)
- void **set_errno** (**error_t** errno)

B.4.1. Detailed Description

This file contains definitions of error numbers, similar to POSIX.

Definition in file **error.h**.

B.4.2. Define Documentation

B.4.2.1. #define E_OK 0

Definition at line 53 of file error.h.

B.4.2.2. #define EPERM 1

Operation not permitted

Definition at line 57 of file error.h.

B.4.2.3. #define ENOENT 2

No such file or directory

Definition at line 58 of file error.h.

B.4.2.4. #define ESRCH 3

No such process

Definition at line 59 of file error.h.

B.4.2.5. #define EINTR 4

Interrupted system call

Definition at line 60 of file error.h.

B.4.2.6. #define EIO 5

I/O error

Definition at line 61 of file error.h.

B.4.2.7. #define ENXIO 6

No such device or address

Definition at line 62 of file error.h.

B.4.2.8. #define E2BIG 7

Argument list too long

Definition at line 63 of file error.h.

B.4.2.9. #define ENOEXEC 8

Exec format error

Definition at line 64 of file error.h.

B.4.2.10. #define EBADF 9

Bad file number

Definition at line 65 of file error.h.

B.4.2.11. #define ECHILD 10

No child processes

Definition at line 66 of file error.h.

B.4.2.12. #define EAGAIN 11

Try again

Definition at line 67 of file error.h.

B.4.2.13. #define ENOMEM 12

Out of memory

Definition at line 68 of file error.h.

B.4.2.14. #define EACCES 13

Permission denied

Definition at line 69 of file error.h.

B.4.2.15. #define EFAULT 14

Bad address

Definition at line 70 of file error.h.

B.4.2.16. #define ENOTBLK 15

Block device required

Definition at line 71 of file error.h.

B.4.2.17. #define EBUSY 16

Device or resource busy

Definition at line 72 of file error.h.

B.4.2.18. #define EEXIST 17

File exists

Definition at line 73 of file error.h.

B.4.2.19. #define EXDEV 18

Cross-device link

Definition at line 74 of file error.h.

B.4.2.20. #define ENODEV 19

No such device

Definition at line 75 of file error.h.

B.4.2.21. #define ENOTDIR 20

Not a directory

Definition at line 76 of file error.h.

B.4.2.22. #define EISDIR 21

Is a directory

Definition at line 77 of file error.h.

B.4.2.23. #define EINVAL 22

Invalid argument

Definition at line 78 of file error.h.

B.4.2.24. #define ENFILE 23

File table overflow

Definition at line 79 of file error.h.

B.4.2.25. #define EMFILE 24

Too many open files

Definition at line 80 of file error.h.

B.4.2.26. #define ENOTTY 25

Not a typewriter

Definition at line 81 of file error.h.

B.4.2.27. #define ETXTBSY 26

Text file busy

Definition at line 82 of file error.h.

B.4.2.28. #define EFBIG 27

File too large

Definition at line 83 of file error.h.

B.4.2.29. #define ENOSPC 28

No space left on device

Definition at line 84 of file error.h.

B.4.2.30. #define ESPIPE 29

Illegal seek

Definition at line 85 of file error.h.

B.4.2.31. #define EROFS 30

Read-only file system

Definition at line 86 of file error.h.

B.4.2.32. #define EMLINK 31

Too many links

Definition at line 87 of file error.h.

B.4.2.33. #define EPIPE 32

Broken pipe

Definition at line 88 of file error.h.

B.4.2.34. #define EDOM 33

Math argument out of domain of func

Definition at line 89 of file error.h.

B.4.2.35. #define ERANGE 34

Math result not representable

Definition at line 90 of file error.h.

B.4.3. Function Documentation

B.4.3.1. error_t get_errno (void)

Get the error number of the actual thread.

Returns:

E_OK if there is no error, otherwise the specified error number.

B.4.3.2. void set_errno (error_t *errno*)

Set the error number of the actual thread.

B.5. fcntl.h File Reference

Open or create a device for reading or writing.

Functions

- int **open** (const char ∗path, int flags,...)

B.5.1. Detailed Description

Definition in file **fcntl.h**.

B.5.2. Function Documentation

B.5.2.1. int open (const char ∗ *path*, int *flags*, ...)

Open or create a device for reading or writing.

The device name specified by path is opened for reading and/or writing as specified by the argument flags and the descriptor returned to the calling process.

Returns:

If successful, **open()** (p. 50) returns a non-negative integer, termed a device descriptor. It returns -1 on failure, and sets errno to indicate the error.

B.6. log.h File Reference

Logging macros for the virtual output of the MERASA simulator.

`#include <config.h>`

`#include <vo.h>`

`#include <pthread.h>`

Defines

- #define **LOGLEVEL_DEBUG** 5
- #define **LOGLEVEL_INFO** 4
- #define **LOGLEVEL_WARN** 3
- #define **LOGLEVEL_ERR** 2
- #define **LOGLEVEL_FATAL** 1
- #define **LOGLEVEL_NONE** 0
- #define **LOGDBL_D4C**(w, c1, c2, c3, c4)
- #define **LOGDBL_X4C**(w, c1, c2, c3, c4)
- #define **log_debug_s**(msg, args...)
- #define **log_debug**(msg)
- #define **log_info_s**(msg, args...)
- #define **log_info**(msg)
- #define **log_warn_s**(msg, args...)
- #define **log_warn**(msg)
- #define **log_err_s**(msg, args...)
- #define **log_err**(msg)
- #define **log_fatal_s**(msg, args...)
- #define **log_fatal**(msg)

Functions

- int **sprintf** (char *str, const char *format,...)

Variables

- **pthread_mutex_t log_mutex**

B.6.1. Detailed Description

This file provides logging facilities by means of different loglevels (DEBUG, INFO, WARN, ERR, FATAL, NONE); the global loglevel is usually defined in the Makefile. Attention: As the log output of a thread is locked by one common mutex, in some cases there may be problems like priority inversion. For this reason, we suggest to use the fast one-cycle logging methods LOGDBL_D4C, LOGDBL_DS, LOGDBL_X4C and LOGDBL_XS (for details of usage see below).

Definition in file **log.h**.

B.6.2. Define Documentation

B.6.2.1. #define LOGLEVEL_DEBUG 5

Definition at line 57 of file log.h.

B.6.2.2. #define LOGLEVEL_INFO 4

Definition at line 58 of file log.h.

B.6.2.3. #define LOGLEVEL_WARN 3

Definition at line 59 of file log.h.

B.6.2.4. #define LOGLEVEL_ERR 2

Definition at line 60 of file log.h.

B.6.2.5. #define LOGLEVEL_FATAL 1

Definition at line 61 of file log.h.

B.6.2.6. #define LOGLEVEL_NONE 0

Definition at line 62 of file log.h.

B.6.2.7. #define LOGDBL_D4C(w, c1, c2, c3, c4)

Value:

```
{ \
        uint64_t v = (w) | \
        ((uint64_t)(c1)<<32) | \
        ((uint64_t)(c2)<<40) | \
        ((uint64_t)(c3)<<48) | \
        ((uint64_t)(c4)<<56); \
        P64(VO_DBL_D4C) = v; \
}
```

Fast log of one decimal and four chars.

This macro enables a very fast logging output of one decimal and four chars (usually describing the decimal). As the output is finished within one cycle, you don't need the logging mutex and no priority inversion can occur.

Definition at line 91 of file log.h.

B.6.2.8. #define LOGDBL_X4C(w, c1, c2, c3, c4)

Value:

```
{ \
        uint64_t v = (w) | \
        ((uint64_t)(c1)<<32) | \
        ((uint64_t)(c2)<<40) | \
        ((uint64_t)(c3)<<48) | \
        ((uint64_t)(c4)<<56); \
        P64(VO_DBL_X4C) = v; \
}
```

Fast log of one decimal and a string.

This macro enables a very fast logging output of one decimal and a string. This string should contain 4 characters - if it's longer, it will be cut. As the output is finished within one cycle, you don't need the logging mutex and no priority inversion can occur.

Fast log of one hexadecimal and four chars This macro enables a very fast logging output of one hexadecimal and four chars (usually describing the hexadecimal). As the output is finished within one cycle, you don't need the logging mutex and no priority inversion can occur.

Definition at line 113 of file log.h.

B.6.2.9. #define log_debug_s(msg, args...)

Value:

```
{ \
        LOCK_LOG(); \
        char buffer[LOGBUFFLEN]; \
        sprintf(buffer, msg, args); \
        printf("[5,%u] %s:%d %s\n", get_thread(),__FILE__,  __LINE__, buffer); \
        UNLOCK_LOG(); \
}
```

Fast log of one hexadecimal and a string.

This macro enables a very fast logging output of one hexadecimal and a string. This string should contain 4 characters - if it's longer, it will be cut. As the output is finished within one cycle, you don't need the logging mutex and no priority inversion can occur.

Logging function for debug output with several arguments This macro enables a logging output for debugging. It is used similar to the common printf() with an undefined number of arguments.

Definition at line 141 of file log.h.

B.6.2.10. #define log_debug(msg)

Value:

```
{ \
        LOCK_LOG(); \
        printf("[5,%u] %s:%d %s\n", get_thread(),__FILE__,  __LINE__, msg); \
        UNLOCK_LOG(); \
}
```

Logging function for a string debug output.

This macro enables a logging output for debugging. It is used similar to the common printf(), but only for one argument.

Definition at line 158 of file log.h.

B.6.2.11. #define log_info_s(msg, args...)

Value:

```
{ \
        LOCK_LOG(); \
        char buffer[LOGBUFFLEN]; \
        sprintf(buffer, msg, args); \
        printf("[4,%u] %s:%d %s\n", get_thread(),__FILE__,  __LINE__, buffer); \
        UNLOCK_LOG(); \
}
```

Logging function for info output with several arguments.

This macro enables a logging output for information. It is used similar to the common printf() with an undefined number of arguments.

Definition at line 174 of file log.h.

B.6.2.12. #define log_info(msg)

Value:

```
{ \
        LOCK_LOG(); \
        printf("[4,%u] %s:%d %s\n", get_thread(),__FILE__,  __LINE__, msg); \
        UNLOCK_LOG(); \
}
```

Logging function for a string info output.

This macro enables a logging output for information. It is used similar to the common printf(), but only for one argument.

Definition at line 190 of file log.h.

B.6.2.13. #define log_warn_s(msg, args...)

Value:

```
{ \
        LOCK_LOG(); \
        char buffer[LOGBUFFLEN]; \
        sprintf(buffer, msg, args); \
        printf("[3,%u] %s:%d %s\n", get_thread(),__FILE__,  __LINE__, buffer); \
        UNLOCK_LOG(); \
}
```

Logging function for warning output with several arguments.

This macro enables a logging output for warning. It is used similar to the common printf() with an undefined number of arguments.

Definition at line 206 of file log.h.

B.6.2.14. #define log_warn(msg)

Value:

```
{ \
        LOCK_LOG(); \
        printf("[3,%u] %s:%d %s\n", get_thread(),__FILE__,  __LINE__, msg); \
        UNLOCK_LOG(); \
}
```

Logging function for a string warning output.

This macro enables a logging output for warning. It is used similar to the common printf(), but only for one argument.

Definition at line 222 of file log.h.

B.6.2.15. #define log_err_s(msg, args...)

Value:

```
{ \
        LOCK_LOG(); \
        char buffer[LOGBUFFLEN]; \
        sprintf(buffer, msg, args); \
        printf("[2,%u] %s:%d %s\n", get_thread(),__FILE__,  __LINE__, buffer); \
        UNLOCK_LOG(); \
}
```

Logging function for error output with several arguments.

This macro enables a logging output for error. It is used similar to the common printf() with an undefined number of arguments.

Definition at line 238 of file log.h.

B.6.2.16. #define log_err(msg)

Value:

```
{ \
        LOCK_LOG(); \
        printf("[2,%u] %s:%d %s\n", get_thread(),__FILE__,  __LINE__, msg); \
        UNLOCK_LOG(); \
}
```

Logging function for a string error output.

This macro enables a logging output for error. It is used similar to the common printf(), but only for one argument.

Definition at line 254 of file log.h.

B.6.2.17. #define log_fatal_s(msg, args...)

Value:

```
{ \
        LOCK_LOG(); \
        char buffer[LOGBUFFLEN]; \
        sprintf(buffer, msg, args); \
        printf("[1,%u] %s:%d %s\n", get_thread(),__FILE__,  __LINE__, buffer); \
        UNLOCK_LOG(); \
}
```

Logging function for fatal error output with several arguments.

This macro enables a logging output for fatal error. It is used similar to the common printf() with an undefined number of arguments.

Definition at line 270 of file log.h.

B.6.2.18. #define log_fatal(msg)

Value:

```
{ \
        LOCK_LOG(); \
        printf("[1,%u] %s:%d %s\n", get_thread(),__FILE__,  __LINE__, msg); \
        UNLOCK_LOG(); \
}
```

Logging function for a string fatal error output.

This macro enables a logging output for fatal error. It is used similar to the common printf(), but only for one argument.

Definition at line 286 of file log.h.

B.6.3. Function Documentation

B.6.3.1. int sprintf (char ∗ *str*, const char ∗ *format*, ...)

Better use the logging macros.

B.6.4. Variable Documentation

B.6.4.1. pthread_mutex_t log_mutex

The mutex for the virtual output of the MERASA simulator. Necessary, if you don't want a mixture of chars from different threads.

B.7. memory-desc.h File Reference

Description of memory.

`#include <sys/types.h>`

`#include <pthread.h>`

Defines

- #define **MEMORY_CFG**(b, l, ac, c, f)

Variables

- **memory_t node_mem_config** [ ]
- const size_t **mem_config_len**

B.7.1. Detailed Description

This file provides functionalities to define and characterize a memory region.

Definition in file **memory-desc.h**.

B.7.2. Define Documentation

B.7.2.1. #define MEMORY_CFG(b, l, ac, c, f)

Value:

```
{ .begin = b, \
  .length = l, \
  .end = b + l, \
  .access_cycles = ac, \
  .cost = c, \
  .flags = f, \
  .brk = 0, \
  .binlist = 0 \
}
```

Use this macro to define a memory region. See an example of usage in the userapp.c.

Parameters:

b begin

l lenght

ac access_cycles

c cost

f flags

Definition at line 83 of file memory-desc.h.

B.7.3. Variable Documentation

B.7.3.1. memory_t node_mem_config[ ]

You need to define this constant in the OS for one specific node configuration. Use the **MEMORY_CFG** (p. 58) macro below for filling this array!

B.7.3.2. const size_t mem_config_len

For correct access to **node_mem_config** (p. 59), you have to set this constant as sizeof(node_mem_config)/sizeof(memory_t).

B.8. memory.h File Reference

Basic functionalities of memory management.

`#include <pthread.h>`

`#include <sys/types.h>`

`#include <memory-desc.h>`

Functions

- void * **malloc** (size_t size)
- void * **calloc** (size_t number, size_t size)
- void * **realloc** (void *ptr, size_t size)
- void **free** (void *ptr)
- void * **tcmemcpy** (void *dest, const void *src, size_t n)
- **bool_t has_write_permission** (void *mem)

B.8.1. Detailed Description

This file provides functionalities to allocate and free memory, but also to copy memory regions or check for write permissions.

Definition in file **memory.h**.

B.8.2. Function Documentation

B.8.2.1. void* malloc (size_t *size*)

The **malloc()** (p. 60) function allocates size bytes of memory and returns a pointer to the allocated memory.

Parameters:

size amount of memory to allocate

B.8.2.2. void* calloc (size_t *number*, size_t *size*)

The **calloc()** (p. 60) function allocates space for number objects, each size bytes in length. The result is identical to calling **malloc()** (p. 60) with an argument of "number * size", with the exception that the allocated memory is explicitly initialized to zero bytes.

B.8.2.3. void* realloc (void * *ptr*, size_t *size*)

The **realloc()** (p. 60) function changes the size of the previously allocated memory referenced by ptr to size bytes. The contents of the memory are unchanged up to the lesser of the new and old sizes. If the new size is larger, the value of the newly allocated portion of the memory is undefined. Upon success, the memory referenced by ptr is freed and a pointer to the newly allocated memory is returned.

B.8.2.4. void free (void * *ptr*)

Free a previously allocated block of the current thread.

Parameters:

ptr the memory block

B.8.2.5. void* tcmemcpy (void * *dest*, const void * *src*, size_t *n*)

Effective copying of (non-overlapping!) memory.

B.8.2.6. bool_t has_write_permission (void * *mem*)

Check if the thread is allowed to write to the specified address.

B.9. merasa-ssw.h File Reference

All MERASA includes.

```
#include <error.h>
#include <driver.h>
#include <drivermanager.h>
#include <fcntl.h>
#include <log.h>
#include <memory-desc.h>
#include <memory.h>
#include <pthread.h>
#include <stropts.h>
#include <sysmonitor.h>
#include <sys/types.h>
#include <unistd.h>
#include <vo.h>
```

B.9.1. Detailed Description

This file contains all headers of the MERASA include directory. This enables an easy inclusion into your user application.

Definition in file **merasa-ssw.h**.

B.10. pthread.h File Reference

POSIX thread functions and more.

`#include <sys/types.h>`

`#include <config.h>`

Defines

- #define **NO_THREAD** ((**int32_t**)-1)
- #define **NO_OWNER** ((**thread_handler**)-1)
- #define **get_thread**() get_current_thread_handler()
- #define **THREAD_PRIV_NRT** 0x00000001
- #define **THREAD_PRIV_HRT** 0x00000004
- #define **THREAD_PRIV_THRMG** 0x00000008
- #define **THREAD_PRIV_DYNMEM** 0x00000010
- #define **THREAD_PRIV_MEXT** 0x00000020
- #define **THREAD_PRIV_TLSF** 0x00000040
- #define **THREAD_PRIV_MODS** 0x00000100
- #define **THREAD_PRIV_GMOD** 0x00000200
- #define **THREAD_PRIV_APPS** 0x00000400
- #define **THREAD_PRIV_DRVS** 0x00000800

Functions

- **threadptr get_thread4handler** (**thread_handler** th)
- **threadptr get_current_thread** (void)
- **thread_handler get_current_thread_handler** (void)
- **bool_t has_privilege** (**uint32_t** priv)
- void **yield** (void)
- static **uint32_t get_current_core** (void)
- static **uint32_t get_current_slot** (void)
- **uint32_t finish_thread_init** (void)
- int **pthread_create** (**pthread_t** *thread, const **pthread_attr_t** *attr, void *(*start_routine)(void *), void *arg)
- int **pthread_join** (**pthread_t** thread, void **value_ptr)
- **pthread_t pthread_self** (void)
- void **pthread_yield** (void)
- int **pthread_attr_getschedpolicy** (const **pthread_attr_t** *attr, int *policy)
- int **pthread_attr_init** (**pthread_attr_t** *attr)

- int **pthread_attr_setschedpolicy** (**pthread_attr_t** ∗attr, int policy)
- int **pthread_attr_getmemory** (**pthread_attr_t** ∗attr, **memory_t** ∗∗mem)
- int **pthread_attr_setmemory** (**pthread_attr_t** ∗attr, **memory_t** ∗mem)
- int **pthread_attr_getbasicheapsize** (**pthread_attr_t** ∗attr, int ∗heapsize)
- int **pthread_attr_setbasicheapsize** (**pthread_attr_t** ∗attr, int heapsize)
- int **pthread_attr_getflags** (**pthread_attr_t** ∗attr, int ∗flags)
- int **pthread_attr_setflags** (**pthread_attr_t** ∗attr, int flags)
- int **pthread_attr_getiq** (**pthread_attr_t** ∗attr, int ∗iq)
- int **pthread_attr_setiq** (**pthread_attr_t** ∗attr, int iq)
- int **pthread_mutex_destroy** (**pthread_mutex_t** ∗mutex)
- int **pthread_mutex_init** (**pthread_mutex_t** ∗mutex, const **pthread_mutexattr_t** ∗attr)
- int **pthread_mutex_lock** (**pthread_mutex_t** ∗mutex)
- int **pthread_mutex_trylock** (**pthread_mutex_t** ∗mutex)
- int **pthread_mutex_unlock** (**pthread_mutex_t** ∗mutex)
- int **pthread_cond_broadcast** (**pthread_cond_t** ∗cond)
- int **pthread_cond_destroy** (**pthread_cond_t** ∗cond)
- int **pthread_cond_init** (**pthread_cond_t** ∗cond, const **pthread_-condattr_t** ∗attr)
- int **pthread_cond_signal** (**pthread_cond_t** ∗cond)
- int **pthread_cond_wait** (**pthread_cond_t** ∗, **pthread_mutex_t** ∗mutex)
- int **pthread_barrier_destroy** (**pthread_barrier_t** ∗barrier)
- int **pthread_barrier_init** (**pthread_barrier_t** ∗barrier, const **pthread_barrierattr_t** ∗attr, unsigned count)
- int **pthread_barrier_wait** (**pthread_barrier_t** ∗barrier)

B.10.1. Detailed Description

This file contains not only a subset of POSIX thread functions, but also some other definitions and tool functions for thread management.

Definition in file **pthread.h**.

B.10.2. Define Documentation

B.10.2.1. #define NO_THREAD ((int32_t)-1)

Return value if no thread is selected.

Definition at line 51 of file pthread.h.

B.10.2.2. #define NO_OWNER ((thread_handler)-1)

Return value if no owner is selected.

Definition at line 53 of file pthread.h.

B.10.2.3. #define get_thread() get_current_thread_handler()

Returns the actual thread handler.

Definition at line 55 of file pthread.h.

B.10.2.4. #define THREAD_PRIV_NRT 0x00000001

Non real-time scheduling.

Definition at line 59 of file pthread.h.

B.10.2.5. #define THREAD_PRIV_HRT 0x00000004

Hard real-time scheduling. The hard realtime thread should use TLSF (real-time capable memory management).

Definition at line 61 of file pthread.h.

B.10.2.6. #define THREAD_PRIV_THRMG 0x00000008

Thread has access to thread management (usually only boot thread).

Definition at line 63 of file pthread.h.

B.10.2.7. #define THREAD_PRIV_DYNMEM 0x00000010

Thread uses dynamic memory management.

Definition at line 67 of file pthread.h.

B.10.2.8. #define THREAD_PRIV_MEXT 0x00000020

Thread may extend its memory (i.e. the local malloc may do so) (needs DYN-MEM!).

Definition at line 69 of file pthread.h.

B.10.2.9. #define THREAD_PRIV_TLSF 0x00000040

Thread uses TLSF for DSA, or if not set, uses DLAlloc (needs DYNMEN!). When using TLSF, online-memory extension is not allowed (the MEXT flag is ignored). So make sure to reserve enough memory at thread creation!

Definition at line 71 of file pthread.h.

B.10.2.10. #define THREAD_PRIV_MODS 0x00000100

Thread is allowed to load program modules (needs DYNMEM and MEXT too!). Use with care! The local namespace will influence the memory consumption of the thread.

Definition at line 75 of file pthread.h.

B.10.2.11. #define THREAD_PRIV_GMOD 0x00000200

Thread may load modules into global namespace.

Definition at line 77 of file pthread.h.

B.10.2.12. #define THREAD_PRIV_APPS 0x00000400

Thread may load applications.

Definition at line 79 of file pthread.h.

B.10.2.13. #define THREAD_PRIV_DRVS 0x00000800

Thread may manage drivers (load/unload).

Definition at line 81 of file pthread.h.

B.10.3. Function Documentation

B.10.3.1. threadptr get_thread4handler (thread_handler *th*)

Returns the thread pointer of the specified thread.

B.10.3.2. threadptr get_current_thread (void)

Returns the thread pointer of the current thread.

B.10.3.3. thread_handler get_current_thread_handler (void)

Returns the thread handler of the current thread.

B.10.3.4. bool_t has_privilege (uint32_t *priv*)

Returns 1, if the current thread has a specified privilege. Otherwise 0.

B.10.3.5. void yield (void)

Allows the scheduler to run another thread instead of the current one.

B.10.3.6. static uint32_t get_current_core (void) `[inline, static]`

Returns the core of the current thread.

Definition at line 97 of file pthread.h.

References MSS_MY_CORE.

B.10.3.7. static uint32_t get_current_slot (void) `[inline, static]`

Returns the slot of the current thread.

Definition at line 101 of file pthread.h.

References MSS_MY_SLOT.

B.10.3.8. uint32_t finish_thread_init (void)

Finishes the initialisation phase.

This function finishes the thread creation phase and sets the lists of hard- and non real-time threads to the scheduler. Attention: The scheduling will not start before calling this function!

B.10.3.9. int pthread_create (pthread_t * *thread*, const pthread_-attr_t * *attr*, void *(*)(void *) *start_routine*, void * *arg*)

Creates a new thread of execution.

The **pthread_create()** (p. 67) function is used to create a new thread, with attributes specified by attr, within a process. If the attributes specified by attr are modified later, the thread's attributes are not affected. Upon successful completion **pthread_create()** (p. 67) will store the ID of the created thread in the location specified by thread. The thread is created executing start_routine with arg as its sole argument.

Returns:

If successful, the **pthread_create()** (p. 67) function will return zero. Otherwise an error number will be returned to indicate the error.

B.10.3.10. int pthread_join (pthread_t *thread*, void ** *value_ptr*)

Causes the calling thread to wait for the termination of the specified thread.

The **pthread_join()** (p. 67) function suspends execution of the calling thread until the target thread terminates unless the target thread has already terminated.

When a **pthread_join()** (p. 67) returns successfully, the target thread has been terminated. The results of multiple simultaneous calls to **pthread_join()** (p. 67) specifying the same tar- get thread are undefined. If the thread calling **pthread_join()** (p. 67) is can- celled, then the target thread is not detached.

Returns:

If successful, the **pthread_join()** (p. 67) function will return zero. Otherwise an error number will be returned to indicate the error.

B.10.3.11. pthread_t pthread_self (void)

Returns the thread ID of the calling thread.

Returns:

The **pthread_self()** (p. 68) function returns the thread ID of the calling thread.

B.10.3.12. void pthread_yield (void)

Allows the scheduler to run another thread instead of the current one.

B.10.3.13. int pthread_attr_getschedpolicy (const pthread_attr_t * *attr*, int * *policy*)

Get the scheduling policy attribute from a thread attributes object.

B.10.3.14. int pthread_attr_init (pthread_attr_t * *attr*)

Initialize a thread attributes object with default values.

Thread attributes are used to specify parameters to **pthread_create()** (p. 67). One attribute object can be used in multiple calls to **pthread_create()** (p. 67), with or without modifications between calls.

The **pthread_attr_init()** (p. 68) function initializes attr with all the default thread attributes.

Returns:

If successful, this function returns 0. Otherwise, an error number is returned to indicate the error.

B.10.3.15. int pthread_attr_setschedpolicy (pthread_attr_t * *attr*, int *policy*)

Set the scheduling policy attribute in a thread attributes object.

This is necessary to select the right thread type. Possible values are

- `SCHED_HRT`: hard real-time thread

- `SCHED_NRT`: non real-time thread

After initialization, the policy is set to `SCHED_NONE` (a thread is not able to run with this value).

Returns:

If successful, this function returns 0. Otherwise, an error number is returned to indicate the error.

B.10.3.16. int pthread_attr_getmemory (pthread_attr_t * *attr*, memory_t ** *mem*)

Get the memory type used for the thread.

Returns:

If successful, this function returns 0. Otherwise, an error number is returned to indicate the error.

B.10.3.17. int pthread_attr_setmemory (pthread_attr_t * *attr*, memory_t * *mem*)

Set the memory to use for the thread.

If set to NULL, system standard memory will be used.

Returns:

If successful, this function returns 0. Otherwise, an error number is returned to indicate the error.

B.10.3.18. int pthread_attr_getbasicheapsize (pthread_attr_t * *attr*, int * *heapsize*)

Get the initial heap size used for the thread.

See **pthread_attr_setbasicheapsize()** (p. 69) for more details.

Returns:

If successful, this function returns 0. Otherwise, an error number is returned to indicate the error.

B.10.3.19. int pthread_attr_setbasicheapsize (pthread_attr_t * *attr*, int *heapsize*)

Set the initial heap size used for the thread.

Set the initial memory for the thread (only sensible, if the thread needs not to extend its memory). Calculation of the parameter thread_mem: For each variable, you need to allocate, add 4 bytes (1 word) management overhead and round each of these values up to a 8-byte-alignment. The minimum amount of memory that can be allocated is 16 bytes (4 word).

sz: amount of memory needed for a variable

=> real_sz = (sz+4 + 7) & ~8

Thus, all real_sz values added up result in the thread_mem parameter.

Returns:

> If successful, this function returns 0. Otherwise, an error number is returned to indicate the error.

B.10.3.20. int pthread_attr_getflags (pthread_attr_t * *attr*, int * *flags*)

Get thread flags describing the thread's behaviour.

See **pthread_attr_setflags()** (p. 70) for an explanation of possible values.

Returns:

> If successful, this function returns 0. Otherwise, an error number is returned to indicate the error.

B.10.3.21. int pthread_attr_setflags (pthread_attr_t * *attr*, int *flags*)

Set thread flags describing the thread's behaviour.

Possible values are:

- `THREAD_PRIV_THRMG`: thread has access to thread management (creating, killing, renicing)
- `THREAD_PRIV_DYNMEM`: thread uses dynamic memory management
- `THREAD_PRIV_MEXT`: thread my extend its memory (i.e. the local malloc may do so)
- `THREAD_PRIV_TLSF`: thread uses TLSF for DSA, or if not set, uses DLAlloc (needs DYNMEN!) When using TLSF, online-memory extension is not allowed (the MEXT flag is ignored). So make sure to reserve enough memory at thread creation!
- `THREAD_PRIV_MODS`: thread is allowed to load program modules (needs DYNMEM and MEXT too!). Use with care! The local namespace will influence the memory consumption of the thread.
- `THREAD_PRIV_GMOD`: thread may load modules into global namespace
- `THREAD_PRIV_APPS`: thread may load applications
- `THREAD_PRIV_DRVS`: thread may manage drivers (load/unload)

Returns:

> If successful, this function returns 0. Otherwise, an error number is returned to indicate the error.

B.10.3.22. int pthread_attr_getiq (pthread_attr_t * *attr*, int * *iq*)

Get the instruction quantum.

Deprecated

The actual MERASA hardware scheduler ignores this value.

Returns:

If successful, this function returns 0. Otherwise, an error number is returned to indicate the error.

B.10.3.23. int pthread_attr_setiq (pthread_attr_t * *attr*, int *iq*)

Set the instruction quantum.

Deprecated

The actual MERASA hardware scheduler ignores this value.

This value is given to the hardware-scheduler to decide how to schedule the different threads.

Returns:

If successful, this function returns 0. Otherwise, an error number is returned to indicate the error.

B.10.3.24. int pthread_mutex_destroy (pthread_mutex_t * *mutex*)

Destroy a mutex.

The **pthread_mutex_destroy()** (p. 71) function frees the resources allocated for mutex.

Returns:

If successful, **pthread_mutex_destroy()** (p. 71) will return zero, otherwise an error number will be returned to indicate the error.

B.10.3.25. int pthread_mutex_init (pthread_mutex_t * *mutex*, const pthread_mutexattr_t * *attr*)

Initialize a mutex with specified attributes.

The **pthread_mutex_init()** (p. 71) function creates a new mutex, with attributes specified with attr. If attr is NULL the default attributes are used.

Returns:

If successful, **pthread_mutex_init()** (p. 71) will return zero and put the new mutex id into mutex, otherwise an error number will be returned to indicate the error.

B.10.3.26. int pthread_mutex_lock (pthread_mutex_t * *mutex*)

Lock a mutex and block until it becomes available.

The **pthread_mutex_lock()** (p. 72) function locks mutex. If the mutex is already locked, the calling thread will block until the mutex becomes available.

Returns:

> If successful, **pthread_mutex_lock()** (p. 72) will return zero, otherwise an error number will be returned to indicate the error.

B.10.3.27. int pthread_mutex_trylock (pthread_mutex_t * *mutex*)

Try to lock a mutex, but do not block if the mutex is locked by another thread, including the current thread.

The **pthread_mutex_trylock()** (p. 72) function locks mutex. If the mutex is already locked, **pthread_mutex_trylock()** (p. 72) will not block waiting for the mutex, but will return an error condition.

Returns:

> If successful, **pthread_mutex_trylock()** (p. 72) will return zero, otherwise an error number will be returned to indicate the error.

B.10.3.28. int pthread_mutex_unlock (pthread_mutex_t * *mutex*)

Unlock a mutex.

If the current thread holds the lock on mutex, then the **pthread_mutex_unlock()** (p. 72) function unlocks mutex.

Returns:

> If successful, **pthread_mutex_unlock()** (p. 72) will return zero, otherwise an error number will be returned to indicate the error.

B.10.3.29. int pthread_cond_broadcast (pthread_cond_t * *cond*)

Unblock all threads currently blocked on the specified condition variable.

The **pthread_cond_broadcast()** (p. 72) function unblocks all threads waiting for the condition variable cond.

Returns:

> If successful, the **pthread_cond_broadcast()** (p. 72) function will return zero, otherwise an error number will be returned to indicate the error.

B.10.3.30. int pthread_cond_destroy (pthread_cond_t * *cond*)

Destroy a condition variable.

The **pthread_cond_destroy()** (p. 72) function frees the resources allocated by the condition variable cond.

Returns:

> If successful, the **pthread_cond_destroy()** (p. 72) function will return zero, otherwise an error number will be returned to indicate the error.

B.10.3.31. int pthread_cond_init (pthread_cond_t * *cond*, const pthread_condattr_t * *attr*)

Initialize a condition variable with specified attributes.

The **pthread_cond_init()** (p. 73) function creates a new condition variable, with attributes specified with attr. If attr is NULL the default attributes are used.

Returns:

> If successful, the **pthread_cond_init()** (p. 73) function will return zero and put the new condition variable id into cond, otherwise an error number will be returned to indicate the error.

B.10.3.32. int pthread_cond_signal (pthread_cond_t * *cond*)

Unblock at least one of the threads blocked on the specified condition variable.

The **pthread_cond_signal()** (p. 73) function unblocks one thread waiting for the condition variable cond.

Returns:

> If successful, the **pthread_cond_signal()** (p. 73) function will return zero, otherwise an error number will be returned to indicate the error.

B.10.3.33. int pthread_cond_wait (pthread_cond_t *, pthread_-mutex_t * *mutex*)

Wait for a condition and lock the specified mutex.

The **pthread_cond_wait()** (p. 73) function atomically blocks the current thread waiting on the condition variable specified by cond, and releases the mutex specified by mutex. The waiting thread unblocks only after another thread calls **pthread_cond_signal()** (p. 73), or **pthread_cond_-broadcast()** (p. 72) with the same condition variable, and the current thread reacquires the lock on mutex.

Returns:

> If successful, the **pthread_cond_wait()** (p. 73) function will return zero. Other- wise an error number will be returned to indicate the error.

B.10.3.34. int pthread_barrier_destroy (pthread_barrier_t * *barrier*)

Destroy a barrier.

The pthread_barrier_destroy function will destroy barrier and release any resources that may have been allocated on its behalf.

Returns:

If successful, the **pthread_barrier_destroy()** (p. 74) function will return zero, otherwise an error number will be returned to indicate the error.

B.10.3.35. int pthread_barrier_init (pthread_barrier_t * *barrier*, const pthread_barrierattr_t * *attr*, unsigned *count*)

Initialise a barrier.

The pthread_barrier_init function will initialize a barrier with attributes specified in attr, or if it is NULL, with default attributes. The number of threads that must call pthread_barrier_wait before any of the waiting threads can be released is specified by count.

Returns:

If successful, the **pthread_barrier_init()** (p. 74) function will return zero. Other- wise an error number will be returned to indicate the error.

B.10.3.36. int pthread_barrier_wait (pthread_barrier_t * *barrier*)

Wait on a barrier.

The pthread_barrier_wait function will synchronize calling threads at the barrier. The threads will be blocked from making further progress until a sufficient number of threads calls this function. The number of threads that must call it before any of them will be released is determined by the count argument to pthread_barrier_init. Once the threads have been released the barrier will be reset.

Returns:

If successful, the **pthread_barrier_wait()** (p. 74) function will return zero. Other- wise an error number will be returned to indicate the error.

B.11. stropts.h File Reference

Control a device.

Functions

- int **ioctl** (int d, int request,...)

B.11.1. Detailed Description

Definition in file **stropts.h**.

B.11.2. Function Documentation

B.11.2.1. int ioctl (int *d*, int *request*, *...*)

Control a device.

The **ioctl()** (p. 75) system call manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g. terminals) may be controlled with **ioctl()** (p. 75) requests. The argument d must be an open file descriptor.

Valid values for request and further parameters depend on the specific device, see the driver's header file.

Returns:

If an error has occurred, a value of -1 is returned and errno is set to indicate the error.

B.12. sysmonitor.h File Reference

Type definitions to describe the state of memory.

`#include <sys/types.h>`

Data Structures

- struct **memorystatistics**
- struct **thread_memorystatistics**

Typedefs

- typedef struct **memorystatistics** ∗ **memstatptr**
- typedef struct **thread_memorystatistics** ∗ **tmemstatptr**

B.12.1. Detailed Description

Definition in file **sysmonitor.h**.

B.12.2. Typedef Documentation

B.12.2.1. typedef struct memorystatistics∗ memstatptr

A shortcut

Definition at line 68 of file sysmonitor.h.

B.12.2.2. typedef struct thread_memorystatistics∗ tmemstatptr

A shortcut

Definition at line 84 of file sysmonitor.h.

B.13. types.h File Reference

Definitions of basic data types.

`#include <stddef.h>`

Data Structures

- struct **sched_param**
- struct **pthread_mutex**
- struct **pthread_mutexattr_t**
- struct **pthread_cond**
- struct **pthread_condattr_t**
- struct **pthread_barrier_t**
- struct **pthread_barrierattr_t**
- struct **mem_cfg_data**
- struct **pthread_attr_t**

Defines

- #define **false** FALSE
- #define **true** TRUE

Typedefs

- typedef unsigned char **uint8_t**
- typedef signed char **int8_t**
- typedef unsigned short int **uint16_t**
- typedef signed short int **int16_t**
- typedef unsigned int **uint32_t**
- typedef signed int **int32_t**
- typedef unsigned long long **uint64_t**
- typedef signed long long **int64_t**
- typedef char * **address**
- typedef **int32_t** **error_t**
- typedef **uint32_t** **version_t**
- typedef signed int **ssize_t**
- typedef **int32_t** **thread_handler**
- typedef struct thread * **threadptr**
- typedef struct thread_control_block_t **tcb_t**
- typedef **uint32_t** **sched_t**

- typedef **int32_t cc_spinlock_t**
- typedef struct **pthread_mutex pthread_mutex_t**
- typedef struct **pthread_cond pthread_cond_t**
- typedef struct **mem_cfg_data memory_t**
- typedef **thread_handler pthread_t**

Enumerations

- enum **bool_t** { **FALSE** = 0, **TRUE** = 1 }
- enum **sched_policy** { **SCHED_NONE** = 0, **SCHED_HRT**, **SCHED_NRT** }

B.13.1. Detailed Description

Definition in file **types.h**.

B.13.2. Define Documentation

B.13.2.1. #define false FALSE

Definition at line 77 of file types.h.

B.13.2.2. #define true TRUE

Definition at line 78 of file types.h.

B.13.3. Typedef Documentation

B.13.3.1. typedef unsigned char uint8_t

Unsigned 8-bit integer

Definition at line 48 of file types.h.

B.13.3.2. typedef signed char int8_t

Signed 8-bit integer

Definition at line 50 of file types.h.

B.13.3.3. typedef unsigned short int uint16_t

Unsigned 16-bit integer

Definition at line 52 of file types.h.

B.13.3.4. typedef signed short int int16_t

Signed 16-bit integer

Definition at line 54 of file types.h.

B.13.3.5. typedef unsigned int uint32_t

Unsigned 32-bit integer

Definition at line 56 of file types.h.

B.13.3.6. typedef signed int int32_t

Signed 32-bit integer

Definition at line 58 of file types.h.

B.13.3.7. typedef unsigned long long uint64_t

Unsigned 64-bit integer

Definition at line 60 of file types.h.

B.13.3.8. typedef signed long long int64_t

Signed 64-bit integer

Definition at line 62 of file types.h.

B.13.3.9. typedef char* address

Data type for addresses

Definition at line 84 of file types.h.

B.13.3.10. typedef int32_t error_t

Data type for errors

Definition at line 86 of file types.h.

B.13.3.11. typedef uint32_t version_t

Data type for **driver** (p. 19) versions

Definition at line 88 of file types.h.

B.13.3.12. typedef signed int ssize_t

Data type for sizes

Definition at line 90 of file types.h.

B.13.3.13. typedef int32_t thread_handler

The thread handler is just an ID; this value is used as an offset into the TCB array.

Definition at line 96 of file types.h.

B.13.3.14. typedef struct thread∗ threadptr

Thread pointer

Definition at line 98 of file types.h.

B.13.3.15. typedef struct thread_control_block_t tcb_t

Thread control block

Definition at line 100 of file types.h.

B.13.3.16. typedef uint32_t sched_t

Scheduling information

Definition at line 106 of file types.h.

B.13.3.17. typedef int32_t cc_spinlock_t

Spinlock variable (busy waiting) for synchronisation

Definition at line 118 of file types.h.

B.13.3.18. typedef struct pthread_mutex pthread_mutex_t

B.13.3.19. typedef struct pthread_cond pthread_cond_t

B.13.3.20. typedef struct mem_cfg_data memory_t

B.13.3.21. typedef thread_handler pthread_t

POSIX thread

Definition at line 176 of file types.h.

B.13.4. Enumeration Type Documentation

B.13.4.1. enum bool_t

Boolean data type

Enumerator:

- ***FALSE***
- ***TRUE***

Definition at line 76 of file types.h.

B.13.4.2. enum sched_policy

Scheduling policy

Enumerator:

- ***SCHED_NONE***
- ***SCHED_HRT***
- ***SCHED_NRT***

Definition at line 108 of file types.h.

B.14. unistd.h File Reference

Generic read and write operations.

`#include <sys/types.h>`

Functions

- int **close** (int d)
- **ssize_t read** (int d, void *buf, size_t nbytes)
- **ssize_t write** (int d, const void *buf, size_t nbytes)

B.14.1. Detailed Description

Definition in file **unistd.h**.

B.14.2. Function Documentation

B.14.2.1. int close (int *d*)

Delete a descriptor.

The **close()** (p. 82) system call deletes a descriptor from the per-process object reference table. If this is the last reference to the underlying object, the object will be deactivated.

Returns:

The **close()** (p. 82) function returns the value 0 if successful; otherwise the value -1 is returned and the global variable errno is set to indicate the error.

B.14.2.2. ssize_t read (int *d*, void * *buf*, size_t *nbytes*)

Read input.

The **read()** (p. 82) system call attempts to read nbytes of data from the object referenced by the descriptor d into the buffer pointed to by buf.

Returns:

If successful, the number of bytes actually read is returned. Upon reading end-of-file, zero is returned. Otherwise, a -1 is returned and the global variable errno is set to indicate the error.

B.14.2.3. ssize_t write (int *d*, const void * *buf*, size_t *nbytes*)

Write output.

The **write()** (p. 82) system call attempts to write nbytes of data to the object referenced by the descriptor d from the buffer pointed to by buf.

Returns:

Upon successful completion the number of bytes which were written is returned. Otherwise a -1 is returned and the global variable errno is set to indicate the error.

B.15. vo.h File Reference

Definitions of memory regions for virtual output.

Defines

- #define **PER_VIO** (0xe0010000)
- #define **VO_ADDRESS** (PER_VIO + 0x10)
- #define **VO_WORD** (PER_VIO + 0x14)
- #define **VO_DBL_D4C** (PER_VIO + 0x20)
- #define **VO_DBL_X4C** (PER_VIO + 0x28)
- #define **VIO_PREFIX** "%#"
- #define **P64**(a) (*((**uint64_t** volatile *) a))

B.15.1. Detailed Description

Definition in file **vo.h**.

B.15.2. Define Documentation

B.15.2.1. #define PER_VIO (0xe0010000)

The base address for the virtual output.

Definition at line 44 of file vo.h.

B.15.2.2. #define VO_ADDRESS (PER_VIO + 0x10)

Bytes written to this address are put into the virtual output.

Definition at line 47 of file vo.h.

B.15.2.3. #define VO_WORD (PER_VIO + 0x14)

Write one word to STDOUT.

Definition at line 50 of file vo.h.

B.15.2.4. #define VO_DBL_D4C (PER_VIO + 0x20)

Special log, hi is interpreted as chars, lo as word dec.

Definition at line 53 of file vo.h.

B.15.2.5. #define VO_DBL_X4C (PER_VIO + 0x28)

Special log, hi is interpreted as chars, lo as word hex.

Definition at line 56 of file vo.h.

B.15.2.6. #define VIO_PREFIX "%#"

The prefix allows easy filtering of VIO output in console mode.

Definition at line 59 of file vo.h.

B.15.2.7. #define P64(a) (*((uint64_t volatile *) a))

Fast access to a 64-bit value.

Definition at line 62 of file vo.h.

References

[1] FAQ of comp.realtime. http://www.faqs.org/faqs/realtime-computing/faq/, July 1998. visited July 2007.

[2] Lea, D. A Memory Allocator. *unix/mail* (Dec. 1996).

[3] Masmano, M., Ripoll, I., Crespo, A., and Real, J. TLSF: A New Dynamic Memory Allocator for Real-Time Systems. In *ECRTS '04: Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS'04)* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 79–86.

[4] Morphew, G. Debugging complex embedded applications. http://www.ddj.com/embedded/184406044, April 2005. visited April 2008.

[5] IEEE Std 1003.1, 2004 Edition. The Open Group Base Specifications Issue 6, 2004.

[6] QNX Software Systems. http://www.qnx.com/. Visited April 2008.

[7] Uhrig, S., Maier, S., and Ungerer, T. Toward a Processor Core for Real-time Capable Autonomic Systems. In *Proceedings of the 5th IEEE International Symposium on Signal Processing and Information Technology* (Dec. 2005), pp. 19–22.