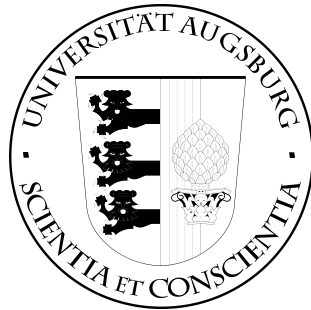


UNIVERSITÄT AUGSBURG

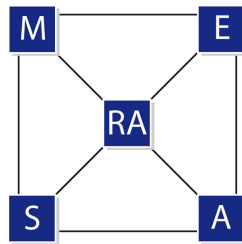


**Refined System-Level Software
for a Single-Core MERASA Processor**

Florian Kluge, Julian Wolf

Report 2008-15

Oktober 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 —

Contents

1	Introduction	4
2	Requirements	5
2.1	Functional Requirements	5
2.2	Requirements for the user interface	7
3	Architectural overview	7
4	Application Programming Interface	8
4.1	Thread Management	9
4.2	Dynamic Memory Management	9
4.3	Resource Management	9
4.4	Common header files	10
4.5	POSIX interface	10
5	Implementation	11
5.1	Thread Management	11
5.2	Dynamic Memory Management	12
5.3	Resource Management	13
6	Conclusion	14
A	File description	15
A.1	ccerrno.h File Reference	15
A.2	driver.h File Reference	34
A.3	drivermanager.h File Reference	37
A.4	fcntl.h File Reference	42
A.5	log.h File Reference	43
A.6	memdev.h File Reference	49
A.7	memory-desc.h File Reference	52
A.8	memory.h File Reference	54
A.9	merasa-ssw.h File Reference	56
A.10	pthread.h File Reference	57
A.11	stropts.h File Reference	67
A.12	types.h File Reference	68
A.13	unistd.h File Reference	71

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 / CarCore 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 2 developed for a single multi-threaded core processor running on the CarCore / MERASA SystemC Simulator version 2.

1 Introduction

The main objective of the MERASA¹ 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 [13] which is binary compatible to the Infineon TriCore.

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

¹Multi-Core Execution of Hard Real-Time Applications Supporting Analysability, a STREP project within the Seventh Framework Programme of the European Union

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.

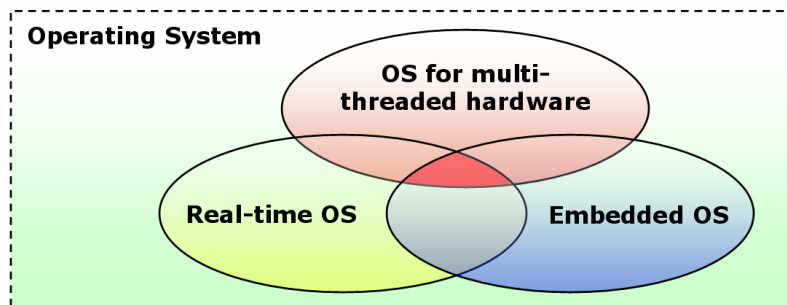


Figure 1: Combination of requirements for different OS types

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 characteristic 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 [9]) 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 [10] 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 reusability of functions to provide a standard platform for automotive systems.

However, the current version of the standard targets at an implementation for single-threaded processors. As SMT processors require additional functionalities for priority-based scheduling and resource management, an adaption can only be carried out with a high effort, resulting in a reduced performance. Similar problems will also arise for multi-core processors. Hence, the AUTOSAR standard should be extended, before an implementation in the field of SMT and multi-core processors is suggestive.

To facilitate the development of applications, it may instead be useful to gear functions towards the popular POSIX [11] 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 [12]). It will be easy to port applications developed for other systems using the POSIX interface as well (see sect. 4.5).

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

3 Architectural overview

The design of the MERASA system-level software joins several well-known OS techniques. The basic kernel comprises the most important management func-

functionalities following the microkernel principle. Additional functions may run outside this kernel as separate components.

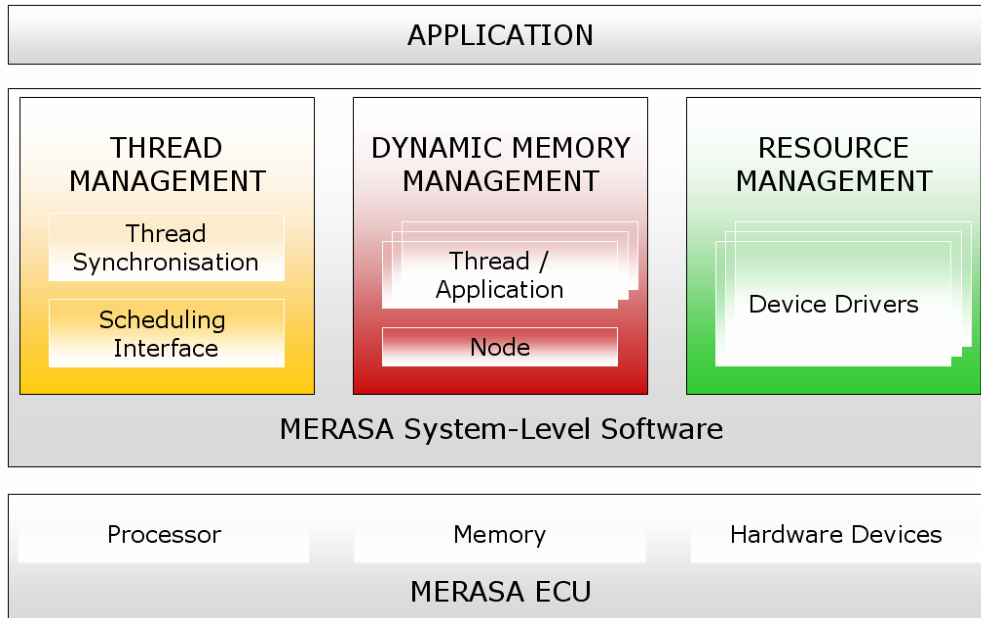


Figure 2: Architecture of the MERASA system-level software

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/`). 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 implemented as a subset of POSIX functions:

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

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

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

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

A special resource currently included in the MERASA system-level software is the *Virtual Output*. It implements functions very similar to `stdio.h` with

different format modifiers for various data types. In the MERASA simulator the output is written to `STDOUT`, prefaced with some special characters (`%#`). This enables an easy way to debug applications but will influence the timing behaviour of the application. For more information see the description of `log.h` in section 4.4.

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.

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. Also, this header provides *fast-logging facilities* to output just few bytes of data. These functions come with less and predictable timing overhead.

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

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.

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

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

4.5 POSIX interface

The MERASA system-level software does not yet implement the full POSIX interface. However, an essential subset of functions for thread and resource management is already provided.

The following list comprises the header files containing POSIX functions:

- `fcntl.h`
- `pthread.h`

- stropts.h
- unistd.h

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.

The *basic system-level software for a single-core MERASA processor*, as presented in [4], was based on an SMT processor with four hardware thread slots and fixed priority scheduling. In the meantime, the processor core was developed further and a hardware-based real-time scheduler was introduced [7, 8]. This scheduler is based on a periodic timing-driven scheduling policy, which allows to run three different types of threads:

Hard Real-Time Threads At each time there can run at most one hard real-time thread within the four hardware thread slots. This thread gets highest-priority access to all hardware resources for a predefined number of cycles (*cycle quantum (CQ)*) within each scheduling period [7].

Soft Real-Time Threads Scheduling of these threads is based on a *periodic instruction quantum (PIQ)*. As some instructions take more clock cycles to execute than others, the PIQ might be missed in some periods. However, the scheduler ensures that the missed instructions are made up in the following periods. Thus, in the long run there are also timing guarantees for such threads, but particular deadlines might be missed [8].

Non-Real-Time Threads The remaining processing time is filled with instructions from non-real-time threads. These are scheduled following a weighted round-robin scheme. No timing guarantees are given.

The final MERASA processor will permit at most one hard real-time thread per core. This way WCET analysability will be preserved.

Synchronization

As a second important task, synchronization 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 follows the POSIX [11] interface.

While in the previous version of the MERASA system-level software we provided mechanisms to prevent priority inversion, this is no longer necessary. The hardware-based scheduling technique divides the execution time into periods to guarantee a certain amount of time to the different hard and soft real-time threads [8]. This makes an occur of priority inversions impossible and therefore the mechanisms of priority inheritance are no longer needed.

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 exclusive 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.

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 [5] (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 [6], is a general purpose dynamic memory allocator specifically

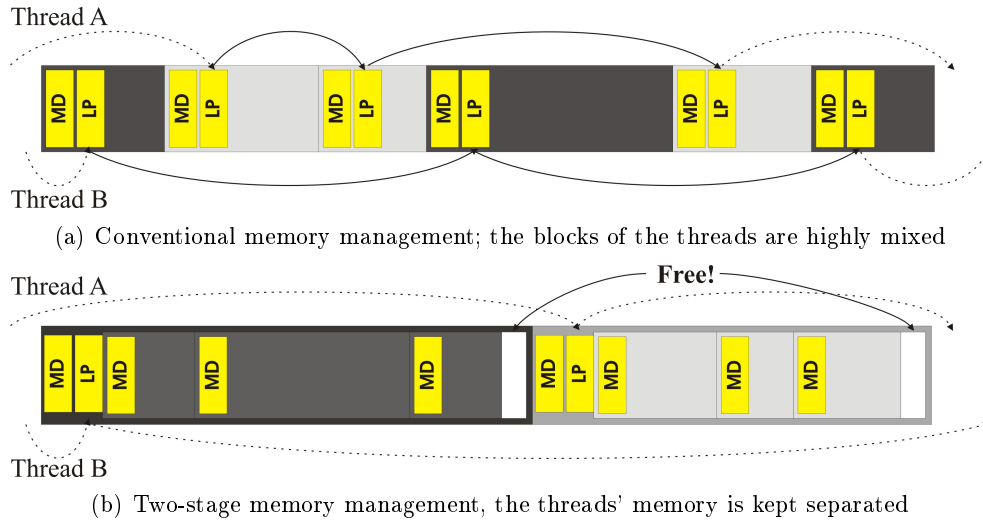


Figure 3: Example layout of used memory with two threads; MD: Management data of the memory allocator, LP: List Pointers to keep track of thread's memory

designed to meet real-time requirements. Using this alternative, the computation of worst-case execution time (WCET) is simplified. Whereas in DLAlloc, the execution time depends on the current state of the allocator and on the previous de- / allocations, TLSF provides a bounded execution time regardless of its former operation at the cost of higher internal fragmentation.

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

5.3 Resource Management

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

The implementation of the resource management, containing a driver manager, is geared towards the POSIX standard [11]. It 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 synchronization. For this, the Thread Manager already pro-

vides 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, the timing behaviour depends on this quantity. For the use in real-time applications, the number of drivers must be limited to guarantee device accesses in bounded time. As the number of devices an application uses is known in advance, constant-access-time handlers can be arranged during the preparation of the application's execution environment. Thus, device accesses can be performed in constant time.

6 Conclusion

In this report we presented the refined 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 synchronization. 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. For thread synchronization it provides lock (mutex) and conditional variables. The locks are not equipped with a priority inheritance mechanism, because through the timing-based scheduling technique of the hard and soft real-time threads no priority inversions can occur. 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 multithreaded hardware, the software provides mechanisms for synchronization. So it is easy for an application programmer to avoid race conditions between different threads running in parallel.

In the future, the system-level software will be extended towards the MERASA multi-core processor. This work will be done in parallel with the multi-core development. Concerning the thread management, especially the synchronization unit, it is necessary to guarantee the avoidance of deadlocks. Due to the low-level parallelism in a SMT processor, a traditional priority ceiling protocol will not be sufficient.

A File description

A.1 ccerrno.h File Reference

```
#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

Functions

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

A.1.1 Define Documentation

A.1.1.1 #define CCERRNO_H_ 1

Definition at line 35 of file ccerrno.h.

A.1.1.2 #define E2BIG 7

Argument list too long

Definition at line 58 of file ccerrno.h.

A.1.1.3 #define E_OK 0

Definition at line 47 of file ccerrno.h.

A.1.1.4 #define EACCES 13

Permission denied

Definition at line 64 of file ccerrno.h.

A.1.1.5 #define EADDRINUSE 98

Address already in use

Definition at line 156 of file ccerrno.h.

A.1.1.6 #define EADDRNOTAVAIL 99

Cannot assign requested address

Definition at line 157 of file ccerrno.h.

A.1.1.7 #define EADV 68

Advertise error

Definition at line 126 of file ccerrno.h.

A.1.1.8 #define EAFNOSUPPORT 97

Address family not supported by protocol

Definition at line 155 of file ccerrno.h.

A.1.1.9 #define EAGAIN 11

Try again

Definition at line 62 of file ccerrno.h.

A.1.1.10 #define EALREADY 114

Operation already in progress

Definition at line 172 of file ccerrno.h.

A.1.1.11 #define EBADE 52

Invalid exchange

Definition at line 108 of file ccerrno.h.

A.1.1.12 #define EBADF 9

Bad file number

Definition at line 60 of file ccerrno.h.

A.1.1.13 #define EBADFD 77

File descriptor in bad state

Definition at line 135 of file ccerrno.h.

A.1.1.14 #define EBADMSG 74

Not a data message

Definition at line 132 of file ccerrno.h.

A.1.1.15 #define EBADR 53

Invalid request descriptor

Definition at line 109 of file ccerrno.h.

A.1.1.16 #define EBADRQC 56

Invalid request code

Definition at line 112 of file ccerrno.h.

A.1.1.17 #define EBADSLT 57

Invalid slot

Definition at line 113 of file ccerrno.h.

A.1.1.18 #define EBFONT 59

Bad font file format

Definition at line 117 of file ccerrno.h.

A.1.1.19 #define EBUSY 16

Device or resource busy

Definition at line 67 of file ccerrno.h.

A.1.1.20 #define ECANCELED 125

Operation Canceled

Definition at line 184 of file ccerrno.h.

A.1.1.21 #define ECHILD 10

No child processes

Definition at line 61 of file ccerrno.h.

A.1.1.22 #define ECHRNG 44

Channel number out of range

Definition at line 100 of file ccerrno.h.

A.1.1.23 #define ECOMM 70

Communication error on send

Definition at line 128 of file ccerrno.h.

A.1.1.24 #define ECONNABORTED 103

Software caused connection abort

Definition at line 161 of file ccerrno.h.

A.1.1.25 #define ECONNREFUSED 111

Connection refused

Definition at line 169 of file ccerrno.h.

A.1.1.26 #define ECONNRESET 104

Connection reset by peer

Definition at line 162 of file ccerrno.h.

A.1.1.27 #define EDEADLK 35

Resource deadlock would occur

Definition at line 91 of file ccerrno.h.

A.1.1.28 #define EDEADLOCK EDEADLK

Definition at line 115 of file ccerrno.h.

A.1.1.29 #define EDESTADDRREQ 89

Destination address required

Definition at line 147 of file ccerrno.h.

A.1.1.30 #define EDOM 33

Math argument out of domain of func
Definition at line 84 of file ccerrno.h.

A.1.1.31 #define EDOTDOT 73

RFS specific error
Definition at line 131 of file ccerrno.h.

A.1.1.32 #define EDQUOT 122

Quota exceeded
Definition at line 180 of file ccerrno.h.

A.1.1.33 #define EEXIST 17

File exists
Definition at line 68 of file ccerrno.h.

A.1.1.34 #define EFAULT 14

Bad address
Definition at line 65 of file ccerrno.h.

A.1.1.35 #define EFBIG 27

File too large
Definition at line 78 of file ccerrno.h.

A.1.1.36 #define EHOSTDOWN 112

Host is down
Definition at line 170 of file ccerrno.h.

A.1.1.37 #define EHOSTUNREACH 113

No route to host
Definition at line 171 of file ccerrno.h.

A.1.1.38 #define EIDRM 43

Identifier removed
Definition at line 99 of file ccerrno.h.

A.1.1.39 #define EILSEQ 84

Illegal byte sequence

Definition at line 142 of file ccerrno.h.

A.1.1.40 #define EINPROGRESS 115

Operation now in progress

Definition at line 173 of file ccerrno.h.

A.1.1.41 #define EINTR 4

Interrupted system call

Definition at line 55 of file ccerrno.h.

A.1.1.42 #define EINVAL 22

Invalid argument

Definition at line 73 of file ccerrno.h.

A.1.1.43 #define EIO 5

I/O error

Definition at line 56 of file ccerrno.h.

A.1.1.44 #define EISCONN 106

Transport endpoint is already connected

Definition at line 164 of file ccerrno.h.

A.1.1.45 #define EISDIR 21

Is a directory

Definition at line 72 of file ccerrno.h.

A.1.1.46 #define EISNAM 120

Is a named type file

Definition at line 178 of file ccerrno.h.

A.1.1.47 #define EKEYEXPIRED 127

Key has expired

Definition at line 186 of file ccerrno.h.

A.1.1.48 #define EKEYREJECTED 129

Key was rejected by service

Definition at line 188 of file ccerrno.h.

A.1.1.49 #define EKEYREVOKED 128

Key has been revoked

Definition at line 187 of file ccerrno.h.

A.1.1.50 #define EL2HLT 51

Level 2 halted

Definition at line 107 of file ccerrno.h.

A.1.1.51 #define EL2NSYNC 45

Level 2 not synchronized

Definition at line 101 of file ccerrno.h.

A.1.1.52 #define EL3HLT 46

Level 3 halted

Definition at line 102 of file ccerrno.h.

A.1.1.53 #define EL3RST 47

Level 3 reset

Definition at line 103 of file ccerrno.h.

A.1.1.54 #define ELIBACC 79

Can not access a needed shared library

Definition at line 137 of file ccerrno.h.

A.1.1.55 #define ELIBBAD 80

Accessing a corrupted shared library

Definition at line 138 of file ccerrno.h.

A.1.1.56 #define ELIBEXEC 83

Cannot exec a shared library directly

Definition at line 141 of file ccerrno.h.

A.1.1.57 #define ELIBMAX 82

Attempting to link in too many shared libraries

Definition at line 140 of file ccerrno.h.

A.1.1.58 #define ELIBSCN 81

.lib section in a.out corrupted

Definition at line 139 of file ccerrno.h.

A.1.1.59 #define ELNRNG 48

Link number out of range

Definition at line 104 of file ccerrno.h.

A.1.1.60 #define ELOOP 40

Too many symbolic links encountered

Definition at line 96 of file ccerrno.h.

A.1.1.61 #define EMEDIUMTYPE 124

Wrong medium type

Definition at line 183 of file ccerrno.h.

A.1.1.62 #define EMFILE 24

Too many open files

Definition at line 75 of file ccerrno.h.

A.1.1.63 #define EMLINK 31

Too many links

Definition at line 82 of file ccerrno.h.

A.1.1.64 #define EMSGSIZE 90

Message too long

Definition at line 148 of file ccerrno.h.

A.1.1.65 #define EMULTIHOP 72

Multihop attempted

Definition at line 130 of file ccerrno.h.

A.1.1.66 #define ENAMETOOLONG 36

File name too long

Definition at line 92 of file ccerrno.h.

A.1.1.67 #define ENAVAIL 119

No XENIX semaphores available

Definition at line 177 of file ccerrno.h.

A.1.1.68 #define ENETDOWN 100

Network is down

Definition at line 158 of file ccerrno.h.

A.1.1.69 #define ENETRESET 102

Network dropped connection because of reset

Definition at line 160 of file ccerrno.h.

A.1.1.70 #define ENETUNREACH 101

Network is unreachable

Definition at line 159 of file ccerrno.h.

A.1.1.71 #define ENFILE 23

File table overflow

Definition at line 74 of file ccerrno.h.

A.1.1.72 #define ENOANO 55

No anode

Definition at line 111 of file ccerrno.h.

A.1.1.73 #define ENOBUFS 105

No buffer space available

Definition at line 163 of file ccerrno.h.

A.1.1.74 #define ENOCSI 50

No CSI structure available

Definition at line 106 of file ccerrno.h.

A.1.1.75 #define ENODATA 61

No data available

Definition at line 119 of file ccerrno.h.

A.1.1.76 #define ENODEV 19

No such device

Definition at line 70 of file ccerrno.h.

A.1.1.77 #define ENOENT 2

No such file or directory

Definition at line 53 of file ccerrno.h.

A.1.1.78 #define ENOEXEC 8

Exec format error

Definition at line 59 of file ccerrno.h.

A.1.1.79 #define ENOKEY 126

Required key not available

Definition at line 185 of file ccerrno.h.

A.1.1.80 #define ENOLCK 37

No record locks available

Definition at line 93 of file ccerrno.h.

A.1.1.81 #define ENOLINK 67

Link has been severed

Definition at line 125 of file ccerrno.h.

A.1.1.82 #define ENOMEDIUM 123

No medium found

Definition at line 182 of file ccerrno.h.

A.1.1.83 #define ENOMEM 12

Out of memory

Definition at line 63 of file ccerrno.h.

A.1.1.84 #define ENOMSG 42

No message of desired type

Definition at line 98 of file ccerrno.h.

A.1.1.85 #define ENONET 64

Machine is not on the network

Definition at line 122 of file ccerrno.h.

A.1.1.86 #define ENOPKG 65

Package not installed

Definition at line 123 of file ccerrno.h.

A.1.1.87 #define ENOPROTOOPT 92

Protocol not available

Definition at line 150 of file ccerrno.h.

A.1.1.88 #define ENOSPC 28

No space left on device

Definition at line 79 of file ccerrno.h.

A.1.1.89 #define ENOSR 63

Out of streams resources

Definition at line 121 of file ccerrno.h.

A.1.1.90 #define ENOSTR 60

Device not a stream

Definition at line 118 of file ccerrno.h.

A.1.1.91 #define ENOSYS 38

Function not implemented

Definition at line 94 of file ccerrno.h.

A.1.1.92 #define ENOTBLK 15

Block device required

Definition at line 66 of file ccerrno.h.

A.1.1.93 #define ENOTCONN 107

Transport endpoint is not connected
Definition at line 165 of file ccerrno.h.

A.1.1.94 #define ENOTDIR 20

Not a directory
Definition at line 71 of file ccerrno.h.

A.1.1.95 #define ENOTEMPTY 39

Directory not empty
Definition at line 95 of file ccerrno.h.

A.1.1.96 #define ENOTNAM 118

Not a XENIX named type file
Definition at line 176 of file ccerrno.h.

A.1.1.97 #define ENOTRECOVERABLE 131

State not recoverable
Definition at line 192 of file ccerrno.h.

A.1.1.98 #define ENOTSOCK 88

Socket operation on non-socket
Definition at line 146 of file ccerrno.h.

A.1.1.99 #define ENOTTY 25

Not a typewriter
Definition at line 76 of file ccerrno.h.

A.1.1.100 #define ENOTUNIQ 76

Name not unique on network
Definition at line 134 of file ccerrno.h.

A.1.1.101 #define ENXIO 6

No such device or address
Definition at line 57 of file ccerrno.h.

A.1.1.102 #define EOPNOTSUPP 95

Operation not supported on transport endpoint

Definition at line 153 of file ccerrno.h.

A.1.1.103 #define EOVERFLOW 75

Value too large for defined data type

Definition at line 133 of file ccerrno.h.

A.1.1.104 #define EOWNERDEAD 130

Owner died

Definition at line 191 of file ccerrno.h.

A.1.1.105 #define EPERM 1

Operation not permitted

Definition at line 52 of file ccerrno.h.

A.1.1.106 #define EPNOSUPPORT 96

Protocol family not supported

Definition at line 154 of file ccerrno.h.

A.1.1.107 #define EPIPE 32

Broken pipe

Definition at line 83 of file ccerrno.h.

A.1.1.108 #define EPROTO 71

Protocol error

Definition at line 129 of file ccerrno.h.

A.1.1.109 #define EPROTONOSUPPORT 93

Protocol not supported

Definition at line 151 of file ccerrno.h.

A.1.1.110 #define EPROTOTYPE 91

Protocol wrong type for socket

Definition at line 149 of file ccerrno.h.

A.1.1.111 #define ERANGE 34

Math result not representable

Definition at line 85 of file ccerrno.h.

A.1.1.112 #define EREMCHG 78

Remote address changed

Definition at line 136 of file ccerrno.h.

A.1.1.113 #define EREMOTE 66

Object is remote

Definition at line 124 of file ccerrno.h.

A.1.1.114 #define EREMOTEIO 121

Remote I/O error

Definition at line 179 of file ccerrno.h.

A.1.1.115 #define ERESTART 85

Interrupted system call should be restarted

Definition at line 143 of file ccerrno.h.

A.1.1.116 #define EROFS 30

Read-only file system

Definition at line 81 of file ccerrno.h.

A.1.1.117 #define ESHUTDOWN 108

Cannot send after transport endpoint shutdown

Definition at line 166 of file ccerrno.h.

A.1.1.118 #define ESOCKTNOSUPPORT 94

Socket type not supported

Definition at line 152 of file ccerrno.h.

A.1.1.119 #define ESPIPE 29

Illegal seek

Definition at line 80 of file ccerrno.h.

A.1.1.120 #define ESRCH 3

No such process

Definition at line 54 of file ccerrno.h.

A.1.1.121 #define ESRMNT 69

Srmount error

Definition at line 127 of file ccerrno.h.

A.1.1.122 #define ESTALE 116

Stale NFS file handle

Definition at line 174 of file ccerrno.h.

A.1.1.123 #define ESTRPIPE 86

Streams pipe error

Definition at line 144 of file ccerrno.h.

A.1.1.124 #define ETIME 62

Timer expired

Definition at line 120 of file ccerrno.h.

A.1.1.125 #define ETIMEDOUT 110

Connection timed out

Definition at line 168 of file ccerrno.h.

A.1.1.126 #define ETOOMANYREFS 109

Too many references: cannot splice

Definition at line 167 of file ccerrno.h.

A.1.1.127 #define ETXTBSY 26

Text file busy

Definition at line 77 of file ccerrno.h.

A.1.1.128 #define EUCLEAN 117

Structure needs cleaning

Definition at line 175 of file ccerrno.h.

A.1.1.129 #define EUNATCH 49

Protocol driver not attached

Definition at line 105 of file ccerrno.h.

A.1.1.130 #define EUSERS 87

Too many users

Definition at line 145 of file ccerrno.h.

A.1.1.131 #define EWOULDBLOCK EAGAIN

Operation would block

Definition at line 97 of file ccerrno.h.

A.1.1.132 #define EXDEV 18

Cross-device link

Definition at line 69 of file ccerrno.h.

A.1.1.133 #define EXFULL 54

Exchange full

Definition at line 110 of file ccerrno.h.

A.1.2 Function Documentation**A.1.2.1 error_t get_errno (void)****A.1.2.2 void set_errno (error_t *errno*)**

A.2 driver.h File Reference

Macros for writing MERASA device drivers This file provides the macros and data types that are necessary to write a hardware device driver for the MERASA Operating System. For examples how to use them, see existing driver files.

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

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

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

Data Structures

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

Defines

- #define **DRIVER_H_1**
- #define **SDRIVER**(n, v, i, c, o)
Macro for a static 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**
- typedef **driver** **driver_t**

A.2.1 Detailed Description

Macros for writing MERASA device drivers This file provides the macros and data types that are necessary to write a hardware device driver for the MERASA Operating System. 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`.

A.2.2 Define Documentation

A.2.2.1 `#define DRIVER_H_ 1`

Definition at line 49 of file `driver.h`.

A.2.2.2 `#define SDRIVER(n, v, i, c, o)`

Value:

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

Macro for a static driver.

Parameters:

- n* name
- v* version
- i* init function
- c* cleanup function
- o* operations struct (`drvif_t` (p. 36))

Definition at line 131 of file `driver.h`.

A.2.3 Typedef Documentation

A.2.3.1 `typedef struct driver driver_t`

This struct describes a device driver. Do not use it directly, instead publish your driver using the `DRIVER` macro below!

A.2.3.2 `typedef int32_t(*) drv_cleanup_fn_t(void)`

Definition at line 73 of file `driver.h`.

A.2.3.3 `typedef int32_t(*) drv_close_fn(void)`

Definition at line 76 of file `driver.h`.

A.2.3.4 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 72 of file driver.h.

A.2.3.5 typedef size_t(*) drv_ioctl_fn(uint32_t, va_list)

Definition at line 79 of file driver.h.

A.2.3.6 typedef int32_t(*) drv_open_fn(void)

Definition at line 75 of file driver.h.

A.2.3.7 typedef size_t(*) drv_read_fn(void *, size_t)

Definition at line 77 of file driver.h.

A.2.3.8 typedef size_t(*) drv_write_fn(const void *, size_t)

Definition at line 78 of file driver.h.

A.2.3.9 typedef struct driver_interface drvif_t

This strict holds the functions a driver must implement

A.3 drivermanager.h File Reference

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

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

Defines

- `#define DRIVERMANAGER_H_1`
- `#define MAX_DRV_PERFORMANCE (0xFF)`
- `#define EXTERN_DRIVER(name) extern driver_t sdriver_ ##name;`
- `#define REGISTER_DRIVER(name) &sdriver_ ##name`

Typedefs

- `typedef int32_t drv_handler`

Functions

- `uint32_t install_driver (const void *elf, size_t len, const char *dev_id, const void *base_address)`
- `uint32_t remove_driver (const char *dev_id)`
- `drv_handler cc_open (const char *driver, uint32_t flags)`
- `drv_handler cc_bopen (const char *driver, uint32_t flags)`
- `uint32_t cc_close (drv_handler handle)`
- `size_t cc_read (drv_handler handle, void *buf, size_t count)`
- `size_t cc_write (drv_handler handle, const void *buf, size_t count)`
- `int32_t cc_ioctl (drv_handler handle, uint32_t request,...)`
- `thread_handler get_drv_owner (const char *dev_id)`

A.3.1 Define Documentation

A.3.1.1 `#define DRIVERMANAGER_H_1`

Definition at line 36 of file drivermanager.h.

A.3.1.2 `#define EXTERN_DRIVER(name) extern driver_t sdriver_ ##name;`

Macro for the declaration of a driver.

Definition at line 190 of file drivermanager.h.

A.3.1.3 #define MAX_DRV_PERFORMANCE (0xFF)

Definition at line 50 of file drivermanager.h.

A.3.1.4 #define REGISTER_DRIVER(name) &sdriver_##name

Macro to register a driver in the user application

Definition at line 196 of file drivermanager.h.

A.3.2 Typedef Documentation

A.3.2.1 typedef int32_t drv_handler

Definition at line 55 of file drivermanager.h.

A.3.3 Function Documentation

A.3.3.1 drv_handler cc_bopen (const char * *driver*, uint32_t *flags*)

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

Parameters:

driver device/driver identifier

flags flags

Returns:

a handler for the device; -1 if the device could not be opened. Check `errno` via `get_errno` (p. 33) for details: **EBUSY** (p. 20) if the device is already opened; **ENODEV** (p. 27) if the specified device does not exist

A.3.3.2 uint32_t cc_close (drv_handler *handle*)

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

Parameters:

handle the device handler

Returns:

EACCES (p. 19) the thread does not own the device; **EPERM** if it is not on top of the thread's device stack

Deprecated

Use POSIX interface in `unistd.h` (p. 71) for this function!

A.3.3.3 int32_t cc_ioctl (drv_handler *handle*, uint32_t *request*, ...)

Perform a control operation. 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.19) if the calling thread does not own the device Further errnos might be set by the driver, see driver documentation therefor.

Deprecated

Use POSIX interface in **stropts.h** (p.67) for this function!

A.3.3.4 drv_handler cc_open (const char * *driver*, uint32_t *flags*)

Open a device

Parameters:

driver device/driver identifier

flags flags

Returns:

a handler for the device; -1 if the device could not be opened. Check errno via **get_errno** (p.33) for details: **EBUSY** (p.20) if the device is already opened; **ENODEV** (p.27) if the specified device does not exist

Deprecated

Use POSIX interface in **fcntl.h** (p.42) for this function!

A.3.3.5 size_t cc_read (drv_handler *handle*, void * *buf*, size_t *count*)

Read from a device

Parameters:

handle the handler of the device

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

count max 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.33)): **EACCES** (p.19) if the calling thread does not own the device; **EPERM** (p.30) if buf is not

writable by the calling thread. Further `errno`s might be set by the driver, see driver documentation therefor.

Deprecated

Use POSIX interface in `unistd.h` (p. 71) for this function!

A.3.3.6 `size_t cc_write (drv_handler handle, const void * buf, size_t count)`

Write to a device

Parameters:

handle the handler of the device

buf the data to write

count number of bytes to write

Returns:

the number of bytes written. If the operation fails, -1 is returned. For more information check `errno` (`get_errno` (p. 33)): `EACCES` (p. 19) if the calling thread does not own the device. Further `errno`s might be set by the driver, see driver documentation therefor.

Deprecated

Use POSIX interface in `unistd.h` (p. 71) for this function!

A.3.3.7 `thread_handler get_drv_owner (const char * dev_id)`

Read the current owner of the specified device

Parameters:

dev_id device/driver identifier

Returns:

the handler for the driver's owner, `NO_THREAD` if the driver is currently free

A.3.3.8 `uint32_t install_driver (const void * elf, size_t len, const char * dev_id, const void * base_address)`

Install a driver for a device

Parameters:

elf the ELF image

len 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. 30) if the thread doesn't have the privilege to use the driver manager (`THREAD_PRIV_DRVS`); **EEXIST** (p. 22) if a driver with the same ID is already loaded

A.3.3.9 uint32_t remove_driver (const char * dev_id)

Remove a driver from the system

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_DRVS`); **EBUSY** (p. 20) if the driver is currently used; **ENODEV** (p. 27) if the given device *dev_id* does not exist

A.4 fcntl.h File Reference

Defines

- `#define FCNTL_H_1`

Functions

- `int open (const char *path, int flags,...)`
Open or create a device for reading or writing.

A.4.1 Define Documentation

A.4.1.1 `#define FCNTL_H_1`

Definition at line 35 of file fcntl.h.

A.4.2 Function Documentation

A.4.2.1 `int open (const char * path, int flags, ...)`

Open or create a device for reading or writing.

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

Returns:

If successful, `open()` (p. 42) returns a non-negative integer, termed a device descriptor. It returns -1 on failure, and sets `errno` to indicate the error.

A.5 log.h File Reference

```
#include "../config.h"
#include <memdev.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 **LOGLEVEL** LOGLEVEL_WARN
- #define **FASTLOG**(c) { P32(VO_ADDRESS) = (c); P32(VO_ADDRESS) = '\n'; }
- #define **LOGWORD**(w) { P32(VO_WORD) = (w); }
- #define **LOGDBL_D4C**(w, c1, c2, c3, c4)
- #define **LOGDBL_X4C**(w, c1, c2, c3, c4)
- #define **LOGDBL_DX**(d, h)
- #define **LOCK_LOG**() cthread_mutex_lock(&log_mutex)
- #define **UNLOCK_LOG**() cthread_mutex_unlock(&log_mutex)
- #define **log_debug_s**(msg, args...) ((void)0)
- #define **log_debug**(msg) ((void)0)
- #define **log_info_s**(msg, args...) ((void)0)
- #define **log_info**(msg) ((void)0)
- #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`

A.5.1 Detailed Description

This file provides logging facilities by means of different loglevels (DEBUG, INFO, WARN, ERR, FATAL, NONE). You can use the loglevel defined globally in this file (WARN). If you want to use a different loglevel for your file, just set the LOGLEVEL macro to the intended value (see LOGLEVEL_ definitions below) **before** including this header.

Definition in file `log.h`.

A.5.2 Define Documentation

A.5.2.1 #define FASTLOG(c) { P32(VO_ADDRESS) = (c); P32(VO_ADDRESS) = '\n'; }

Definition at line 100 of file `log.h`.

A.5.2.2 #define LOCK_LOG() ccthread_mutex_lock(&log_mutex)

Definition at line 123 of file `log.h`.

A.5.2.3 #define log_debug(msg) ((void)0)

Definition at line 146 of file `log.h`.

A.5.2.4 #define log_debug_s(msg, args...) ((void)0)

Definition at line 137 of file `log.h`.

A.5.2.5 #define log_err(msg)

Value:

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

Definition at line 213 of file `log.h`.

A.5.2.6 #define log_err_s(msg, args...)

Value:

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

Definition at line 200 of file log.h.

A.5.2.7 #define log_fatal(msg)

Value:

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

Definition at line 237 of file log.h.

A.5.2.8 #define log_fatal_s(msg, args...)

Value:

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

Definition at line 224 of file log.h.

A.5.2.9 #define LOG_H_1

Definition at line 48 of file log.h.

A.5.2.10 #define log_info(msg) ((void)0)

Definition at line 170 of file log.h.

A.5.2.11 #define log_info_s(msg, args...) ((void)0)

Definition at line 161 of file log.h.

A.5.2.12 #define log_warn(msg)**Value:**

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

Definition at line 189 of file log.h.

A.5.2.13 #define log_warn_s(msg, args...)**Value:**

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

Definition at line 176 of file log.h.

A.5.2.14 #define LOGDBL_D4C(w, c1, c2, c3, c4)**Value:**

```
{
uint64_t v = (w) | ((uint64_t)(c1)<<56) | ((uint64_t)(c2)<<48) | ((uint64_t)(c3)<<40) | ((uint64_t)(c4)<<32);
P64(V0_DBL_DCCCC) = v;
}
```

Decimal + 4 chars (lo+hi)

Definition at line 105 of file log.h.

A.5.2.15 #define LOGDBL_DX(d, h)**Value:**

```
{
uint64_t v = (d) | (((uint64_t)(h))<<32);
P64(V0_DBL_DX) = v;
}
```

Dec + Hex (lo+hi)

Definition at line 117 of file log.h.

A.5.2.16 #define LOGDBL_X4C(w, c1, c2, c3, c4)

Value:

```

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

```

Hex + 4 chars (lo+hi)

Definition at line 111 of file log.h.

A.5.2.17 #define LOGLEVEL LOGLEVEL_WARN

Definition at line 69 of file log.h.

A.5.2.18 #define LOGLEVEL_DEBUG 5

Definition at line 61 of file log.h.

A.5.2.19 #define LOGLEVEL_ERR 2

Definition at line 64 of file log.h.

A.5.2.20 #define LOGLEVEL_FATAL 1

Definition at line 65 of file log.h.

A.5.2.21 #define LOGLEVEL_INFO 4

Definition at line 62 of file log.h.

A.5.2.22 #define LOGLEVEL_NONE 0

Definition at line 66 of file log.h.

A.5.2.23 #define LOGLEVEL_WARN 3

Definition at line 63 of file log.h.

A.5.2.24 #define LOGWORD(w) { P32(VO_WORD) = (w); }

Definition at line 101 of file log.h.

A.5.2.25 #define UNLOCK_LOG() ccthread_mutex_unlock(&log_mutex)

Definition at line 124 of file log.h.

A.5.3 Function Documentation

A.5.3.1 int sprintf (char * *str*, const char * *format*, ...)

A.5.4 Variable Documentation

A.5.4.1 ccthread_mutex_t log_mutex

A.6 memdev.h File Reference

Defines

- `#define MEMDEV_HPP_1`
- `#define SERVICE_MASK (0xfffff00)`
- `#define PER_FMSG (0xe0000000)`
- `#define PER_VIO (0xe0010000)`
- `#define PER_BAT (0xe0010400)`
- `#define PER_BATC (0xe0010500)`
- `#define PER_CLK (0xe0010800)`
- `#define PER_CLKB (0xe0010900)`
- `#define PER_CLKT (0xe0010a00)`
- `#define PER_RTC (0xe0010c00)`
- `#define VO_ADDRESS (PER_VIO + 0x0)`
- `#define VO_WORD (PER_VIO + 0x4)`
- `#define VO_DBL_DCCCC (PER_VIO + 0x10)`
- `#define VO_DBL_XCCCC (PER_VIO + 0x18)`
- `#define VO_DBL_DX (PER_VIO + 0x20)`
- `#define VIO_PREFIX "%#"`
- `#define P32(a) (*((uint32_t volatile *) a)`
- `#define P64(a) (*((uint64_t volatile *) a)`

A.6.1 Define Documentation

A.6.1.1 `#define MEMDEV_HPP_1`

Definition at line 35 of file memdev.h.

A.6.1.2 `#define P32(a) (*((uint32_t volatile *) a)`

Fast access to 32-bit value

Definition at line 88 of file memdev.h.

A.6.1.3 `#define P64(a) (*((uint64_t volatile *) a)`

Fast access to 64-bit value

Definition at line 90 of file memdev.h.

A.6.1.4 `#define PER_BAT (0xe0010400)`

Definition at line 57 of file memdev.h.

A.6.1.5 #define PER_BATC (0xe0010500)

Definition at line 58 of file memdev.h.

A.6.1.6 #define PER_CLK (0xe0010800)

Definition at line 59 of file memdev.h.

A.6.1.7 #define PER_CLKB (0xe0010900)

Definition at line 60 of file memdev.h.

A.6.1.8 #define PER_CLKT (0xe0010a00)

Definition at line 61 of file memdev.h.

A.6.1.9 #define PER_FMSG (0xe0000000)

Definition at line 55 of file memdev.h.

A.6.1.10 #define PER_RTC (0xe0010c00)

Definition at line 62 of file memdev.h.

A.6.1.11 #define PER_VIO (0xe0010000)

Definition at line 56 of file memdev.h.

A.6.1.12 #define SERVICE_MASK (0xfffff00)

Definition at line 53 of file memdev.h.

A.6.1.13 #define VIO_PREFIX "%#"

allows filtering of VIO output in console mode

Definition at line 83 of file memdev.h.

A.6.1.14 #define VO_ADDRESS (PER_VIO + 0x0)

bytes written to this address are put into the virtual output

Definition at line 68 of file memdev.h.

A.6.1.15 #define VO_DBL_DCCCC (PER_VIO + 0x10)

special log, hi is interpreted as chars, lo as word dec

Definition at line 74 of file memdev.h.

A.6.1.16 #define VO_DBL_DX (PER_VIO + 0x20)

special log, both words are interpreted as words, separated by |; hi is hex, lo is dec

Definition at line 80 of file memdev.h.

A.6.1.17 #define VO_DBL_XCCCC (PER_VIO + 0x18)

special log, hi is interpreted as chars, lo as word hex

Definition at line 77 of file memdev.h.

A.6.1.18 #define VO_WORD (PER_VIO + 0x4)

write one word to STDOUT

Definition at line 71 of file memdev.h.

A.7 memory-desc.h File Reference

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

Data Structures

- struct `mem_cfg_data`

Defines

- `#define MEMORY_DESC_H_1`
- `#define MEMORY_CFG(b, l, ac, c, f)`

Typedefs

- typedef `mem_cfg_data` `memory_t`

Variables

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

A.7.1 Define Documentation

A.7.1.1 `#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* begin
- l* lenght
- ac* access_cycles
- c* cost
- f* flags

Definition at line 92 of file memory-desc.h.

A.7.1.2 #define MEMORY_DESC_H_1

Definition at line 35 of file memory-desc.h.

A.7.2 Typedef Documentation

A.7.2.1 typedef struct mem_cfg_data memory_t

A.7.3 Variable Documentation

A.7.3.1 const size_t mem_config_len

For correct access to `node_mem_config` (p. 53), you have to set this constant as `sizeof(node_mem_config)/sizeof(memory_t)`.

A.7.3.2 memory_t node_mem_config[]

You need to define this constant in the OS for one specific node configuration. Use the `MEMORY_CFG` (p. 52) macro below for filling this array!

A.8 memory.h File Reference

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

Defines

- `#define MEMORY_H_1`

Functions

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

A.8.1 Define Documentation

A.8.1.1 `#define MEMORY_H_1`

Definition at line 35 of file memory.h.

A.8.2 Function Documentation

A.8.2.1 `void* calloc (size_t number, size_t size)`

The `calloc()` (p. 54) function allocates space for `number` objects, each `size` bytes in length. The result is identical to calling `malloc()` (p. 55) with an argument of “`number * size`”, with the exception that the allocated memory is explicitly initialized to zero bytes.

A.8.2.2 `void free (void * ptr)`

Free a previously allocated block of the current thread

Parameters:

ptr the memory block

A.8.2.3 `bool_t has_write_permission (void * mem)`

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

A.8.2.4 void* malloc (size_t size)

Allocate memory for the current thread

Parameters:

size amount of memory to allocate

A.8.2.5 void* realloc (void * ptr, size_t size)

The `realloc()` (p. 55) function changes the size of the previously allocated memory referenced by `ptr` to `size` bytes. The contents of the memory are unchanged up to the lesser of the new and old sizes. If the new size is larger, the value of the newly allocated portion of the memory is undefined. Upon success, the memory referenced by `ptr` is freed and a pointer to the newly allocated memory is returned.

A.8.2.6 void* tcmemcpy (void * dest, const void * src, size_t n)

effective copying of (non-overlapping!) memory

A.9 merasa-ssw.h File Reference

```
#include <cerrno.h>
#include <ccthread.h>
#include <ccthreadtypes.h>
#include <csfr.h>
#include <driver.h>
#include <drivermanager.h>
#include <fcntl.h>
#include <log.h>
#include <memdev.h>
#include <memory-desc.h>
#include <memory.h>
#include <pthread.h>
#include <regcarcore.h>
#include <stropts.h>
#include <sync.h>
#include <sysmonitor.h>
#include <tcintrinsics.h>
#include <types.h>
#include <unistd.h>
```

A.10 pthread.h File Reference

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

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

```
#include <sched.h>
```

Defines

- `#define PTHREAD_H_ 1`

Functions

- int **pthread_create** (pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)
Creates a new thread of execution.
- int **pthread_join** (pthread_t thread, void **value_ptr)
Causes the calling thread to wait for the termination of the specified thread.
- pthread_t **pthread_self** (void)
Returns the thread ID of the calling thread.
- void **pthread_yield** (void)
Allows the scheduler to run another thread instead of the current one.
- int **pthread_attr_getschedparam** (const pthread_attr_t *attr, sched_param *param)
Get the scheduling parameter attribute from a thread attributes object.
- int **pthread_attr_getschedpolicy** (const pthread_attr_t *attr, int *policy)
Get the scheduling policy attribute from a thread attributes object.
- int **pthread_attr_init** (pthread_attr_t *attr)
Initialize a thread attributes object with default values.
- int **pthread_attr_setschedparam** (pthread_attr_t *attr, sched_param *param)
Set the scheduling parameter attribute in a thread attributes object.
- int **pthread_attr_setschedpolicy** (pthread_attr_t *attr, int policy)

Set the scheduling policy attribute in a thread attributes object.

- int **pthread_attr_getmemory** (pthread_attr_t *attr, memory_t **mem)

Get the memory type used for the thread.

- int **pthread_attr_setmemory** (pthread_attr_t *attr, memory_t *mem)

Set the memory to use for the thread.

- int **pthread_attr_getbasicheapsize** (pthread_attr_t *attr, int *heapsize)

Get the initial heap size used for the thread.

- int **pthread_attr_setbasicheapsize** (pthread_attr_t *attr, int heapsize)

Set the initial heap size used for the thread.

- int **pthread_attr_getflags** (pthread_attr_t *attr, int *flags)

Get thread flags describing the thread's behaviour.

- int **pthread_attr_setflags** (pthread_attr_t *attr, int flags)

Set thread flags describing the thread's behaviour.

- int **pthread_attr_getiq** (pthread_attr_t *attr, int *iq)

Get the instruction quantum.

- int **pthread_attr_setiq** (pthread_attr_t *attr, int iq)

Set the instruction quantum.

- int **pthread_mutex_destroy** (pthread_mutex_t *mutex)

Destroy a mutex.

- int **pthread_mutex_init** (pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)

Initialize a mutex with specified attributes.

- int **pthread_mutex_lock** (pthread_mutex_t *mutex)

Lock a mutex and block until it becomes available.

- int **pthread_mutex_trylock** (pthread_mutex_t *mutex)

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

including the current thread.

- int **pthread_mutex_unlock** (pthread_mutex_t *mutex)
Unlock a mutex.
- int **pthread_cond_broadcast** (pthread_cond_t *cond)
Unblock all threads currently blocked on the specified condition variable.
- int **pthread_cond_destroy** (pthread_cond_t *cond)
Destroy a condition variable.
- int **pthread_cond_init** (pthread_cond_t *cond, const pthread_condattr_t *attr)
Initialize a condition variable with specified attributes.
- int **pthread_cond_signal** (pthread_cond_t *cond)
Unblock at least one of the threads blocked on the specified condition variable.
- int **pthread_cond_wait** (pthread_cond_t *, pthread_mutex_t *mutex)
Wait for a condition and lock the specified mutex.

A.10.1 Define Documentation

A.10.1.1 #define PTHREAD_H_1

Definition at line 35 of file pthread.h.

A.10.2 Function Documentation

A.10.2.1 int pthread_attr_getbasiceapsize (pthread_attr_t * attr, int * heapsize)

Get the initial heap size used for the thread.

See **pthread_attr_setbasiceapsize()** (p. 61) for more details.

Returns:

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

A.10.2.2 int pthread_attr_getflags (pthread_attr_t * attr, int * flags)

Get thread flags describing the thread's behaviour.

See `pthread_attr_setflags()` (p.61) for an explanation of possible values.

Returns:

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

A.10.2.3 int pthread_attr_getiq (pthread_attr_t * attr, int * iq)

Get the instruction quantum.

Returns:

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

A.10.2.4 int pthread_attr_getmemory (pthread_attr_t * attr, memory_t ** mem)

Get the memory type used for the thread.

Returns:

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

A.10.2.5 int pthread_attr_getschedparam (const pthread_attr_t * attr, sched_param * param)

Get the scheduling parameter attribute from a thread attributes object.

A.10.2.6 int pthread_attr_getschedpolicy (const pthread_attr_t * attr, int * policy)

Get the scheduling policy attribute from a thread attributes object.

A.10.2.7 int pthread_attr_init (pthread_attr_t * attr)

Initialize a thread attributes object with default values.

Thread attributes are used to specify parameters to `pthread_create()` (p.64). One attribute object can be used in multiple calls to `pthread_create()` (p.64), with or without modifications between calls.

The `pthread_attr_init()` (p.60) function initializes attr with all the default thread attributes.

Returns:

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

A.10.2.8 int pthread_attr_setbasichheapsize (pthread_attr_t * attr, int heapsize)

Set the initial heap size used for the thread.

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

sz: amount of memory needed for a variable

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

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

Returns:

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

A.10.2.9 int pthread_attr_setflags (pthread_attr_t * attr, int flags)

Set thread flags describing the thread's behaviour.

Possible values are:

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

Returns:

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

A.10.2.10 int pthread_attr_setiq (pthread_attr_t * attr, int iq)

Set the instruction quantum.

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

Returns:

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

A.10.2.11 int pthread_attr_setmemory (pthread_attr_t * attr, memory_t * mem)

Set the memory to use for the thread.

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

Returns:

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

A.10.2.12 int pthread_attr_setschedparam (pthread_attr_t * attr, sched_param * param)

Set the scheduling parameter attribute in a thread attributes object.

A.10.2.13 int pthread_attr_setschedpolicy (pthread_attr_t * attr, int policy)

Set the scheduling policy attribute in a thread attributes object.

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

- SCHED_RT: hard real-time thread
- SCHED_PIQ: soft real-time thread
- SCHED_RR: non-real-time thread.

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

Returns:

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

A.10.2.14 int pthread_cond_broadcast (pthread_cond_t * cond)

Unblock all threads currently blocked on the specified condition variable.

The `pthread_cond_broadcast()` (p. 62) function unblocks all threads waiting for the condition variable `cond`.

Returns:

If successful, the `pthread_cond_broadcast()` (p. 62) function will return zero, otherwise an error number will be returned to indicate the error.

A.10.2.15 int pthread_cond_destroy (pthread_cond_t * cond)

Destroy a condition variable.

The `pthread_cond_destroy()` (p. 63) function frees the resources allocated by the condition variable `cond`.

Returns:

If successful, the `pthread_cond_destroy()` (p. 63) function will return zero, otherwise an error number will be returned to indicate the error.

A.10.2.16 int pthread_cond_init (pthread_cond_t * cond, const pthread_condattr_t * attr)

Initialize a condition variable with specified attributes.

The `pthread_cond_init()` (p. 63) function creates a new condition variable, with attributes specified with `attr`. If `attr` is `NULL` the default attributes are used.

Returns:

If successful, the `pthread_cond_init()` (p. 63) function will return zero and put the new condition variable id into `cond`, otherwise an error number will be returned to indicate the error.

A.10.2.17 int pthread_cond_signal (pthread_cond_t * cond)

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

The `pthread_cond_signal()` (p. 63) function unblocks one thread waiting for the condition variable `cond`.

Returns:

If successful, the `pthread_cond_signal()` (p. 63) function will return zero, otherwise an error number will be returned to indicate the error.

A.10.2.18 int pthread_cond_wait (pthread_cond_t *, pthread_mutex_t * mutex)

Wait for a condition and lock the specified mutex.

The `pthread_cond_wait()` (p. 63) function atomically blocks the current thread waiting on the condition variable specified by `cond`, and releases the mutex specified by `mutex`. The waiting thread unblocks only after another thread calls `pthread_cond_signal()` (p. 63), or `pthread_cond_`

broadcast() (p. 62) with the same condition variable, and the current thread reacquires the lock on mutex.

Returns:

If successful, the **pthread_cond_wait()** (p. 63) function will return zero. Otherwise an error number will be returned to indicate the error.

A.10.2.19 int pthread_create (pthread_t * thread, const pthread_attr_t * attr, void (*)(void *) start_routine, void * arg)

Creates a new thread of execution.

The **pthread_create()** (p. 64) function is used to create a new thread, with attributes specified by *attr*, within a process. If the attributes specified by *attr* are modified later, the thread's attributes are not affected. Upon successful completion **pthread_create()** (p. 64) will store the ID of the created thread in the location specified by *thread*. The thread is created executing *start_routine* with *arg* as its sole argument.

Returns:

If successful, the **pthread_create()** (p. 64) function will return zero. Otherwise an error number will be returned to indicate the error.

A.10.2.20 int pthread_join (pthread_t thread, void ** value_ptr)

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

The **pthread_join()** (p. 64) function suspends execution of the calling thread until the target thread terminates unless the target thread has already terminated.

When a **pthread_join()** (p. 64) returns successfully, the target thread has been terminated. The results of multiple simultaneous calls to **pthread_join()** (p. 64) specifying the same target thread are undefined. If the thread calling **pthread_join()** (p. 64) is cancelled, then the target thread is not detached.

Returns:

If successful, the **pthread_join()** (p. 64) function will return zero. Otherwise an error number will be returned to indicate the error.

A.10.2.21 int pthread_mutex_destroy (pthread_mutex_t * mutex)

Destroy a mutex.

The **pthread_mutex_destroy()** (p. 64) function frees the resources allocated for mutex.

Returns:

If successful, **pthread_mutex_destroy()** (p. 64) will return zero, otherwise an error number will be returned to indicate the error.

A.10.2.22 int pthread_mutex_init (pthread_mutex_t * *mutex*, const pthread_mutexattr_t * *attr*)

Initialize a mutex with specified attributes.

The **pthread_mutex_init()** (p. 65) function creates a new mutex, with attributes specified with *attr*. If *attr* is NULL the default attributes are used.

Returns:

If successful, **pthread_mutex_init()** (p. 65) will return zero and put the new mutex id into *mutex*, otherwise an error number will be returned to indicate the error.

A.10.2.23 int pthread_mutex_lock (pthread_mutex_t * *mutex*)

Lock a mutex and block until it becomes available.

The **pthread_mutex_lock()** (p. 65) function locks *mutex*. If the mutex is already locked, the calling thread will block until the mutex becomes available.

Returns:

If successful, **pthread_mutex_lock()** (p. 65) will return zero, otherwise an error number will be returned to indicate the error.

A.10.2.24 int pthread_mutex_trylock (pthread_mutex_t * *mutex*)

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

The **pthread_mutex_trylock()** (p. 65) function locks *mutex*. If the mutex is already locked, **pthread_mutex_trylock()** (p. 65) will not block waiting for the mutex, but will return an error condition.

Returns:

If successful, **pthread_mutex_trylock()** (p. 65) will return zero, otherwise an error number will be returned to indicate the error.

A.10.2.25 int pthread_mutex_unlock (pthread_mutex_t * *mutex*)

Unlock a mutex.

If the current thread holds the lock on *mutex*, then the **pthread_mutex_unlock()** (p. 65) function unlocks *mutex*.

Returns:

If successful, **pthread_mutex_unlock()** (p. 65) will return zero, otherwise an error number will be returned to indicate the error.

A.10.2.26 pthread_t pthread_self (void)

Returns the thread ID of the calling thread.

Returns:

The `pthread_self()` (p. 65) function returns the thread ID of the calling thread.

A.10.2.27 void pthread_yield (void)

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

A.11 stropts.h File Reference

Defines

- #define **STROPTS_H_1**

Functions

- int **ioctl** (int d, int request,...)
Control a device.

A.11.1 Define Documentation

A.11.1.1 #define STROPTS_H_1

Definition at line 35 of file stropts.h.

A.11.2 Function Documentation

A.11.2.1 int ioctl (int d, int request, ...)

Control a device.

The **ioctl()** (p. 67) system call manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g. terminals) may be controlled with **ioctl()** (p. 67) requests. The argument d must be an open file descriptor.

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

Returns:

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

A.12 types.h File Reference

```
#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`
- typedef `int32_t` `error_t`

Enumerations

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

A.12.1 Define Documentation

A.12.1.1 `#define false FALSE`

Definition at line 68 of file types.h.

A.12.1.2 `#define true TRUE`

Definition at line 69 of file types.h.

A.12.1.3 `#define TYPES_H_ 1`

Definition at line 35 of file types.h.

A.12.2 Typedef Documentation

A.12.2.1 typedef char* address

Definition at line 71 of file types.h.

A.12.2.2 typedef int32_t error_t

Definition at line 78 of file types.h.

A.12.2.3 typedef signed short int int16_t

Definition at line 51 of file types.h.

A.12.2.4 typedef signed int int32_t

Definition at line 53 of file types.h.

A.12.2.5 typedef signed long long int64_t

Definition at line 55 of file types.h.

A.12.2.6 typedef signed char int8_t

Definition at line 49 of file types.h.

A.12.2.7 typedef unsigned short int uint16_t

Definition at line 50 of file types.h.

A.12.2.8 typedef unsigned int uint32_t

Definition at line 52 of file types.h.

A.12.2.9 typedef unsigned long long uint64_t

Definition at line 54 of file types.h.

A.12.2.10 typedef unsigned char uint8_t

Definition at line 48 of file types.h.

A.12.2.11 typedef uint32_t version_t

Definition at line 75 of file types.h.

A.12.3 Enumeration Type Documentation

A.12.3.1 enum bool_t

Enumerator:

FALSE

TRUE

Definition at line 67 of file types.h.

A.13 unistd.h File Reference

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

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

Defines

- `#define UNISTD_H_1`

Functions

- `int close (int d)`
Delete a descriptor.
- `ssize_t read (int d, void *buf, size_t nbytes)`
Read input.
- `ssize_t write (int d, const void *buf, size_t nbytes)`
Write output.

A.13.1 Define Documentation

A.13.1.1 `#define UNISTD_H_1`

Definition at line 34 of file unistd.h.

A.13.2 Function Documentation

A.13.2.1 `int close (int d)`

Delete a descriptor.

The `close()` (p. 71) system call deletes a descriptor from the per-process object reference table. If this is the last reference to the underlying object, the object will be deactivated.

Returns:

The `close()` (p. 71) function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

A.13.2.2 `ssize_t read (int d, void * buf, size_t nbytes)`

Read input.

The **read()** (p. 71) system call attempts to read *nbytes* of data from the object referenced by the descriptor *d* into the buffer pointed to by *buf*.

Returns:

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

A.13.2.3 `ssize_t write (int d, const void * buf, size_t nbytes)`

Write output.

The **write()** (p. 72) system call attempts to write *nbytes* of data to the object referenced by the descriptor *d* from the buffer pointed to by *buf*.

Returns:

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

References

- [1] AUTOSAR AUTomotive Open System ARchitecture. <http://www.autosar.org/>.
- [2] FAQ of comp.realtime. <http://www.faqs.org/faqs/realtime-computing/faq/>, July 1998. visited July 2007.
- [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), Springer, pp. 343–357.
- [4] KLUGE, F., AND WOLF, J. Basic System-Level Software for a Single-Core MERASA Processor. Tech. Rep. 2008-06, Department of Computer Science, University of Augsburg, Apr. 2008.
- [5] LEA, D. A Memory Allocator. *unix/mail* (Dec. 1996).
- [6] 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.
- [7] MISCHE, J., UHRIG, S., KLUGE, F., AND UNGERER, T. Exploiting Spare Resources of In-order SMT Processors Executing Hard Real-time Threads. In *IEEE International Conference on Computer Design 2008 (ICCD 08)* (Oct. 2008).
- [8] MISCHE, J., UHRIG, S., KLUGE, F., AND UNGERER, T. IPC Control for Multiple Real-Time Threads on an In-order SMT Processor. In *The 4th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC 2009)* (Paphos, Cyprus, Jan. 2009).
- [9] MORPHEW, G. Debugging complex embedded applications. <http://www.ddj.com/embedded/184406044>, April 2005. visited April 2008.
- [10] OSEK VDX Portal. <http://www.osek-vdx.org>. Visited April 2008.
- [11] IEEE Std 1003.1, 2004 Edition. The Open Group Base Specifications Issue 6, 2004.
- [12] QNX Software Systems. <http://www.qnx.com/>. Visited April 2008.
- [13] UHRIG, S., MAIER, S., AND UNGERER, T. Toward a Processor Core for Real-time Capable Autonomic Systems. In *Proceedings of the 5th IEEE International Symposium on Signal Processing and Information Technology* (Dec. 2005), pp. 19–22.