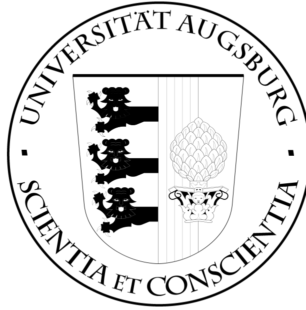
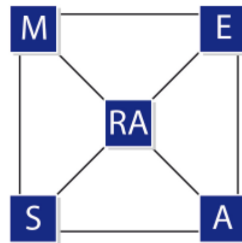
UNIVERSITÄT AUGSBURG



Basic System-Level Software for a Single-Core MERASA Processor

Florian Kluge, Julian Wolf

Report 2008-06 April 2008



INSTITUT FÜR INFORMATIK

D-86135 AUGSBURG

Copyright © Florian Kluge, Julian Wolf
Institut für Informatik
Universität Augsburg
D–86135 Augsburg, Germany
http://www.Informatik.Uni-Augsburg.DE
— all rights reserved —

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 functions of a real-time operating system.

This report presents requirements for a multi-threaded hard real-time capable system-level software in embedded systems and the transfer to the implemented MERASA / CareCore single-core processor showing details of the thread management, the dynamic memory management and the resource management. It summarizes the MERASA system-level software version 1 developed for a single multi-threaded core processor running on the CarCore / MERASA SystemC Simulator version 1.

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. The multi-core MERASA processor is currently under development. It will be adapted from a simultaneous multi-threaded (SMT) MERASA core processor that is developed based on the SMT CarCore processor [11] which is binary compatible to the Infineon TriCore.

This report describes the basic system-level software based on the CAROS architecture [3]. In this version it provides functionalities for an SMT single-core MERASA processor. The full support of the multi-core processor model will be

[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

enhanced during the next stages of the MERASA project in parallel with the multi-core development.

This report is organized 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 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 summarize 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.

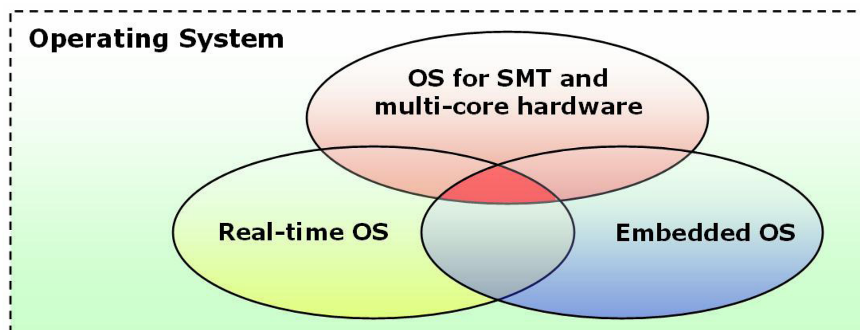


Figure 1: Combination of requirements for different OS types

Operating systems can be categorized 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 [2] we can summarize 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 [6]) that the OS has to avoid race conditions or deadlocks caused by timing problems.

2.2 Requirements for the user interface

The developed software should be geared towards common embedded operating systems. The OSEK consortium [7] defines the interface of an operating system for automotive applications. These specifications also cover communication within and between control units, as well as network management. AUTOSAR [1] as a new standard extends the OSEK specifications by more specific hardware interfaces. Its goals are the modularity, scalability, transferability and re-usability of functions to provide a standard platform for automotive systems. This enables system wide configuration and optimization to meet requirements of automotive devices. As in AUTOSAR all tasks are defined at compilation and a creation of new tasks during runtime is not prescribed, it offers a good opportunity for worst-case execution time (WCET) analysis.

To facilitate the development of applications, it may also be useful to gear functions towards the popular POSIX [8] interface. Thus, one can provide a familiar handling of parameters, return values and function names. POSIX is widely used also in the fields of embedded real-time (e.g. QNX [9]). It will be easy to port applications developed for other systems using the POSIX interface as well.

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.

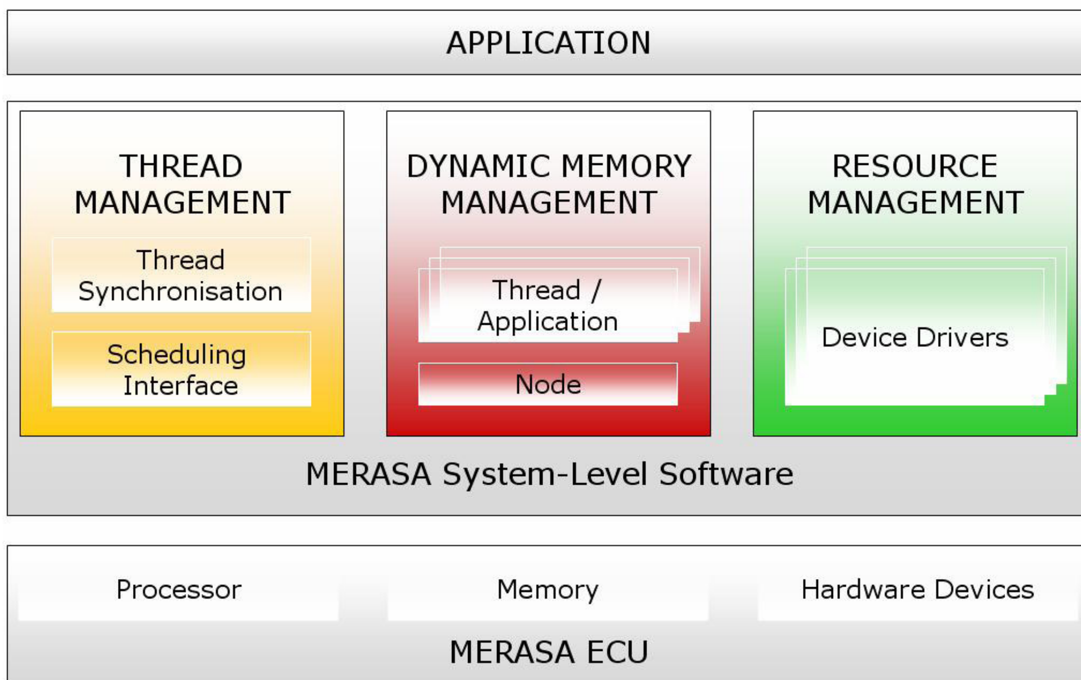


Figure 2: Architecture of the MERASA system-level software

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. At the creation of an application, resources are allocated as far as possible. 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.

4 Application Programming Interface

Here we describe the programming interface to access the MERASA system-level software (directory `include/`). It is very similar to the POSIX interface but not yet fully compliant. 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 three architectural parts of the architecture. The last part explains some common header files.

4.1 Thread Management

The Thread Management is split up into two files:

ccthread.h This header provides essential functions for the Thread Management. It allows the creation of threads and the management of the hardware thread slots, also taking the threads' priority into account. Also, thread privileges are defined here as a base for a security manager.

sync.h Here an interface to access the thread synchronisation mechanisms is defined. It is very similar to the POSIX header pthread.h and provides functions for both mutex and conditional variables.

4.2 Dynamic Memory Management

For the usage of the Dynamic Memory Management it is necessary to utilize 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 two 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.

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.

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:

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

ccthreadtypes.h In this file, several types used by the operating system's user interface are defined.

csfr.h This file provides several Core Special Function Register definitions.

log.h Especially for debugging 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/config.mk`) for compilation.

memdev.h Here, one can find definitions of memory adresses for peripheral device access.

peripheral-desc.h This header contains a base for peripheral configuration, used for the initialization of the system. By this means, device drivers can automatically be loaded at boot-up.

regcarcore.h Here, some definitions of the CarCore special registers and related data can be found.

sysmonitor.h This file provides monitoring functionalities of system parameters. It is possible to get detailed statistics on the usage of global and each thread's memory.

tcintrinsics.h In this file, some intrinsic functions are defined to facilitate application development.

types.h This header contains platform-specific definitions of integer types.

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

5 Implementation

5.1 Thread Management

Task Scheduler

As mentioned in section 2, the scheduler is the most basic part of the task and thread management. The MERASA core processor already provides a hardware-based, real-time capable thread scheduling unit. So, the development of a software scheduler is dispensable. Instead, it was necessary to integrate the hardware thread management and create an interface to deal with the hardware thread slots.

Currently the basic version of the MERASA core processor is equipped with a fixed priority scheduling of four thread slots. Hence, the basic duty for the thread management is to ensure the correct allocation and deallocation of these slots and the consistent setting of priorities. The system-level software contains functionalities to create and start threads, to change the scheduling priority of a thread or to swap the priority with another thread. As threads may be equipped with privileges like the management of drivers, the creation or modification of other threads or the enhancement of allocated memory, the thread manager also provides diagnostic tools to read status informations from threads.

Synchronisation

As a second important task, synchronisation is integrated into the Thread Management module. It features the conventional mechanisms of lock and conditional variables to avoid the simultaneous use of a common resource by critical sections. The implementation is similar to the POSIX [8] interface.

Moreover, there must be mechanisms to prevent priority inversion or even deadlocks. Our way to avoid this problem is the appliance of priority inheritance, like proposed in [10]. When a high priority thread tries to lock a resource held by a low priority thread, the low priority thread is temporary hoisted to the level of the upper thread. In parallel, the waiting thread is added to a waiting list ordered by priority, so the execution can be resumed as soon as the mutex variable is unlocked by the current owner. Providing this mechanism of inheritance, the execution of a resource-locking thread will be continued and no other thread will have possibilities to cause delays. Of course, the appliance of priority inheritance is also intertwined with the scheduler, because it directly manipulates the scheduling parameters of the involved threads.

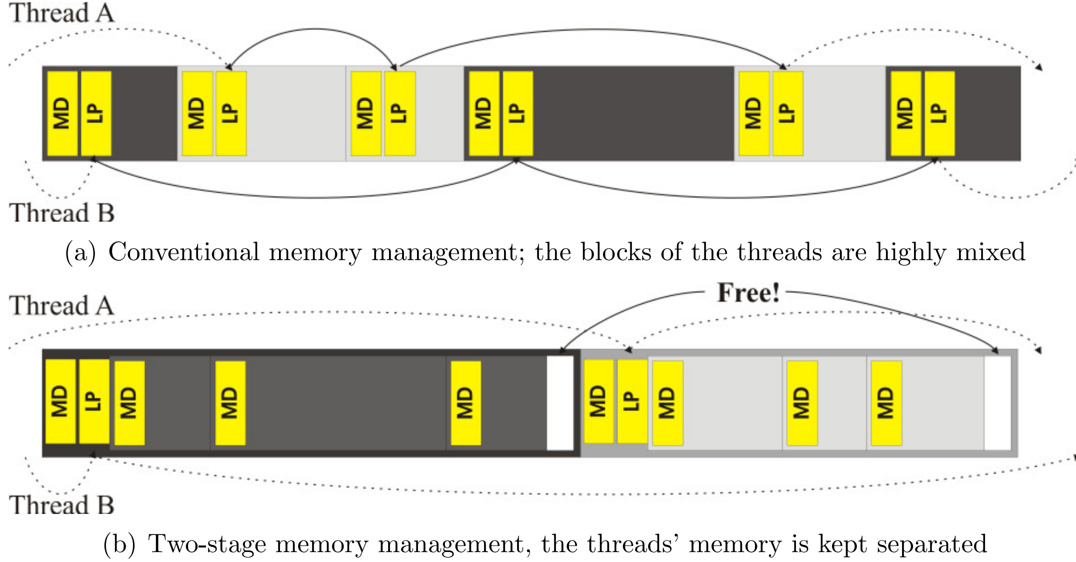
5.2 Dynamic Memory Management

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 minimize 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 usually done before a thread is started, there are no influences on the real-time behaviour of the system. On the *thread* layer the memory management allocates memory to the executed program in the specific thread. This can be done 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 3. 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 3(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 3: 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 [4] (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 [5], 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 as well as a module manager, is geared towards the POSIX standard [8]. The driver interface provides functionalities to `install` and initalize device drivers from an ELF[2] image and to `remove` them from the system. It also contains the generic `open` / `close` operations as well as access to the device (`read` / `write` operations) and 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 synchronisation. For this, the Thread Manager already provides solutions. However, in future the Resource Management may extend these mechanisms or implement better solutions.

In general, there is no limitation on the number of drivers supported by the resource manager. However, this leads to the use of dynamic data structures within the manager, which cannot guarantee a bounded timing behaviour for device accesses. For the use in real-time application, an *a priori* resolution of the used devices must be made. As the number of devices an application uses is limited and 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. The device access for non-real-time applications can still be done over a dynamic name resolution or similar.

[2] **E**xecutable and **L**inkable **F**ormat, a common standard file format for executables, object code and shared libraries

6 Conclusion

In this report we presented the basic system-level software for a single-core MERASA processor. It represents an abstraction layer between application software and hard real-time capable SMT hardware. The architecture consists of three main parts: The *Thread Manager* provides an interface to the hardware-based real-time capable thread scheduling unit of the MERASA core processor and mechanisms for thread synchronisation. The *Dynamic Memory Management* minimizes 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 summarize how they are fulfilled in detail: The MERASA system-level software is multi-threaded and preemptible. It provides mechanisms for thread synchronisation, in particular a system of priority inheritance. 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. 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 multi-threaded hardware, the software provides mechanisms for synchronisation. So it is easy for an application programmer to avoid race conditions between different threads running in parallel.

In the future, we want to extend and improve the system-level software. Concerning the thread management, especially the synchronisation unit, it is necessary to guarantee an avoidance of deadlocks. This can be achieved by the usage of the priority ceiling protocol. Moreover, one should consider the benefit of changing all functions to be a subset of the POSIX interface and provide full compatibility.

However, during the next steps of the MERASA project the full support of the multi-core processor model will be enhanced in parallel with the multi-core development.

References

[1] AUTOSAR AUTomotive Open System ARchitecture. http://autosar.org. Visited April 2008.

[2] FAQ of comp.realtime. http://www.faqs.org/faqs/realtime-computing/faq/, July 1998. visited April 2008.

[3] Kluge, F., Mische, J., Uhrig, S., and Ungerer, T. An operating system architecture for organic computing in embedded real-time systems. In *Proceedings of the 5th International Conference on Autonomic and Trusted Computing (ATC-08)* (Oslo, Norway, June 2008), accepted for publication.

[4] Lea, D. A memory allocator. *Unix/Mail 6/96* (1996).

[5] 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.

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

[7] OSEK VDX Portal. http://www.osek-vdx.org. Visited April 2008.

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

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

[10] Sha, L., Rajkumar, R., and Lehoczky, J. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers 39*, 9 (1990), 1175–1185.

[11] 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* (Athens, Greece, Dec. 2005), pp. 19–22.

ANNEX

MERASA System Level Software - Reference Manual

Generated by Doxygen 1.5.1

1 MERASA System Level Software Data Structure Documentation

1.1 ccthread_cond Struct Reference

Definition of a conditional variable.

`#include <ccthreadtypes.h>`

Data Fields

- **ccthread_mutex_t * mutex**

1.1.1 Detailed Description

Definition at line 103 of file ccthreadtypes.h.

1.1.2 Field Documentation

1.1.2.1 ccthread_mutex_t* ccthread_cond::mutex

The mutex associated with the conditional variable.

Definition at line 105 of file ccthreadtypes.h.

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

- **ccthreadtypes.h**

1.2 ccthread_mutex Struct Reference

The mutex definition.

`#include <ccthreadtypes.h>`

Data Fields

- **cc_spinlock_t the_lock**
- **cc_spinlock_t guard**
- **thread_handler owner**
- **sched_t prev_sched**
- **int32_t waiting_list** [4]

1.2.1 Detailed Description

A high-level implementation of a mutex variable

Definition at line 87 of file ccthreadtypes.h.

1.2.2 Field Documentation

1.2.2.1 cc_spinlock_t ccthread_mutex::the_lock

Main spinlock.

Definition at line 89 of file ccthreadtypes.h.

1.2.2.2 cc_spinlock_t ccthread_mutex::guard

Additional spinlock granting the access to the waiting lists.

Definition at line 91 of file ccthreadtypes.h.

1.2.2.3 thread_handler ccthread_mutex::owner

The handler of the locking thread.

Definition at line 93 of file ccthreadtypes.h.

1.2.2.4 sched_t ccthread_mutex::prev_sched

Original scheduling parameters before an eventual hoisting caused by priority inheritance.

Definition at line 95 of file ccthreadtypes.h.

1.2.2.5 int32_t ccthread_mutex::waiting_list[4]

A list of threads waiting for a lock.

Definition at line 97 of file ccthreadtypes.h.

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

- **ccthreadtypes.h**

1.3 driver Struct Reference

This struct describes a device driver.

`#include <driver.h>`

Data Fields

- const char * **name**
- **version_t version**
- **drv_init_fn_t init**
- **drv_cleanup_fn_t cleanup**

- **drvif_t** * **ops**
- void * **pi**
- **ccthread_mutex_t lock**
- **thread_handler owner**
- **driver** * **sp_next**

1.3.1 Detailed Description

Do not use it directly, instead publish your driver using the **DRIVER** (p. 57) macro!

Definition at line 88 of file driver.h.

1.3.2 Field Documentation

1.3.2.1 const char* driver::name

Name of the driver.

Definition at line 90 of file driver.h.

1.3.2.2 version_t driver::version

Version of this driver. This field is currently not used, it should be set to 0x10.

Definition at line 91 of file driver.h.

1.3.2.3 drv_init_fn_t driver::init

Initialisation function.

Definition at line 93 of file driver.h.

1.3.2.4 drv_cleanup_fn_t driver::cleanup

Cleanup function; currently unused.

Definition at line 94 of file driver.h.

1.3.2.5 drvif_t* driver::ops

The operations struct.

Definition at line 95 of file driver.h.

1.3.2.6 void* driver::pi

Process image, only for internal use.

Definition at line 96 of file driver.h.

1.3.2.7 ccthread_mutex_t driver::lock

The lock for this driver.

Definition at line 97 of file driver.h.

1.3.2.8 thread_handler driver::owner

Current owner of this driver, -1 if unused.

Definition at line 98 of file driver.h.

1.3.2.9 struct driver* driver::sp_next

Pointer for a thread's driver stack.

Definition at line 99 of file driver.h.

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

- **driver.h**

1.4 driver_interface Struct Reference

This struct holds the functions a driver must implement, i.e. open, close, read, write and ioctl.

`#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**

1.4.1 Detailed Description

Definition at line 74 of file driver.h.

1.4.2 Field Documentation

1.4.2.1 drv_open_fn driver_interface::open

Definition at line 75 of file driver.h.

1.4.2.2 drv_close_fn driver_interface::close

Definition at line 76 of file driver.h.

1.4.2.3 drv_read_fn driver_interface::read

Definition at line 77 of file driver.h.

1.4.2.4 drv_write_fn driver_interface::write

Definition at line 78 of file driver.h.

1.4.2.5 drv_ioctl_fn driver_interface::ioctl

Definition at line 79 of file driver.h.

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

- **driver.h**

1.5 mem_cfg_data Struct Reference

Structure to describe the configuration of memory.

`#include <memory-desc.h>`

Data Fields

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

1.5.1 Detailed Description

Definition at line 51 of file memory-desc.h.

1.5.2 Field Documentation

1.5.2.1 char* mem_cfg_data::begin

First byte of memory area.

Definition at line 52 of file memory-desc.h.

1.5.2.2 char* mem_cfg_data::end

First byte after the memory area.

Definition at line 53 of file memory-desc.h.

1.5.2.3 size_t mem_cfg_data::length

Length of memory area.

Definition at line 54 of file memory-desc.h.

1.5.2.4 uint32_t mem_cfg_data::access_cycles

Average number of cycles to accesss one word of this memory.

Definition at line 55 of file memory-desc.h.

1.5.2.5 int32_t mem_cfg_data::cost

Cost for use of this memory.

Definition at line 56 of file memory-desc.h.

1.5.2.6 uint32_t mem_cfg_data::flags

Flags for further use.

Definition at line 57 of file memory-desc.h.

1.5.2.7 char* mem_cfg_data::brk

Current break (allocated up to this address).

Definition at line 58 of file memory-desc.h.

1.5.2.8 void* mem_cfg_data::binlist

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

Definition at line 59 of file memory-desc.h.

1.5.2.9 ccthread_mutex_t mem_cfg_data::mutex

Mutex for gmalloc.

Definition at line 60 of file memory-desc.h.

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

- **memory-desc.h**

1.6 memorystatistics Struct Reference

Memorystatistics describing the state of global memory.

`#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**

1.6.1 Detailed Description

Definition at line 48 of file sysmonitor.h.

1.6.2 Field Documentation

1.6.2.1 size_t memorystatistics::used

Total inuse.

Definition at line 49 of file sysmonitor.h.

1.6.2.2 uint32_t memorystatistics::freepages_count

The count of free pages.

Definition at line 50 of file sysmonitor.h.

1.6.2.3 size_t memorystatistics::freepages_size

Size of all free pages.

Definition at line 51 of file sysmonitor.h.

1.6.2.4 size_t memorystatistics::min_freepagesize

Size of greatest free page.

Definition at line 52 of file sysmonitor.h.

1.6.2.5 size_t memorystatistics::max_freepagesize

Size of smallest free page.

Definition at line 53 of file sysmonitor.h.

1.6.2.6 size_t memorystatistics::max_ext

Maximum amount of memory that can be allocated using sbrk().

Definition at line 54 of file sysmonitor.h.

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

- **sysmonitor.h**

1.7 per_cfg_data Struct Reference

Struct for peripheral configuration.

`#include <peripheral-desc.h>`

Data Fields

- char * **dev_id**
- void * **base_address**
- char * **drv_id**
- **version_t min_version**

1.7.1 Detailed Description

Definition at line 46 of file peripheral-desc.h.

1.7.2 Field Documentation

1.7.2.1 char* per_cfg_data::dev_id

The ID of the device.

Definition at line 47 of file peripheral-desc.h.

1.7.2.2 void* per_cfg_data::base_address

The base address.

Definition at line 48 of file peripheral-desc.h.

1.7.2.3 char* per_cfg_data::drv_id

The ID of the driver.

Definition at line 49 of file peripheral-desc.h.

1.7.2.4 version_t per_cfg_data::min_version

The minimum version needed.

Definition at line 50 of file peripheral-desc.h.

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

- **peripheral-desc.h**

1.8 thread_memorystatistics Struct Reference

Thread_memorystatistics describing the state of a thread's memory.

`#include <sysmonitor.h>`

Data Fields

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

1.8.1 Detailed Description

Definition at line 64 of file sysmonitor.h.

1.8.2 Field Documentation

1.8.2.1 size_t thread_memorystatistics::reserved

Reserved size.

Definition at line 65 of file sysmonitor.h.

1.8.2.2 size_t thread_memorystatistics::used

Used size.

Definition at line 66 of file sysmonitor.h.

1.8.2.3 uint32_t thread_memorystatistics::freechunks_count

The count of free chunks.

Definition at line 67 of file sysmonitor.h.

1.8.2.4 size_t thread_memorystatistics::min_freechunksize

The minimum size of free chunks.

Definition at line 68 of file sysmonitor.h.

1.8.2.5 size_t thread_memorystatistics::max_freechunksize

The maximum size of free chunks.

Definition at line 69 of file sysmonitor.h.

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

- **sysmonitor.h**

2 MERASA System Level Software File Documentation

2.1 caros.h File Reference

Containing ALL header files!

```
#include <ccerrno.h>
#include <ccthread.h>
#include <ccthreadtypes.h>
#include <csfr.h>
#include <driver.h>
#include <drivermanager.h>
#include <log.h>
#include <memory-desc.h>
#include <memory.h>
#include <module.h>
#include <modulemanager.h>
#include <peripheral-desc.h>
#include <regcarcore.h>
#include <sync.h>
#include <sysmonitor.h>
#include <tcintrinsics.h>
#include <types.h>
```

2.1.1 Detailed Description

Definition in file **caros.h**.

2.2 ccerrno.h File Reference

Definition of error numbers and types.

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

Defines

- #define **CCERRNO_H_** 1
- #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
- #define **EDEADLK** 35
- #define **ENAMETOOLONG** 36
- #define **ENOLCK** 37
- #define **ENOSYS** 38
- #define **ENOTEMPTY** 39
- #define **ELOOP** 40

- #define **EWOULDBLOCK** EAGAIN
- #define **ENOMSG** 42
- #define **EIDRM** 43
- #define **ECHRNG** 44
- #define **EL2NSYNC** 45
- #define **EL3HLT** 46
- #define **EL3RST** 47
- #define **ELNRNG** 48
- #define **EUNATCH** 49
- #define **ENOCSI** 50
- #define **EL2HLT** 51
- #define **EBADE** 52
- #define **EBADR** 53
- #define **EXFULL** 54
- #define **ENOANO** 55
- #define **EBADRQC** 56
- #define **EBADSLT** 57
- #define **EDEADLOCK** EDEADLK
- #define **EBFONT** 59
- #define **ENOSTR** 60
- #define **ENODATA** 61
- #define **ETIME** 62
- #define **ENOSR** 63
- #define **ENONET** 64
- #define **ENOPKG** 65
- #define **EREMOTE** 66
- #define **ENOLINK** 67
- #define **EADV** 68
- #define **ESRMNT** 69
- #define **ECOMM** 70
- #define **EPROTO** 71
- #define **EMULTIHOP** 72
- #define **EDOTDOT** 73
- #define **EBADMSG** 74
- #define **EOVERFLOW** 75
- #define **ENOTUNIQ** 76
- #define **EBADFD** 77
- #define **EREMCHG** 78
- #define **ELIBACC** 79

- #define **ELIBBAD** 80
- #define **ELIBSCN** 81
- #define **ELIBMAX** 82
- #define **ELIBEXEC** 83
- #define **EILSEQ** 84
- #define **ERESTART** 85
- #define **ESTRPIPE** 86
- #define **EUSERS** 87
- #define **ENOTSOCK** 88
- #define **EDESTADDRREQ** 89
- #define **EMSGSIZE** 90
- #define **EPROTOTYPE** 91
- #define **ENOPROTOOPT** 92
- #define **EPROTONOSUPPORT** 93
- #define **ESOCKTNOSUPPORT** 94
- #define **EOPNOTSUPP** 95
- #define **EPFNOSUPPORT** 96
- #define **EAFNOSUPPORT** 97
- #define **EADDRINUSE** 98
- #define **EADDRNOTAVAIL** 99
- #define **ENETDOWN** 100
- #define **ENETUNREACH** 101
- #define **ENETRESET** 102
- #define **ECONNABORTED** 103
- #define **ECONNRESET** 104
- #define **ENOBUFS** 105
- #define **EISCONN** 106
- #define **ENOTCONN** 107
- #define **ESHUTDOWN** 108
- #define **ETOOMANYREFS** 109
- #define **ETIMEDOUT** 110
- #define **ECONNREFUSED** 111
- #define **EHOSTDOWN** 112
- #define **EHOSTUNREACH** 113
- #define **EALREADY** 114
- #define **EINPROGRESS** 115
- #define **ESTALE** 116
- #define **EUCLEAN** 117
- #define **ENOTNAM** 118

- #define **ENAVAIL** 119
- #define **EISNAM** 120
- #define **EREMOTEIO** 121
- #define **EDQUOT** 122
- #define **ENOMEDIUM** 123
- #define **EMEDIUMTYPE** 124
- #define **ECANCELED** 125
- #define **ENOKEY** 126
- #define **EKEYEXPIRED** 127
- #define **EKEYREVOKED** 128
- #define **EKEYREJECTED** 129
- #define **EOWNERDEAD** 130
- #define **ENOTRECOVERABLE** 131

Typedefs

- typedef **int32_t error_t**

Functions

- **error_t get_errno** ()
  Returns the error nummer.
- void **set_errno** (**error_t** errno)
  Sets a specified error number.

2.2.1 Detailed Description

This file provides definitions of error numbers and types. We use the POSIX error number definitions.

Definition in file **ccerrno.h**.

2.2.2 Define Documentation

2.2.2.1 #define CCERRNO_H_ 1

Definition at line 32 of file ccerrno.h.

2.2.2.2 #define E2BIG 7

Argument list too long.

Definition at line 63 of file ccerrno.h.

2.2.2.3 #define E_OK 0

No error.

Definition at line 45 of file ccerrno.h.

2.2.2.4 #define EACCES 13

Permission denied.

Definition at line 75 of file ccerrno.h.

2.2.2.5 #define EADDRINUSE 98

Address already in use.

Definition at line 249 of file ccerrno.h.

2.2.2.6 #define EADDRNOTAVAIL 99

Cannot assign requested address.

Definition at line 251 of file ccerrno.h.

2.2.2.7 #define EADV 68

Advertise error.

Definition at line 189 of file ccerrno.h.

2.2.2.8 #define EAFNOSUPPORT 97

Address family not supported by protocol.

Definition at line 247 of file ccerrno.h.

2.2.2.9 #define EAGAIN 11

Try again.

Definition at line 71 of file ccerrno.h.

2.2.2.10 #define EALREADY 114

Operation already in progress.

Definition at line 281 of file ccerrno.h.

2.2.2.11 #define EBADE 52

Invalid exchange.

Definition at line 157 of file ccerrno.h.

2.2.2.12 #define EBADF 9

Bad file number.

Definition at line 67 of file ccerrno.h.

2.2.2.13 #define EBADFD 77

File descriptor in bad state.

Definition at line 207 of file ccerrno.h.

2.2.2.14 #define EBADMSG 74

Not a data message.

Definition at line 201 of file ccerrno.h.

2.2.2.15 #define EBADR 53

Invalid request descriptor.

Definition at line 159 of file ccerrno.h.

2.2.2.16 #define EBADRQC 56

Invalid request code.

Definition at line 165 of file ccerrno.h.

2.2.2.17 #define EBADSLT 57

Invalid slot.

Definition at line 167 of file ccerrno.h.

2.2.2.18 #define EBFONT 59

Bad font file format.

Definition at line 171 of file ccerrno.h.

2.2.2.19 #define EBUSY 16

Device or resource busy.

Definition at line 81 of file ccerrno.h.

2.2.2.20 #define ECANCELED 125

Operation Canceled.

Definition at line 303 of file ccerrno.h.

2.2.2.21 #define ECHILD 10

No child processes.

Definition at line 69 of file ccerrno.h.

2.2.2.22 #define ECHRNG 44

Channel number out of range.

Definition at line 141 of file ccerrno.h.

2.2.2.23 #define ECOMM 70

Communication error on send.

Definition at line 193 of file ccerrno.h.

2.2.2.24 #define ECONNABORTED 103

Software caused connection abort.

Definition at line 259 of file ccerrno.h.

2.2.2.25 #define ECONNREFUSED 111

Connection refused.

Definition at line 275 of file ccerrno.h.

2.2.2.26 #define ECONNRESET 104

Connection reset by peer.

Definition at line 261 of file ccerrno.h.

2.2.2.27 #define EDEADLK 35

Resource deadlock would occur.

Definition at line 123 of file ccerrno.h.

2.2.2.28 #define EDEADLOCK EDEADLK

Definition at line 169 of file ccerrno.h.

2.2.2.29 #define EDESTADDRREQ 89

Destination address required.

Definition at line 231 of file ccerrno.h.

2.2.2.30 #define EDOM 33

Math argument out of domain of func.

Definition at line 115 of file ccerrno.h.

2.2.2.31 #define EDOTDOT 73

RFS specific error.

Definition at line 199 of file ccerrno.h.

2.2.2.32 #define EDQUOT 122

Quota exceeded.

Definition at line 297 of file ccerrno.h.

2.2.2.33 #define EEXIST 17

File exists.

Definition at line 83 of file ccerrno.h.

2.2.2.34 #define EFAULT 14

Bad address.

Definition at line 77 of file ccerrno.h.

2.2.2.35 #define EFBIG 27

File too large.

Definition at line 103 of file ccerrno.h.

2.2.2.36 #define EHOSTDOWN 112

Host is down.

Definition at line 277 of file ccerrno.h.

2.2.2.37 #define EHOSTUNREACH 113

No route to host.

Definition at line 279 of file ccerrno.h.

2.2.2.38 #define EIDRM 43

Identifier removed.

Definition at line 139 of file ccerrno.h.

2.2.2.39 #define EILSEQ 84

Illegal byte sequence.

Definition at line 221 of file ccerrno.h.

2.2.2.40 #define EINPROGRESS 115

Operation now in progress.

Definition at line 283 of file ccerrno.h.

2.2.2.41 #define EINTR 4

Interrupted system call.

Definition at line 57 of file ccerrno.h.

2.2.2.42 #define EINVAL 22

Invalid argument.

Definition at line 93 of file ccerrno.h.

2.2.2.43 #define EIO 5

I/O error.

Definition at line 59 of file ccerrno.h.

2.2.2.44 #define EISCONN 106

Transport endpoint is already connected.

Definition at line 265 of file ccerrno.h.

2.2.2.45 #define EISDIR 21

Is a directory.

Definition at line 91 of file ccerrno.h.

2.2.2.46 #define EISNAM 120

Is a named type file.

Definition at line 293 of file ccerrno.h.

2.2.2.47 #define EKEYEXPIRED 127

Key has expired.

Definition at line 307 of file ccerrno.h.

2.2.2.48 #define EKEYREJECTED 129

Key was rejected by service.

Definition at line 311 of file ccerrno.h.

2.2.2.49 #define EKEYREVOKED 128

Key has been revoked.

Definition at line 309 of file ccerrno.h.

2.2.2.50 #define EL2HLT 51

Level 2 halted.

Definition at line 155 of file ccerrno.h.

2.2.2.51 #define EL2NSYNC 45

Level 2 not synchronized.

Definition at line 143 of file ccerrno.h.

2.2.2.52 #define EL3HLT 46

Level 3 halted.

Definition at line 145 of file ccerrno.h.

2.2.2.53 #define EL3RST 47

Level 3 reset.

Definition at line 147 of file ccerrno.h.

2.2.2.54 #define ELIBACC 79

Can not access a needed shared library.

Definition at line 211 of file ccerrno.h.

2.2.2.55 #define ELIBBAD 80

Accessing a corrupted shared library.

Definition at line 213 of file ccerrno.h.

2.2.2.56 #define ELIBEXEC 83

Cannot exec a shared library directly.

Definition at line 219 of file ccerrno.h.

2.2.2.57 #define ELIBMAX 82

Attempting to link in too many shared libraries.

Definition at line 217 of file ccerrno.h.

2.2.2.58 #define ELIBSCN 81

.lib section in a.out corrupted.

Definition at line 215 of file ccerrno.h.

2.2.2.59 #define ELNRNG 48

Link number out of range.

Definition at line 149 of file ccerrno.h.

2.2.2.60 #define ELOOP 40

Too many symbolic links encountered.

Definition at line 133 of file ccerrno.h.

2.2.2.61 #define EMEDIUMTYPE 124

Wrong medium type.

Definition at line 301 of file ccerrno.h.

2.2.2.62 #define EMFILE 24

Too many open files.

Definition at line 97 of file ccerrno.h.

2.2.2.63 #define EMLINK 31

Too many links.

Definition at line 111 of file ccerrno.h.

2.2.2.64 #define EMSGSIZE 90

Message too long.

Definition at line 233 of file ccerrno.h.

2.2.2.65 #define EMULTIHOP 72

Multihop attempted.

Definition at line 197 of file ccerrno.h.

2.2.2.66 #define ENAMETOOLONG 36

File name too long.

Definition at line 125 of file ccerrno.h.

2.2.2.67 #define ENAVAIL 119

No XENIX semaphores available.

Definition at line 291 of file ccerrno.h.

2.2.2.68 #define ENETDOWN 100

Network is down.

Definition at line 253 of file ccerrno.h.

2.2.2.69 #define ENETRESET 102

Network dropped connection because of reset.

Definition at line 257 of file ccerrno.h.

2.2.2.70 #define ENETUNREACH 101

Network is unreachable.

Definition at line 255 of file ccerrno.h.

2.2.2.71 #define ENFILE 23

File table overflow.

Definition at line 95 of file ccerrno.h.

2.2.2.72 #define ENOANO 55

No anode.

Definition at line 163 of file ccerrno.h.

2.2.2.73 #define ENOBUFS 105

No buffer space available.

Definition at line 263 of file ccerrno.h.

2.2.2.74 #define ENOCSI 50

No CSI structure available.

Definition at line 153 of file ccerrno.h.

2.2.2.75 #define ENODATA 61

No data available.

Definition at line 175 of file ccerrno.h.

2.2.2.76 #define ENODEV 19

No such device.

Definition at line 87 of file ccerrno.h.

2.2.2.77 #define ENOENT 2

No such file or directory.

Definition at line 53 of file ccerrno.h.

2.2.2.78 #define ENOEXEC 8

Exec format error.

Definition at line 65 of file ccerrno.h.

2.2.2.79 #define ENOKEY 126

Required key not available.

Definition at line 305 of file ccerrno.h.

2.2.2.80 #define ENOLCK 37

No record locks available.

Definition at line 127 of file ccerrno.h.

2.2.2.81 #define ENOLINK 67

Link has been severed.

Definition at line 187 of file ccerrno.h.

2.2.2.82 #define ENOMEDIUM 123

No medium found.

Definition at line 299 of file ccerrno.h.

2.2.2.83 #define ENOMEM 12

Out of memory.

Definition at line 73 of file ccerrno.h.

2.2.2.84 #define ENOMSG 42

No message of desired type.

Definition at line 137 of file ccerrno.h.

2.2.2.85 #define ENONET 64

Machine is not on the network.

Definition at line 181 of file ccerrno.h.

2.2.2.86 #define ENOPKG 65

Package not installed.

Definition at line 183 of file ccerrno.h.

2.2.2.87 #define ENOPROTOOPT 92

Protocol not available.

Definition at line 237 of file ccerrno.h.

2.2.2.88 #define ENOSPC 28

No space left on device.

Definition at line 105 of file ccerrno.h.

2.2.2.89 #define ENOSR 63

Out of streams resources.

Definition at line 179 of file ccerrno.h.

2.2.2.90 #define ENOSTR 60

Device not a stream.

Definition at line 173 of file ccerrno.h.

2.2.2.91 #define ENOSYS 38

Function not implemented.

Definition at line 129 of file ccerrno.h.

2.2.2.92 #define ENOTBLK 15

Block device required.

Definition at line 79 of file ccerrno.h.

2.2.2.93 #define ENOTCONN 107

Transport endpoint is not connected.

Definition at line 267 of file ccerrno.h.

2.2.2.94 #define ENOTDIR 20

Not a directory.

Definition at line 89 of file ccerrno.h.

2.2.2.95 #define ENOTEMPTY 39

Directory not empty.

Definition at line 131 of file ccerrno.h.

2.2.2.96 #define ENOTNAM 118

Not a XENIX named type file.

Definition at line 289 of file ccerrno.h.

2.2.2.97 #define ENOTRECOVERABLE 131

State not recoverable.

Definition at line 317 of file ccerrno.h.

2.2.2.98 #define ENOTSOCK 88

Socket operation on non-socket.

Definition at line 229 of file ccerrno.h.

2.2.2.99 #define ENOTTY 25

Not a typewriter.

Definition at line 99 of file ccerrno.h.

2.2.2.100 #define ENOTUNIQ 76

Name not unique on network.

Definition at line 205 of file ccerrno.h.

2.2.2.101 #define ENXIO 6

No such device or address.

Definition at line 61 of file ccerrno.h.

2.2.2.102 #define EOPNOTSUPP 95

Operation not supported on transport endpoint.

Definition at line 243 of file ccerrno.h.

2.2.2.103 #define EOVERFLOW 75

Value too large for defined data type.

Definition at line 203 of file ccerrno.h.

2.2.2.104 #define EOWNERDEAD 130

Owner died.

Definition at line 315 of file ccerrno.h.

2.2.2.105 #define EPERM 1

Operation not permitted.

Definition at line 51 of file ccerrno.h.

2.2.2.106 #define EPFNOSUPPORT 96

Protocol family not supported.

Definition at line 245 of file ccerrno.h.

2.2.2.107 #define EPIPE 32

Broken pipe.

Definition at line 113 of file ccerrno.h.

2.2.2.108 #define EPROTO 71

Protocol error.

Definition at line 195 of file ccerrno.h.

2.2.2.109 #define EPROTONOSUPPORT 93

Protocol not supported.

Definition at line 239 of file ccerrno.h.

2.2.2.110 #define EPROTOTYPE 91

Protocol wrong type for socket.

Definition at line 235 of file ccerrno.h.

2.2.2.111 #define ERANGE 34

Math result not representable.

Definition at line 117 of file ccerrno.h.

2.2.2.112 #define EREMCHG 78

Remote address changed.

Definition at line 209 of file ccerrno.h.

2.2.2.113 #define EREMOTE 66

Object is remote.

Definition at line 185 of file ccerrno.h.

2.2.2.114 #define EREMOTEIO 121

Remote I/O error.

Definition at line 295 of file ccerrno.h.

2.2.2.115 #define ERESTART 85

Interrupted system call should be restarted.

Definition at line 223 of file ccerrno.h.

2.2.2.116 #define EROFS 30

Read-only file system.

Definition at line 109 of file ccerrno.h.

2.2.2.117 #define ESHUTDOWN 108

Cannot send after transport endpoint shutdown.

Definition at line 269 of file ccerrno.h.

2.2.2.118 #define ESOCKTNOSUPPORT 94

Socket type not supported.

Definition at line 241 of file ccerrno.h.

2.2.2.119 #define ESPIPE 29

Illegal seek.

Definition at line 107 of file ccerrno.h.

2.2.2.120 #define ESRCH 3

No such process.

Definition at line 55 of file ccerrno.h.

2.2.2.121 #define ESRMNT 69

Srmount error.

Definition at line 191 of file ccerrno.h.

2.2.2.122 #define ESTALE 116

Stale NFS file handle.

Definition at line 285 of file ccerrno.h.

2.2.2.123 #define ESTRPIPE 86

Streams pipe error.

Definition at line 225 of file ccerrno.h.

2.2.2.124 #define ETIME 62

Timer expired.

Definition at line 177 of file ccerrno.h.

2.2.2.125 #define ETIMEDOUT 110

Connection timed out.

Definition at line 273 of file ccerrno.h.

2.2.2.126 #define ETOOMANYREFS 109

Too many references: cannot splice.

Definition at line 271 of file ccerrno.h.

2.2.2.127 #define ETXTBSY 26

Text file busy.

Definition at line 101 of file ccerrno.h.

2.2.2.128 #define EUCLEAN 117

Structure needs cleaning.

Definition at line 287 of file ccerrno.h.

2.2.2.129 #define EUNATCH 49

Protocol driver not attached.

Definition at line 151 of file ccerrno.h.

2.2.2.130 #define EUSERS 87

Too many users.

Definition at line 227 of file ccerrno.h.

2.2.2.131 #define EWOULDBLOCK EAGAIN

Operation would block.

Definition at line 135 of file ccerrno.h.

2.2.2.132 #define EXDEV 18

Cross-device link.

Definition at line 85 of file ccerrno.h.

2.2.2.133 #define EXFULL 54

Exchange full.

Definition at line 161 of file ccerrno.h.

2.2.3 Typedef Documentation

2.2.3.1 typedef int32_t error_t

Structure to represent an error.

Definition at line 324 of file ccerrno.h.

2.2.4 Function Documentation

2.2.4.1 error_t get_errno ()

2.2.4.2 void set_errno (error_t *errno*)

2.3 ccthread.h File Reference

Functions for thread management, e.g. creation, start and scheduling of a thread.

```
#include <ccthreadtypes.h>
#include <csfr.h>
#include <types.h>
#include <memory-desc.h>
```

Defines

- #define **YIELD**() __asm(".word 0x0440000d\n")
- #define **THREAD_PRIV_RT** 0x00000001
- #define **THREAD_PRIV_DYNMEM** 0x00000002
- #define **THREAD_PRIV_MEXT** 0x00000004
- #define **THREAD_PRIV_MODS** 0x0000000e
- #define **THREAD_PRIV_TLSF** 0x00000010
- #define **THREAD_PRIV_THRMG** 0x00000020
- #define **THREAD_PRIV_COMM** 0x00000040
- #define **THREAD_PRIV_DRVMG** 0x00000080
- #define **THREAD_PRIV_GMOD** 0x00000100
- #define **get_thread**() get_current_thread_handler()

  Returns a handler for the current thread.

Functions

- **thread_handler create_thread** (void(*thread_func)(void), **uint32_t** thread_-flags, **memory_t** *mem, **uint32_t** thread_mem, **sched_t** sched_params)

  Creates a new thread.

- **uint32_t start_thread** (**thread_handler** handler)

  Starts a specific thread.

- **uint32_t renice** (**thread_handler** th, **sched_t** new_params)

  Changes the scheduling policy of a running thread.

- **uint32_t get_priority** (**thread_handler** th)

  Reads the priority of a thread.

- void **swap_priorities** (**thread_handler** th1, **thread_handler** th2)

Swaps the priorities of two different threads.

- **int32_t join** (**thread_handler** thread)
  Waits for thread termination.
- **thread_handler get_current_thread_handler** ()
  Returns a handler for the current thread.
- **bool_t has_privilege** (**uint32_t** priv)
  Performs a check if a thread has a special defined privilege.

2.3.1 Detailed Description

In this file the most important functions for a good management of threads are defined. So a thread can be created, started and the scheduling priority can be changed (reniced) or swapped with another thread.

Definition in file **ccthread.h**.

2.3.2 Define Documentation

2.3.2.1 #define get_thread() get_current_thread_handler()

Returns:

The handler of the current thread.

Definition at line 225 of file ccthread.h.

2.3.2.2 #define THREAD_PRIV_COMM 0x00000040

The thread may use communication channels.

Definition at line 81 of file ccthread.h.

2.3.2.3 #define THREAD_PRIV_DRVMG 0x00000080

The thread may manage drivers (load/unload).

Definition at line 83 of file ccthread.h.

2.3.2.4 #define THREAD_PRIV_DYNMEM 0x00000002

The thread uses dynamic memory management.

Definition at line 65 of file ccthread.h.

2.3.2.5 #define THREAD_PRIV_GMOD 0x00000100

The thread may load modules into global namespace.

Definition at line 85 of file ccthread.h.

2.3.2.6 #define THREAD_PRIV_MEXT 0x00000004

The thread may extend its memory (i.e. the local malloc may do so).

Definition at line 67 of file ccthread.h.

2.3.2.7 #define THREAD_PRIV_MODS 0x0000000e

The 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 74 of file ccthread.h.

2.3.2.8 #define THREAD_PRIV_RT 0x00000001

Realtime thread. This flag does not have any stake in scheduling!

Definition at line 63 of file ccthread.h.

2.3.2.9 #define THREAD_PRIV_THRMG 0x00000020

The thread has access to thread management (creating, killing, renicing).

Definition at line 79 of file ccthread.h.

2.3.2.10 #define THREAD_PRIV_TLSF 0x00000010

The thread uses TLSF for DSA, or if not set, uses DLAlloc (needs DYNMEN!).

Definition at line 76 of file ccthread.h.

2.3.2.11 #define YIELD() __asm(".word 0x0440000d\n")

Not yet implemented, and also not making any sense with the current multithreading concept.

Definition at line 54 of file ccthread.h.

2.3.3 Function Documentation

2.3.3.1 thread_handler create_thread (void(*)(void) *thread_func*, uint32_t *thread_flags*, memory_t * *mem*, uint32_t *thread_mem*, sched_t *sched_params*)

Parameters:

- ***thread_func*** The function to run in the new thread slow.
- ***thread_flags*** Flags describing the thread behaviour (e.g. real time...).

thread_mem The initial memory for the thread (only sensible, if the thread needs not to extend its memory).

sched_params The scheduling parameters (a value from 0 to 3); the remarks on **renice** (p. 51) here also apply!

mem Memory to use for the thread. If set to NULL, system standard memory will be used.

Returns:

A handler for the new thread, **NO_THREAD** (p. 53) on error. For error information check errno: **EACCES** (p. 32) if the calling thread is not allowed to create new threads; **ENOSPC** (p. 41) if no more threads can be created.

Creates a new thread with the given flags and allocate the specified amount of memory for it. The function f is run when the thread is started.

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.

2.3.3.2 thread_handler get_current_thread_handler ()

Returns:

The handler of the current thread.

2.3.3.3 uint32_t get_priority (thread_handler *th*)

Parameters:

th The handler of the thread.

Returns:

The priority of the thread with the specified handler.

2.3.3.4 bool_t has_privilege (uint32_t *priv*)

Parameters:

priv The privilege to check.

2.3.3.5 int32_t join (thread_handler *thread*)

Parameters:

thread wait for this thread

Returns:

EINVAL (p. 36): thread does not refer to a joinable thread; **ESRCH** (p. 45): no thread for the given ID found.

2.3.3.6 uint32_t renice (thread_handler *th*, sched_t *new_params*)

This function changes the priority of the specified thread. As each priority must be allocated exactly once, in fact the priorities of the specified thread and the thread currently holding the specified priority are swapped. No checking is done, whether the other thread is currently allocated and/or running!

Parameters:

th The handler of the thread.

new_params The new scheduling parameters.

Returns:

EACCES (p. 32) If the calling thread may not use the thread manager; **ESRCH** (p. 45) if the specified thread does not exist.

2.3.3.7 uint32_t start_thread (thread_handler *handler*)

Parameters:

handler The handler of the thread to start. The handler must have been gained through a call to **create_thread** (p. 49).

Returns:

EACCES (p. 32) If the calling thread doesn't have the right to manage threads.

2.3.3.8 void swap_priorities (thread_handler *th1*, thread_handler *th2*)

Parameters:

th1 Thread 1, wanting the priority of thread 2.

th2 Thread 2, wanting the priority of thread 1.

2.4 ccthreadtypes.h File Reference

Global type definitions.

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

Data Structures

- struct **ccthread_mutex**

  The mutex definition.

- struct **ccthread_cond**

  Definition of a conditional variable.

Defines

- #define **_CCTHREADTYPES_H_** 1
- #define **NO_THREAD** ((**uint32_t**)-1)
- #define **PRIORITY_0** ((**uint8_t**)0)
- #define **PRIORITY_1** ((**uint8_t**)1)
- #define **PRIORITY_2** ((**uint8_t**)2)
- #define **PRIORITY_3** ((**uint8_t**)3)
- #define **MIN_PRIORITY** ((**uint8_t**)0)
- #define **MAX_PRIORITY** ((**uint8_t**)3)

Typedefs

- typedef **uint32_t thread_handler**

  The thread handler, i.e. an ID for a specific thread.

- typedef **uint32_t sched_t**

  The scheduling parameters.

- typedef **int32_t cc_spinlock_t**

  The spinlock.

- typedef **ccthread_mutex ccthread_mutex_t**

  The mutex definition.

- typedef **ccthread_cond ccthread_cond_t**

  Definition of a conditional variable.

2.4.1 Detailed Description

This file provides definitions of different thread types, like mutexes, spinlocks and conditionals.

Definition in file **ccthreadtypes.h**.

2.4.2 Define Documentation

2.4.2.1 #define _CCTHREADTYPES_H_ 1

Definition at line 31 of file ccthreadtypes.h.

2.4.2.2 #define MAX_PRIORITY ((uint8_t)3)

Maximum priority is equal to **PRIORITY_3** (p. 53)

Definition at line 53 of file ccthreadtypes.h.

2.4.2.3 #define MIN_PRIORITY ((uint8_t)0)

Minimum priority is equal to **PRIORITY_0** (p. 53)

Definition at line 51 of file ccthreadtypes.h.

2.4.2.4 #define NO_THREAD ((uint32_t)-1)

Thread handler ID with the meaning: No thread!

Definition at line 43 of file ccthreadtypes.h.

2.4.2.5 #define PRIORITY_0 ((uint8_t)0)

Definition at line 45 of file ccthreadtypes.h.

2.4.2.6 #define PRIORITY_1 ((uint8_t)1)

Definition at line 46 of file ccthreadtypes.h.

2.4.2.7 #define PRIORITY_2 ((uint8_t)2)

Definition at line 47 of file ccthreadtypes.h.

2.4.2.8 #define PRIORITY_3 ((uint8_t)3)

Definition at line 48 of file ccthreadtypes.h.

2.4.3 Typedef Documentation

2.4.3.1 typedef int32_t cc_spinlock_t

The type of a spinlock: it's just a memory word.

Definition at line 80 of file ccthreadtypes.h.

2.4.3.2 typedef struct ccthread_cond ccthread_cond_t

2.4.3.3 typedef struct ccthread_mutex ccthread_mutex_t

A high-level implementation of a mutex variable

2.4.3.4 typedef uint32_t sched_t

This struct contains one set of scheduling parameters.

Definition at line 73 of file ccthreadtypes.h.

2.4.3.5 typedef uint32_t thread_handler

How to handle a thread: you just need an ID; this value is used as an offset into the TCB array.

Definition at line 62 of file ccthreadtypes.h.

2.5 csfr.h File Reference

Core Special Function Register definitions.

Defines

- #define **_CSFR_H_** 1
- #define **CSFR_CORE** "0x7600"
- #define **CSFR_SLOT** "0x7604"
- #define **CSFR_TCB** "0x7608"
- #define **CSFR_STOP_IMM** "0x760c"
- #define **CSFR_MY_CORE** "0x7610"
- #define **CSFR_MY_SLOT** "0x7614"
- #define **CSFR_MY_TCB** "0x7618"
- #define **CSFR_PRIOS** "0x7620"

2.5.1 Detailed Description

If you don't know what these are about, don't use them! All thread handling is already done by the MERASA system level software.

Definition in file **csfr.h**.

2.5.2 Define Documentation

2.5.2.1 #define _CSFR_H_ 1

Definition at line 35 of file csfr.h.

2.5.2.2 #define CSFR_CORE "0x7600"

The core where to start a thread.

Definition at line 46 of file csfr.h.

2.5.2.3 #define CSFR_MY_CORE "0x7610"

Read the core of my thread.

Definition at line 54 of file csfr.h.

2.5.2.4 #define CSFR_MY_SLOT "0x7614"

Read the slot of my thread.

Definition at line 56 of file csfr.h.

2.5.2.5 #define CSFR_MY_TCB "0x7618"

Read the TCB of my thread.

Definition at line 58 of file csfr.h.

2.5.2.6 #define CSFR_PRIOS "0x7620"

Read the priorities of the threads (32-bit field = 4*8 bit).

Definition at line 60 of file csfr.h.

2.5.2.7 #define CSFR_SLOT "0x7604"

The thread-slot where to start a thread.

Definition at line 48 of file csfr.h.

2.5.2.8 #define CSFR_STOP_IMM "0x760c"

Stop the active thread.

Definition at line 52 of file csfr.h.

2.5.2.9 #define CSFR_TCB "0x7608"

The thread control block (TCB) for a thread.

Definition at line 50 of file csfr.h.

2.6 driver.h File Reference

Macros for writing MERASA device drivers.

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

Data Structures

- struct **driver_interface**

  This struct holds the functions a driver must implement, i.e. open, close, read, write and ioctl.

- struct **driver**

  This struct describes a device driver.

Defines

- #define **DRIVER_H_** 1
- #define **DRIVER**(n, v, i, c, o)

  Macro that describes a device driver.

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 **driver_interface drvif_t**

  This struct holds the functions a driver must implement, i.e. open, close, read, write and ioctl.

- typedef **driver driver_t**

  This struct describes a device driver.

2.6.1 Detailed Description

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

Definition in file **driver.h**.

2.6.2 Define Documentation

2.6.2.1 #define DRIVER(n, v, i, c, o)

Value:

```
static struct driver __this_driver __attribute__((section(".cc.driver"), unused)) = { \
    .name = ##n, \
    .version = ##v, \
    .init = ##i, \
    .cleanup = ##c, \
    .ops = ##o, \
    .pi = NULL, \
    .owner = NO_THREAD, \
    .sp_next = NULL }
```

Parameters:

- ***n*** Name of the driver.
- ***v*** Version of this driver.
- ***i*** Initialisation function.
- ***c*** Cleanup function.
- ***o*** Operations struct (**drvif_t** (p. 58)).

Definition at line 122 of file driver.h.

2.6.2.2 #define DRIVER_H_ 1

Definition at line 38 of file driver.h.

2.6.3 Typedef Documentation

2.6.3.1 typedef struct driver driver_t

Do not use it directly, instead publish your driver using the **DRIVER** (p. 57) macro!

2.6.3.2 typedef int32_t(*) drv_cleanup_fn_t(void)

Definition at line 62 of file driver.h.

2.6.3.3 typedef int32_t(∗) drv_close_fn(void)

Definition at line 65 of file driver.h.

2.6.3.4 typedef int32_t(∗) drv_init_fn_t(const void ∗)

A driver must provide a initialisation function. This is the signatur it must implement. The passed pointer points to the device's start address in memory. Currently, this function must not fail! The initialisation function of a driver

Definition at line 61 of file driver.h.

2.6.3.5 typedef size_t(∗) drv_ioctl_fn(uint32_t, va_list)

Definition at line 68 of file driver.h.

2.6.3.6 typedef int32_t(∗) drv_open_fn(void)

Definition at line 64 of file driver.h.

2.6.3.7 typedef size_t(∗) drv_read_fn(void ∗, size_t)

Definition at line 66 of file driver.h.

2.6.3.8 typedef size_t(∗) drv_write_fn(const void ∗, size_t)

Definition at line 67 of file driver.h.

2.6.3.9 typedef struct driver_interface drvif_t

2.7 drivermanager.h File Reference

Management of device drivers.

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

Defines

- #define **DRIVERMANAGER_H_** 1
- #define **MAX_DRV_PERFORMANCE** (0xFF)

Typedefs

- typedef **int32_t drv_handler**

  The handler for one driver.

Functions

- **uint32_t install_driver** (const void *elf, size_t len, const char *dev_id, const void *base_address)

  Installs a driver for a device.

- **uint32_t remove_driver** (const char *dev_id)

  Removes a driver from the system.

- **drv_handler cc_open** (const char ***driver**, **uint32_t** flags)

  Opens a device.

- **drv_handler cc_bopen** (const char ***driver**, **uint32_t** flags)

  Opens a device; if the device is busy, wait until it gets free again.

- **uint32_t cc_close** (**drv_handler** handle)

  Closes a currently used device.

- size_t **cc_read** (**drv_handler** handle, void *buf, size_t count)

  Reads from a device.

- size_t **cc_write** (**drv_handler** handle, const void *buf, size_t count)

  Writes to a device.

- **int32_t cc_ioctl** (**drv_handler** handle, **uint32_t** request,...)

  Performs a control operation.

- **thread_handler get_drv_owner** (const char *dev_id)

  Reads the current owner of the specified device.

2.7.1 Detailed Description

This file provides the methods to manage the driver access. Hence it is possible to install and remove a driver, as well as perform some open / close / read / write / ioctl operations.

Definition in file **drivermanager.h**.

2.7.2 Define Documentation

2.7.2.1 #define DRIVERMANAGER_H_ 1

Definition at line 36 of file drivermanager.h.

2.7.2.2 #define MAX_DRV_PERFORMANCE (0xFF)

Definition at line 50 of file drivermanager.h.

2.7.3 Typedef Documentation

2.7.3.1 typedef int32_t drv_handler

Definition at line 58 of file drivermanager.h.

2.7.4 Function Documentation

2.7.4.1 drv_handler cc_bopen (const char * *driver*, uint32_t *flags*)

Parameters:

driver The driver device/driver identifier.

flags Some flags associated with the driver.

Returns:

A handler for the device; -1 if the device could not be opened. Check errno via **get_-errno** (p. 46) for details: **EBUSY** (p. 33) if the device is already opened; **ENODEV** (p. 40) if the specified device does not exist.

2.7.4.2 uint32_t cc_close (drv_handler *handle*)

The calling thread must currently own the device, and it must be the last one openend and not already closed (stack/LIFO architecture).

Parameters:

handle The device handler.

Returns:

EACCES (p. 32) the thread does not own the device; EPERM if it is not on top of the thread's device stack.

2.7.4.3 int32_t cc_ioctl (drv_handler *handle*, uint32_t *request*, ...)

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

Parameters:

handle The handler of the device.

request The request code.

Returns:

EACCES (p. 32) if the calling thread does not own the device Further errnos might be set by the driver, see driver documentation therefor.

2.7.4.4 drv_handler cc_open (const char * *driver*, uint32_t *flags*)

Parameters:

driver The device/driver identifier.

flags Some flags associated with the driver.

Returns:

A handler for the device; -1 if the device could not be opened. Check errno via **get_errno** (p. 46) for details: **EBUSY** (p. 33) if the device is already opened; **ENODEV** (p. 40) if the specified device does not exist.

2.7.4.5 size_t cc_read (drv_handler *handle*, void * *buf*, size_t *count*)

Parameters:

handle The handler of the device.

buf The buffer to write (must be writeable by the calling thread).

count The maximum number of bytes to write.

Returns:

The number of bytes written. If the operations fails, -1 is returned. For more information check errno (**get_errno** (p. 46)): **EACCES** (p. 32) if the calling thread does not own the device; **EPERM** (p. 43) if buf is not writeable by the calling thread. Further errnos might be set by the driver, see driver documentation therefor.

2.7.4.6 size_t cc_write (drv_handler *handle*, const void * *buf*, size_t *count*)

Parameters:

handle The handler of the device.

buf The data to write.

count The number of bytes to write.

Returns:

The number of bytes written. If the operation fails, -1 is returned. For more information check erro (**get_errno** (p. 46)): **EACCES** (p. 32) if the calling thread does not own the device. Further errnos might be set by the driver, see driver documentation therefor.

2.7.4.7 thread_handler get_drv_owner (const char * *dev_id*)

Parameters:

dev_id The device/driver identifier.

Returns:

The handler for the driver's owner, **NO_THREAD** (p. 53) if the driver is currently free.

2.7.4.8 uint32_t install_driver (const void * *elf*, size_t *len*, const char * *dev_id*, const void * *base_address*)

Parameters:

elf The ELF image.

len The length of the ELF image.

dev_id An identifier of the device. Over this string the driver can be accessed from the user space.

base_address The address that will be passed to the driver's initialisation routine.

Returns:

EPERM (p. 43) if the thread doesn't have the privilege to use the driver manager (**THREAD_PRIV_DRVMG** (p. 48)); **EEXIST** (p. 35) if a driver with the same ID is already loaded.

2.7.4.9 uint32_t remove_driver (const char * *dev_id*)

Parameters:

dev_id The device ID under which the driver is published.

Returns:

EPERM if the thread doesn't have the privilege to use the driver manager (**THREAD_-PRIV_DRVMG** (p. 48)); **EBUSY** (p. 33) if the driver is currently used; **ENODEV** (p. 40) if the given device dev_id does not exist.

2.8 log.h File Reference

Logging facilities.

```
#include "../config.h"
#include <sync.h>
```

Defines

- #define **LOG_H_** 1
- #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 **LOCK_LOG**() ccthread_mutex_lock(&**log_mutex**)
- #define **UNLOCK_LOG**() ccthread_mutex_unlock(&**log_mutex**)
- #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

- **ccthread_mutex_t log_mutex**

2.8.1 Detailed Description

This file provides easy logging facilities. The output is written to log-file or STDOUT. There are different loglevel-stages: DEBUG, INFO, WARN, ERR, FATAL, NONE.

Definition in file **log.h**.

2.8.2 Define Documentation

2.8.2.1 #define LOCK_LOG() ccthread_mutex_lock(&log_mutex)

Definition at line 76 of file log.h.

2.8.2.2 #define log_debug(msg)

Value:

```
LOCK_LOG(); \
```

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

Definition at line 94 of file log.h.

2.8.2.3 #define log_debug_s(msg, args...)

Value:

```
{ \
  LOCK_LOG(); \
  char buffer[81]; \
  sprintf(buffer, msg, args); \
  printf("[5,%u] %s:%d %s\n", get_thread(),__FILE__,  __LINE__, buffer); \
  UNLOCK_LOG(); \
}
```

Definition at line 81 of file log.h.

2.8.2.4 #define log_err(msg)

Value:

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

Definition at line 166 of file log.h.

2.8.2.5 #define log_err_s(msg, args...)

Value:

```
{ \
  LOCK_LOG(); \
  char buffer[81]; \
  sprintf(buffer, msg, args); \
  printf("[2,%u] %s:%d %s\n", get_thread(),__FILE__,  __LINE__, buffer); \
  UNLOCK_LOG(); \
}
```

Definition at line 153 of file log.h.

2.8.2.6 #define log_fatal(msg)

Value:

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

Definition at line 190 of file log.h.

2.8.2.7 #define log_fatal_s(msg, args...)

Value:

```
{ \
  LOCK_LOG(); \
  char buffer[81]; \
  sprintf(buffer, msg, args); \
  printf("[1,%u] %s:%d %s\n", get_thread(),__FILE__,  __LINE__, buffer); \
  UNLOCK_LOG(); \
}
```

Definition at line 177 of file log.h.

2.8.2.8 #define LOG_H_ 1

Definition at line 31 of file log.h.

2.8.2.9 #define log_info(msg)

Value:

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

Definition at line 118 of file log.h.

2.8.2.10 #define log_info_s(msg, args...)

Value:

```
{ \
  LOCK_LOG(); \
  char buffer[81]; \
  sprintf(buffer, msg, args); \
  printf("[4,%u] %s:%d %s\n", get_thread(),__FILE__,  __LINE__, buffer); \
  UNLOCK_LOG(); \
}
```

Definition at line 105 of file log.h.

2.8.2.11 #define log_warn(msg)

Value:

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

Definition at line 142 of file log.h.

2.8.2.12 #define log_warn_s(msg, args...)

Value:

```
{ \
  LOCK_LOG(); \
  char buffer[81]; \
  sprintf(buffer, msg, args); \
  printf("[3,%u] %s:%d %s\n", get_thread(),__FILE__,  __LINE__, buffer); \
  UNLOCK_LOG(); \
}
```

Definition at line 129 of file log.h.

2.8.2.13 #define LOGLEVEL_DEBUG 5

Definition at line 43 of file log.h.

2.8.2.14 #define LOGLEVEL_ERR 2

Definition at line 46 of file log.h.

2.8.2.15 #define LOGLEVEL_FATAL 1

Definition at line 47 of file log.h.

2.8.2.16 #define LOGLEVEL_INFO 4

Definition at line 44 of file log.h.

2.8.2.17 #define LOGLEVEL_NONE 0

Definition at line 48 of file log.h.

2.8.2.18 #define LOGLEVEL_WARN 3

Definition at line 45 of file log.h.

2.8.2.19 #define UNLOCK_LOG() ccthread_mutex_unlock(&log_mutex)

Definition at line 77 of file log.h.

2.8.3 Function Documentation

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

2.8.4 Variable Documentation

2.8.4.1 ccthread_mutex_t log_mutex

2.9 memdev.h File Reference

Definition of memory adresses for peripheral device access.

Defines

- #define **MEMDEV_HPP_** 1
- #define **SERVICE_MASK** (0xffffff00)
- #define **PER_VIO** (0xe0010000)
- #define **VO_ADDRESS** 0xe0010000
- #define **VIO_PREFIX** "%#"
- #define **P32**(a) (*((**uint32_t** volatile *) a))
- #define **P64**(a) (*((**uint64_t** volatile *) a))

2.9.1 Detailed Description

This file is used for driver development, but usually not within the kernel of the MERASA system level software. User programs also shouldn't care about it!

Definition in file **memdev.h**.

2.9.2 Define Documentation

2.9.2.1 #define MEMDEV_HPP_ 1

Definition at line 12 of file memdev.h.

2.9.2.2 #define P32(a) (*((uint32_t volatile *) a))

Fast access to 32-bit value

Definition at line 25 of file memdev.h.

2.9.2.3 #define P64(a) (*((uint64_t volatile *) a))

Fast access to 64-bit value

Definition at line 27 of file memdev.h.

2.9.2.4 #define PER_VIO (0xe0010000)

Definition at line 15 of file memdev.h.

2.9.2.5 #define SERVICE_MASK (0xffffff00)

Definition at line 14 of file memdev.h.

2.9.2.6 #define VIO_PREFIX "%#"

Allows easy filtering of VIO output in console mode.

Definition at line 22 of file memdev.h.

2.9.2.7 #define VO_ADDRESS 0xe0010000

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

Definition at line 20 of file memdev.h.

2.10 memory-desc.h File Reference

The description of the memory configuration.

`#include <types.h>`

`#include <ccthreadtypes.h>`

Data Structures

- struct **mem_cfg_data**

  Structure to describe the configuration of memory.

Defines

- #define **MEMORY_DESC_H_** 1
- #define **MEMORY_CFG**(b, l, ac, c, f)

  Macro to set up a memory configuration.

Typedefs

- typedef **mem_cfg_data memory_t**

  Structure to describe the configuration of memory.

Variables

- **memory_t node_mem_config** []

- const size_t **mem_config_len**

2.10.1 Detailed Description

This file contains definitions and methods for the configuration of memory, e.g. beginning, end, length, the cost for the use etc.

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

2.10.2 Define Documentation

2.10.2.1 #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 \
}
```

Parameters:

b The first byte of the memory area.

l The length of the memory area.

ac The average number of cycles to accesss one word of this memory.

c The cost for use of this memory.

f Some flags for further use.

Definition at line 94 of file memory-desc.h.

2.10.2.2 #define MEMORY_DESC_H_ 1

Definition at line 31 of file memory-desc.h.

2.10.3 Typedef Documentation

2.10.3.1 typedef struct mem_cfg_data memory_t

2.10.4 Variable Documentation

2.10.4.1 const size_t mem_config_len

For correct access to **node_mem_config** (p. 70), you have to set this constant as sizeof(node_mem_config)/sizeof(memory_t).

2.10.4.2 memory_t node_mem_config[ ]

You need to define this constant in the OS for one specific node configuration. Use the **MEMORY_CFG** (p. 69) macro for filling this array!

2.11 memory.h File Reference

An interface for dynamic memory management.

`#include <ccthreadtypes.h>`

`#include <types.h>`

`#include <memory-desc.h>`

Defines

- #define **MEMORY_H_** 1

Functions

- void * **malloc** (size_t size)

  Allocates memory for the current thread.

- void **free** (void *ptr)

  Frees a previously allocated block of the current thread.

- void * **tcmemcpy** (void *dest, const void *src, size_t n)

  Performs an effective copy of (non-overlapping!) memory .

- **bool_t has_write_permission** (void *mem)

  Checks if a thread is allowed to write to the specified address.

2.11.1 Detailed Description

This offers possibilities to allocate or free memory blocks and to copy non-overlapping memory blocks to another address.

Definition in file **memory.h**.

2.11.2 Define Documentation

2.11.2.1 #define MEMORY_H_ 1

Definition at line 32 of file memory.h.

2.11.3 Function Documentation

2.11.3.1 void free (void * *ptr*)

Parameters:

ptr The memory block to free.

2.11.3.2 bool_t has_write_permission (void * *mem*)

Parameters:

mem The adress where the write check should be performed.

2.11.3.3 void* malloc (size_t *size*)

Parameters:

size Amount of memory to allocate.

2.11.3.4 void* tcmemcpy (void * *dest*, const void * *src*, size_t *n*)

Parameters:

dest The destination for the copy

src The source of the the copy

n The size of the memory to copy

2.12 peripheral-desc.h File Reference

Description of peripheral configuration.

`#include <types.h>`

Data Structures

- struct **per_cfg_data**

  Struct for peripheral configuration.

Defines

- #define **PERIPHERAL_DESC_H_** 1

Typedefs

- typedef **per_cfg_data peripheral_t**
  Struct for peripheral configuration.

2.12.1 Detailed Description

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

2.12.2 Define Documentation

2.12.2.1 #define PERIPHERAL_DESC_H_ 1

Definition at line 29 of file peripheral-desc.h.

2.12.3 Typedef Documentation

2.12.3.1 typedef struct per_cfg_data peripheral_t

2.13 regcarcore.h File Reference

Definition of the CarCore register file and related data.

Defines

- #define **_REGCARCORE_H_** 1
- #define **TCB_COUNT** 128
- #define **TCB_BASE** 0x90000000
- #define **TCB_BEGIN** 0x90000100
- #define **TCB_END** 0x90001000
- #define **CSA_SIZE** 0x2000
- #define **SEGMENT_MASK** 0xf0000000
- #define **STM_TIM0** (*((**uint32_t** volatile *) 0xF0000210))

2.13.1 Detailed Description

Definition in file **regcarcore.h**.

2.13.2 Define Documentation

2.13.2.1 #define _REGCARCORE_H_ 1

Definition at line 28 of file regcarcore.h.

2.13.2.2 #define CSA_SIZE 0x2000

Definition at line 55 of file regcarcore.h.

2.13.2.3 #define SEGMENT_MASK 0xf0000000

Definition at line 59 of file regcarcore.h.

2.13.2.4 #define STM_TIM0 (*((uint32_t volatile *) 0xF0000210))

Definition at line 63 of file regcarcore.h.

2.13.2.5 #define TCB_BASE 0x90000000

Base address of Thread Control Blocks.

Definition at line 44 of file regcarcore.h.

2.13.2.6 #define TCB_BEGIN 0x90000100

Begin address of Thread Control Blocks.

Definition at line 46 of file regcarcore.h.

2.13.2.7 #define TCB_COUNT 128

Maximum number of threads.

Definition at line 40 of file regcarcore.h.

2.13.2.8 #define TCB_END 0x90001000

End address of Thread Control Blocks (first byte after).

Definition at line 48 of file regcarcore.h.

2.14 sync.h File Reference

Synchronisation mechanisms similar to the POSIX interface pthread.h.

`#include <ccthread.h>`

`#include <ccthreadtypes.h>`

Defines

- #define **SYNC_H_** 1

- #define **NO_OWNER** ((**thread_handler**)-1)

Functions

- **int32_t ccthread_mutex_init** (**ccthread_mutex_t** *mutex)

  Initialize a mutex variable.

- **int32_t ccthread_mutex_destroy** (**ccthread_mutex_t** *mutex)

  Destroy a mutex variable.

- **int32_t ccthread_mutex_lock** (**ccthread_mutex_t** *mutex)

  Lock a mutex variable.

- **int32_t ccthread_mutex_unlock** (**ccthread_mutex_t** *mutex)

  Unlock a mutex variable.

- **int32_t ccthread_mutex_trylock** (**ccthread_mutex_t** *mutex)

  Try to lock a mutex variable.

- **int32_t ccthread_cond_init** (**ccthread_cond_t** *cond)

  Initialise a conditional variable.

- **int32_t ccthread_cond_destroy** (**ccthread_cond_t** *cond)

  Destroy a conditional variable.

- **int32_t ccthread_cond_wait** (**ccthread_cond_t** *cond, **ccthread_mutex_t** *mutex)

  Sets a process into the waiting queue for this conditional variable.

- **int32_t ccthread_cond_signal** (**ccthread_cond_t** *cond)

  Notifies one waiting process to let him continue execution.

- **int32_t ccthread_cond_broadcast** (**ccthread_cond_t** *cond)

  Notifies all waiting processes.

2.14.1 Detailed Description

This file contains mechanisms for synchronisation, i.e. mutex und conditional variables. The methods like init(), lock(), trylock(), unlock() etc. are mostly known from the POSIX interface.

Definition in file **sync.h**.

2.14.2 Define Documentation

2.14.2.1 #define NO_OWNER ((thread_handler)-1)

Definition at line 47 of file sync.h.

2.14.2.2 #define SYNC_H_ 1

Definition at line 35 of file sync.h.

2.14.3 Function Documentation

2.14.3.1 int32_t ccthread_cond_broadcast (ccthread_cond_t * *cond*)

Parameters:

cond the conditional

2.14.3.2 int32_t ccthread_cond_destroy (ccthread_cond_t * *cond*)

Parameters:

cond The conditional variable to destroy.

Returns:

EBUSY Some threads are currently waiting on cond.

2.14.3.3 int32_t ccthread_cond_init (ccthread_cond_t * *cond*)

Parameters:

cond The conditional variable to initialise.

2.14.3.4 int32_t ccthread_cond_signal (ccthread_cond_t * *cond*)

Parameters:

cond the conditional

2.14.3.5 int32_t ccthread_cond_wait (ccthread_cond_t * *cond*, ccthread_-mutex_t * *mutex*)

Parameters:

cond The conditional variable.

mutex The associated mutex variable.

2.14.3.6 int32_t ccthread_mutex_destroy (ccthread_mutex_t * *mutex*)

Parameters:

mutex The mutex variable to destroy.

Returns:

EBUSY: the mutex is currently locked. For further use, it must be initialised again using **ccthread_mutex_init** (p. 76), otherwise behaviour is undefined! Trying to destroy a locked mutex will cause an error.

2.14.3.7 int32_t ccthread_mutex_init (ccthread_mutex_t * *mutex*)

Parameters:

mutex The mutex variable to initialize. Always use this function before you use a mutex variable!

2.14.3.8 int32_t ccthread_mutex_lock (ccthread_mutex_t * *mutex*)

Parameters:

mutex The mutex variable to lock.

Returns:

EINVAL: the mutex has not been properly initialised. If the mutex is already locked, the calling thread is suspended until it gets free again.

2.14.3.9 int32_t ccthread_mutex_trylock (ccthread_mutex_t * *mutex*)

Parameters:

mutex The mutex variable to lock. This function returns immediately. If the locking was successful, 0 is returned. If it fails, a non-zero value is returned.

Returns:

EINVAL: the mutex has not been properly initialised; EBUSY: the mutex could not be acquired because it was currently locked.

2.14.3.10 int32_t ccthread_mutex_unlock (ccthread_mutex_t * *mutex*)

Parameters:

mutex The mutex variable to unlock. The mutex variable is unlocked and all waiting threads (if there are any) are woken up.

Returns:

EINVAL: the mutex has not been properly initialised; EPERM: the calling thread does not own the mutex.

2.15 sysmonitor.h File Reference

Monitoring of core system parameters.

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

Data Structures

- struct **memorystatistics**

  Memorystatistics describing the state of global memory.

- struct **thread_memorystatistics**

  Thread_memorystatistics describing the state of a thread's memory.

Defines

- #define **SYSMONITOR_H_** 1

Typedefs

- typedef **memorystatistics** * **memstatptr**
- typedef **thread_memorystatistics** * **tmemstatptr**

2.15.1 Detailed Description

Definition in file **sysmonitor.h**.

2.15.2 Define Documentation

2.15.2.1 #define SYSMONITOR_H_ 1

Definition at line 30 of file sysmonitor.h.

2.15.3 Typedef Documentation

2.15.3.1 typedef struct memorystatistics* memstatptr

A shortcut for memorystatistics.

Definition at line 58 of file sysmonitor.h.

2.15.3.2 typedef struct thread_memorystatistics* tmemstatptr

A shortcut for **thread_memorystatistics** (p. 25).

Definition at line 73 of file sysmonitor.h.

2.16 tcintrinsics.h File Reference

Some intrinsic functions.

Defines

- #define **_TCINTRINSICS_H** 1
- #define **__max**(a, b) ({ int m; asm volatile ("max %0, %1, %2" : "=d"(m) : "d"(a), "d"(b)); m; })
- #define **__min**(a, b) ({ int m; asm volatile ("min %0, %1, %2" : "=d"(m) : "d"(a), "d"(b)); m; })
- #define **__max_u**(a, b) ({ int m; asm volatile ("max.u %0, %1, %2" : "=d"(m) : "d"(a), "d"(b)); m; })
- #define **__min_u**(a, b) ({ int m; asm volatile ("min.u %0, %1, %2" : "=d"(m) : "d"(a), "d"(b)); m; })
- #define **__MTCR**(cr, val) __asm("mtcr "cr", %0" : : "d"(val))
- #define **__MFCR**(cr) ({ int val; asm volatile ("mfcr %0, "cr : "=d"(val) :); val; })

2.16.1 Detailed Description

For the use of symbolic CR constants, use the here defined **__MTCR** (p. 79) and **__MFCR** (p. 78).

Definition in file **tcintrinsics.h**.

2.16.2 Define Documentation

2.16.2.1 #define __max(a, b) ({ int m; asm volatile ("max %0, %1, %2" : "=d"(m) : "d"(a), "d"(b)); m; })

Definition at line 58 of file tcintrinsics.h.

2.16.2.2 #define __max_u(a, b) ({ int m; asm volatile ("max.u %0, %1, %2" : "=d"(m) : "d"(a), "d"(b)); m; })

Definition at line 62 of file tcintrinsics.h.

2.16.2.3 #define __MFCR(cr) ({ int val; asm volatile ("mfcr %0, "cr : "=d"(val) :); val; })

Definition at line 70 of file tcintrinsics.h.

2.16.2.4 #define __min(a, b) ({ int m; asm volatile ("min %0, %1, %2" : "=d"(m) : "d"(a), "d"(b)); m; })

Definition at line 60 of file tcintrinsics.h.

2.16.2.5 #define __min_u(a, b) ({ int m; asm volatile ("min.u %0, %1, %2" : "=d"(m) : "d"(a), "d"(b)); m; })

Definition at line 64 of file tcintrinsics.h.

2.16.2.6 #define __MTCR(cr, val) __asm("mtcr "cr", %0" : : "d"(val))

Definition at line 68 of file tcintrinsics.h.

2.16.2.7 #define _TCINTRINSICS_H 1

Definition at line 33 of file tcintrinsics.h.

2.17 types.h File Reference

Hardware dependent and other types.

```
#include <stddef.h>
```

Defines

- #define **TYPES_H_** 1
- #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 **uint32_t** **version_t**

Enumerations

- enum **bool_t** { **FALSE** = 0, **TRUE** = 1 }

2.17.1 Detailed Description

Definition in file **types.h**.

2.17.2 Define Documentation

2.17.2.1 #define false FALSE

Definition at line 71 of file types.h.

2.17.2.2 #define true TRUE

Definition at line 72 of file types.h.

2.17.2.3 #define TYPES_H_ 1

Definition at line 30 of file types.h.

2.17.3 Typedef Documentation

2.17.3.1 typedef char* address

Definition at line 75 of file types.h.

2.17.3.2 typedef signed short int int16_t

16-bit signed short integer.

Definition at line 50 of file types.h.

2.17.3.3 typedef signed int int32_t

32-bit signed integer.

Definition at line 54 of file types.h.

2.17.3.4 typedef signed long long int64_t

64-bit signed long integer.

Definition at line 58 of file types.h.

2.17.3.5 typedef signed char int8_t

8-bit signed char.

Definition at line 46 of file types.h.

2.17.3.6 typedef unsigned short int uint16_t

16-bit unsigned short integer.

Definition at line 48 of file types.h.

2.17.3.7 typedef unsigned int uint32_t

32-bit unsigned integer.

Definition at line 52 of file types.h.

2.17.3.8 typedef unsigned long long uint64_t

64-bit unsigned long integer.

Definition at line 56 of file types.h.

2.17.3.9 typedef unsigned char uint8_t

8-bit unsigned char.

Definition at line 44 of file types.h.

2.17.3.10 typedef uint32_t version_t

Definition at line 79 of file types.h.

2.17.4 Enumeration Type Documentation

2.17.4.1 enum bool_t

Boolean type.

Enumerator:

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

Definition at line 70 of file types.h.