

libparanut

Generated by Doxygen 1.8.13

Contents

1	libparanut Documentation	1
1.1	Copyright	1
1.2	Modules of the libparanut	1
1.3	Overview	2
1.4	HOWTO	3
1.5	Expectations to the application	4
1.6	Future Ideas for libparanut	5
2	Todo List	7
3	Module Index	9
3.1	Modules	9
4	Class Index	11
4.1	Class List	11
5	File Index	13
5.1	File List	13

6	Module Documentation	15
6.1	libparanut Helpers	15
6.1.1	Detailed Description	15
6.2	Typedefs	16
6.2.1	Detailed Description	16
6.2.2	Typedef Documentation	16
6.2.2.1	_pn_spinlock	16
6.2.2.2	PN_CID	17
6.2.2.3	PN_CMSK	17
6.2.2.4	PN_NUMC	17
6.2.2.5	PN_NUMG	17
6.3	Error Codes	18
6.3.1	Detailed Description	18
6.3.2	Macro Definition Documentation	18
6.3.2.1	PN_ERR_CACHE_LINESIZE	18
6.3.2.2	PN_ERR_COPU	19
6.3.2.3	PN_ERR_EXC	19
6.3.2.4	PN_ERR_LOCKOCC	19
6.3.2.5	PN_ERR_MATCH	19
6.3.2.6	PN_ERR_NOIMP	19
6.3.2.7	PN_ERR_PARAM	19
6.3.2.8	PN_SUCCESS	19
6.4	Modes	20
6.4.1	Detailed Description	20
6.5	libparanut Modules	21
6.5.1	Detailed Description	21
6.6	Base Module	22
6.6.1	Detailed Description	22
6.6.2	Function Documentation	23
6.6.2.1	pn_coreid()	23

6.6.2.2	pn_coreid_g()	23
6.6.2.3	pn_halt()	24
6.6.2.4	pn_halt_CoPU()	24
6.6.2.5	pn_halt_CoPU_gm()	24
6.6.2.6	pn_halt_CoPU_m()	25
6.6.2.7	pn_m2cap()	26
6.6.2.8	pn_m2cap_g()	26
6.6.2.9	pn_m3cap()	27
6.6.2.10	pn_m3cap_g()	27
6.6.2.11	pn_numcores()	28
6.6.2.12	pn_simulation()	28
6.6.2.13	pn_time_ns()	28
6.7	Link Module	29
6.7.1	Detailed Description	29
6.7.2	Function Documentation	29
6.7.2.1	pn_begin_linked()	30
6.7.2.2	pn_begin_linked_gm()	30
6.7.2.3	pn_begin_linked_m()	31
6.7.2.4	pn_end_linked()	32
6.8	Thread Module	33
6.8.1	Detailed Description	33
6.8.2	Function Documentation	33
6.8.2.1	pn_begin_threaded()	33
6.8.2.2	pn_begin_threaded_gm()	34
6.8.2.3	pn_begin_threaded_m()	35
6.8.2.4	pn_end_threaded()	36
6.8.2.5	pn_thread_entry()	36
6.9	Cache Module	37
6.9.1	Detailed Description	37
6.9.2	Function Documentation	37

6.9.2.1	pn_cache_disable()	38
6.9.2.2	pn_cache_enable()	38
6.9.2.3	pn_cache_flush()	38
6.9.2.4	pn_cache_flush_all()	39
6.9.2.5	pn_cache_init()	39
6.9.2.6	pn_cache_invalidate()	39
6.9.2.7	pn_cache_invalidate_all()	40
6.9.2.8	pn_cache_linesize()	40
6.9.2.9	pn_cache_size()	40
6.9.2.10	pn_cache_writeback()	40
6.9.2.11	pn_cache_writeback_all()	41
6.10	Exception Module	42
6.10.1	Detailed Description	42
6.10.2	Function Documentation	42
6.10.2.1	pn_ecall()	42
6.10.2.2	pn_exception_set_handler()	43
6.10.2.3	pn_interrupt_disable()	43
6.10.2.4	pn_interrupt_enable()	43
6.10.2.5	pn_progress_mepc()	44
6.11	Spinlock Module	45
6.11.1	Detailed Description	45
6.11.2	Function Documentation	46
6.11.2.1	pn_spinlock_destroy()	46
6.11.2.2	pn_spinlock_init()	46
6.11.2.3	pn_spinlock_lock()	47
6.11.2.4	pn_spinlock_trylock()	47
6.11.2.5	pn_spinlock_unlock()	48
6.12	libparanut Compile Time Parameters	49
6.12.1	Detailed Description	49
6.12.2	Macro Definition Documentation	49
6.12.2.1	PN_CACHE_LINESIZE	49
6.12.2.2	PN_COMPILE_RAW	50
6.12.2.3	PN_RWIDTH	50
6.13	Comm_def	51
6.13.1	Detailed Description	51
6.14	Comm_glo	52
6.14.1	Detailed Description	52

7	Class Documentation	53
7.1	__pn_spinlock Struct Reference	53
7.1.1	Detailed Description	53
8	File Documentation	55
8.1	common/common.h File Reference	55
8.1.1	Detailed Description	56
8.2	libparanut.h File Reference	56
8.2.1	Detailed Description	60
8.3	Makefile File Reference	60
8.3.1	Detailed Description	60
8.4	pn_base/pn_base.c File Reference	63
8.4.1	Detailed Description	65
8.5	pn_cache/pn_cache.c File Reference	65
8.5.1	Detailed Description	66
8.6	pn_cache/pn_cache_RV32I_buildscript.py File Reference	66
8.6.1	Detailed Description	66
8.7	pn_config.h File Reference	66
8.7.1	Detailed Description	67
8.8	pn_exception/pn_exception.c File Reference	67
8.8.1	Detailed Description	68
8.9	pn_link/pn_link.c File Reference	68
8.9.1	Detailed Description	69
8.10	pn_spinlock/pn_spinlock.c File Reference	69
8.10.1	Detailed Description	70
8.11	pn_thread/pn_thread.c File Reference	70
8.11.1	Detailed Description	71
	Index	73

Chapter 1

libparanut Documentation

1.1 Copyright

Copyright 2019-2020 Anna Pfuetzner (annakerstin.pfuetzner@gmail.com)

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

1.2 Modules of the libparanut

Welcome to the libparanut documentation! This is a hardware abstraction library you can use to program your ParaNut without having to stare at Assembly Code.

To do that, the libparanut serves several functions to you on a silver platter. These are grouped into [libparanut Modules](#) for your convenience.

1. The [Base Module](#) contains functions for getting general information about the ParaNut, like the number of cores and such. There's also some functions for halting cores.

2. The [Link Module](#) contains functions for stopping and starting Linked Mode execution. Don't know what that is? Check [Modes](#) or the fine ParaNut Manual.
3. The [Thread Module](#) contains functions for stopping and starting Threaded Mode execution. Same advice goes.
4. The [Cache Module](#) contains functions for controlling the ParaNut cache and getting information about it.
5. The [Exception Module](#) contains functions for controlling Interrupts and Exceptions.
6. The [Spinlock Module](#) is an implementation of a spinlock so you can properly protect and synchronize your stuff in Threaded Mode.

1.3 Overview

The following picture shows a system overview. It might be a little confusing at first. Don't worry about that, just read some more documentation and it will probably all fall into place.

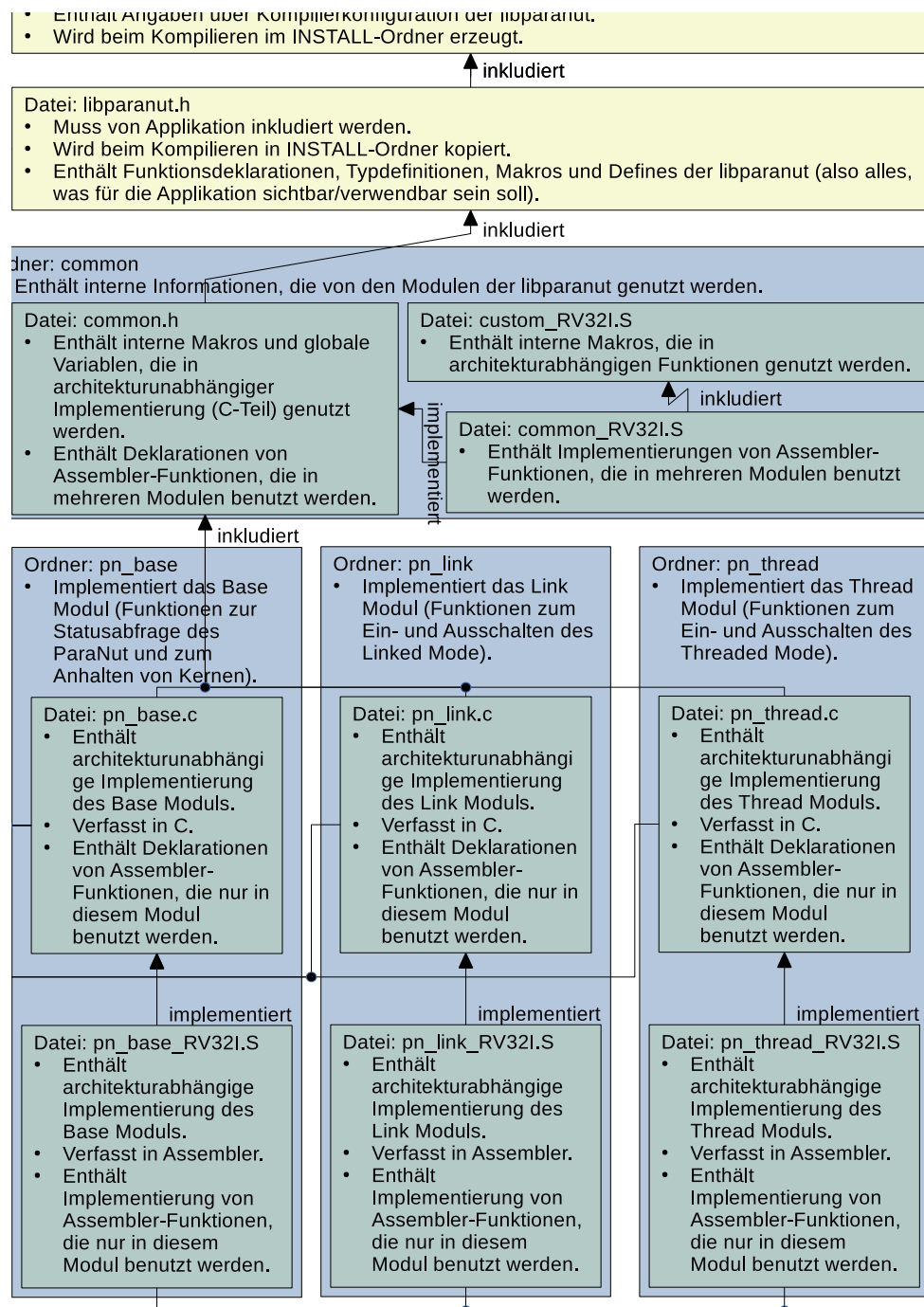


Figure 1.1 Overview of the libparanut

1.4 HOWTO

Todo Add documentation here in case other compilers/ISAs are used someday.

Todo The [Makefile](#) might not be compliant with any other tool than GNU make.

How to make libparanut run with our in-house ParaNut implementation: You will need GNU make and the RISC-V GNU Toolchain for that (see Appendix A in ParaNut Manual). Also, you may need to have Python on your system (so far, no restrictions on the version are known). This assumes you already got the ParaNut up and running!

1. Open up the wonderful [Makefile](#) in the libparanut directory. This is the place where all your wishes come true. Right at the start, there's the section "System Configuration" where you can set a few things to tweak your libparanut. Check out the [libparanut Compile Time Parameters](#) to see what those values mean. The default should be fine at first.
2. Onto the section "Compiler Configuration". See the "PN_ISA" in the [Makefile](#)? This is the Instruction Set Architecture. The very nice [Makefile](#) tree will chose all the right assembly source code files for the libparanut if this variable is set correctly. Currently, our ParaNut implements RISC-V ISA, which is why this value says RV32I. This means you don't have to touch it.
3. The [Makefile](#) also lets you chose the compiler. If you want another compiler than GCC, you will also need to add some if-else logic in the section "Compiler and Assembler Flags".
4. To reduce the code size of the libparanut, you could set "PN_DEBUG" to 0. This means that the libparanut will not be compiled with debug symbols. It should be fine to compile it with debug symbols at first, though.
5. Want everything compiled now? Say

```
gmake all
```

and the magic is done. All the modules are compiled and statically linked into "libparanut.a". You can find it in the newly generated directory named "INSTALL". Do not forget to call clean beforehand if you changed something in the Makefile.

6. Include [libparanut.h](#) in your application. Set your include-path to the INSTALL directory while compiling:

```
-I/path/to/libparanut/INSTALL
```

7. Link libparanut.a into your application. To do that in GCC, you can just put

```
-L/path/to/libparanut/INSTALL -lparanut
```

at the end of your linker/compilation command.

I hope the following example can explain what I mean:

Todo Prettier example, explain Link Module and Spinlock Module.

```
cd /path/to/libparanut
gmake all
cd /path/to/my_application.c
my_gcc -c my_compiler_options -I/path/to/libparanut/INSTALL my_application.c
my_gcc my_link_options my_object.o -L/path/to/libparanut/INSTALL -lparanut
```

Do not forget to also compile in the startup code. To see a full example of what you need to do, check the Makefiles in the other sw/ subdirectories in the GIT.

1.5 Expectations to the application

Here's a list of things that the libparanut expects you to have been done before usage:

1. The [Cache Module](#) expects you to have called [pn_cache_init\(\)](#) before using any of its other functions. Behaviour of the other functions is undefined if not.

2. The [Exception Module](#) expects you to have called `pn_exception_init()` before calling `pn_exception_set_handler()`. I mean, you can set your own exception handler beforehand, but it will not be called if the exception occurs. The `pn_interrupt_enable()`, `pn_interrupt_disable()`, and `pn_progress_mepc()` functions will work with no problem, though.
3. The [Thread Module](#) is the complicated one. It's a little tricky because of its workflow. The CePU internally sets a few needed variables which indicate a status to the CoPU. After a CoPU is woken up, they start executing code at the reset address. This means that the end of the startup code for the CoPUs also needs to be their entrance point to the [Thread Module](#), where they can read the internal variables and figure out what they are supposed to do with them. This entrance point is `pn_thread_entry()`, and it needs to be called in the startup code on all CoPUs. Our in-house startup code does this for you, but I'm leaving that here just in case you want to know.
4. The [Thread Module](#) has one more expectation. The state of the CePU and its memory need to be shared with all cores. That means that there has to be a shared memory section that has to be as big as the memory for the individual cores. The start address of this area has to be put into a globally known location called `shared_mem_start`, and the size has to be put into `shared_mem_size`. This also happens in our build-in startup code, so you do not need to worry about it.

Also, here's some general advice: If you want your startup code to be compatible with many versions of the libparanut, you can check if a certain module is available. The file `pn_config.h`, which is created during compilation time, has some defines set for every module that is enabled. Check out the `pn_config.h` documentation to find out more!

1.6 Future Ideas for libparanut

1. In the [Cache Module](#), split `pn_cache_enable()` and `pn_cache_disable()` into separate functions for data and instruction cache.
2. In the [Thread Module](#), build in POSIX style thread functions.
3. Build a state machine for every core that tracks whether a core is used in linked mode, threaded mode, or doing POSIX threads. Enable the user of the libparanut to use threaded and linked mode in a mixed way.
4. In [Exception Module](#), provide a function to hang in a "raw" exception handler (basically change mtvec).
5. Implement a possibility to check whether or not a module has been initialized already. Also implement a `pn_init()` function which checks for all modules if the module is already initialized, and if it is not, initializes them (provided the module was compiled in).
6. Concerning the `init()` functions: If a function in a module is used despite of the module not being initialized, throw an exception (use `pn_ecall()`).
7. Implement a `pn_strerror()` function which takes an error value and prints what the error means.
8. Implement a global checkable error variable.
9. For threaded and linked mode, implement the possibility to pass a stack size that shall be copied. It should also be able to tell these functions that only the stack from the top function shall be copied.
10. Implement a `pn_atomic_increment()` and `pn_atomic_decrement()` function.
11. Implement `pn_printf()` which can be used in linked mode.
12. In the Makefile, copy the libparanut to the place where the RISC-V toolchain is installed. This would make it easier for applications to compile.

Chapter 2

Todo List

page [libparanut Documentation](#)

Add English Version and fix format

Add documentation here in case other compilers/ISAs are used someday.

The [Makefile](#) might not be compliant with any other tool than GNU make.

Prettier example, explain Link Module and Spinlock Module.

Member [pn_begin_linked](#) (PN_NUMC numcores)

Not yet implemented for more cores than what is available in one group. Will return [PN_ERR_NOIMP](#) when called with more cores.

Member [pn_begin_linked_gm](#) (PN_CMSK *coremask_array, PN_NUMG array_size)

Currently only a stub. Will therefore always return [PN_ERR_NOIMP](#).

Member [pn_begin_threaded](#) (PN_NUMC numcores)

Not yet implemented for more cores than what is available in one group. Will return [PN_ERR_NOIMP](#) when called with more cores.

Member [pn_begin_threaded_gm](#) (PN_CMSK *coremask_array, PN_NUMG array_size)

Currently only a stub. Will therefore always return [PN_ERR_NOIMP](#).

Member [pn_coreid_g](#) (PN_NUMG *groupnum)

Currently only a stub. Will therefore always return [PN_ERR_NOIMP](#).

File [pn_exception.c](#)

Layer this better in later versions of libparanut.

Member [pn_halt_CoPU](#) (PN_CID coreid)

Not yet implemented for given core IDs outside of group 0. Will return [PN_ERR_NOIMP](#) in this case.

Member [pn_halt_CoPU_gm](#) (PN_CMSK *coremask_array, PN_NUMG array_size)

Currently only a stub. Will therefore always return either [PN_ERR_COPU](#) or [PN_ERR_NOIMP](#) if executed on CePU.

Member [pn_m2cap_g](#) (PN_NUMG groupnum)

Currently only a stub. Will therefore always return either [PN_ERR_COPU](#) or [PN_ERR_NOIMP](#) if executed on CePU.

Member [pn_m3cap](#) (void)

If other cores are ever capable of Mode 3 (and if there ever is a register to get the information from), implement this properly.

Member [pn_m3cap_g](#) (PN_NUMG groupnum)

Currently only a stub. Will therefore always return either [PN_ERR_COPU](#) or [PN_ERR_NOIMP](#) if executed on CePU.

Chapter 3

Module Index

3.1 Modules

Here is a list of all modules:

libparanut Helpers	15
Typedefs	16
Error Codes	18
Modes	20
libparanut Modules	21
Base Module	22
Link Module	29
Thread Module	33
Cache Module	37
Exception Module	42
Spinlock Module	45
libparanut Compile Time Parameters	49
Comm_def	51
Comm_glo	52

Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

__pn_spinlock	A synchronization primitive. Use _pn_spinlock instead of this	53
-------------------------------	---	----

Chapter 5

File Index

5.1 File List

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

libparanut.h	API of the libparanut	56
Makefile	Makefile of the libparanut	60
pn_config.h	See Documentaion of libparanut Compile Time Parameters	66
common/common.h	Contains architecture independent internal prototypes and defines needed in libparanut Modules	55
pn_base/pn_base.c	Contains architecture independent implementations of the Base Module functions	63
pn_cache/pn_cache.c	Contains architecture independent implementations of the Cache Module functions	65
pn_cache/pn_cache_RV32I_buildscript.py	Builds the architecture dependent assembly files of the Cache Module since they are performance critical, very dependent on cache line size, and should not take up to much space	66
pn_exception/pn_exception.c	Contains (somewhat) architecture independent implementations of the Exception Module functions	67
pn_link/pn_link.c	Contains architecture independent implementations of the Link Module functions	68
pn_spinlock/pn_spinlock.c	Contains architecture independent implementations of the Spinlock Module functions	69
pn_thread/pn_thread.c	Contains architecture independent implementations of the Thread Module functions	70

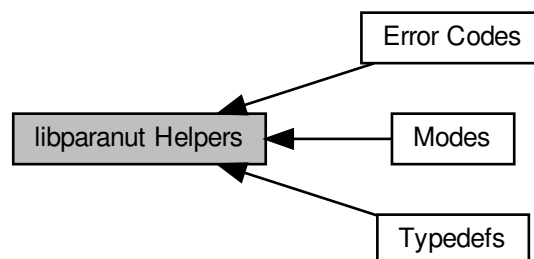
Chapter 6

Module Documentation

6.1 libparanut Helpers

Typedefs and defines of libparanut which can also be used in your application.

Collaboration diagram for libparanut Helpers:



Modules

- [Typedefs](#)
Needed typedefs.
- [Error Codes](#)
Error codes returned by the functions in this library.
- [Modes](#)
Modes of the ParaNut Cores.

6.1.1 Detailed Description

Typedefs and defines of libparanut which can also be used in your application.

6.2 Typedefs

Needed typedefs.

Collaboration diagram for Typedefs:



Typedefs

- typedef struct [__pn_spinlock](#) [__pn_spinlock](#)
Renaming of struct [__pn_spinlock](#) for your convenience.
- typedef int32_t [PN_CID](#)
Signed type that can be used to address any core in this architecture.
- typedef int32_t [PN_NUMC](#)
Signed type that can hold the maximum number of cores in this architecture.
- typedef uint32_t [PN_CMSK](#)
Unsigned type that can act as a core mask.
- typedef int32_t [PN_NUMG](#)
Signed type that can be used to address any group in this architecture.

6.2.1 Detailed Description

Needed typedefs.

Please note that there are several typedefs for differing register widths of the ParaNut. Which ones are used depend on how you set [PN_RWIDTH](#) while compiling ([pn_config.h](#)). Doxygen can only document one of them, so here you are, stuck with the 32-bit version. Check source code of [libparanut.h](#), section Typedefs, if you want to see the others.

6.2.2 Typedef Documentation

6.2.2.1 [__pn_spinlock](#)

[__pn_spinlock](#)

Renaming of struct [__pn_spinlock](#) for your convenience.

Check documentation of [__pn_spinlock](#) to get more information.

6.2.2.2 PN_CID

[PN_CID](#)

Signed type that can be used to address any core in this architecture.

See documentation of [PN_NUMC](#) to understand why we use only the actual register width and not more.

6.2.2.3 PN_CMSK

[PN_CMSK](#)

Unsigned type that can act as a core mask.

This is, of course, the same as the register width of the ParaNut. Unsigned because it directly represents a register.

6.2.2.4 PN_NUMC

[PN_NUMC](#)

Signed type that can hold the maximum number of cores in this architecture.

Let's say your ParaNut has a group register width of 32 bits. This means that there are 4.294.967.296 potential groups. Every group has 32 bits to represent different cores. That means there are 137.438.953.472 cores that can be addressed.

This does, in theory, mean that we need 64 bit to represent the possible number of cores. However, it is deemed to be pretty unrealistic that there will be a ParaNut version with more than 4.294.967.296 cores anytime soon. So, for optimization purposes, we just use 32 bit here. Even half of that is probably not possible in my lifetime, which is why we are not even using unsigned (also because some compilers could throw errors when mixing signed and unsigned types, e.g. in a for loop). One more plus is that we can use these values to signal errors when they are returned by a function.

Same explanation goes in all other register widths. If you really need more, feel free to double the bits.

6.2.2.5 PN_NUMG

[PN_NUMG](#)

Signed type that can be used to address any group in this architecture.

This is, of course, the same as the register width of the ParaNut.

6.3 Error Codes

Error codes returned by the functions in this library.

Collaboration diagram for Error Codes:



- `#define PN_SUCCESS 0`
Successful execution.
- `#define PN_ERR_PARAM (-1)`
Parameter error.
- `#define PN_ERR_NOIMP (-2)`
Function not implemented.
- `#define PN_ERR_COPU (-3)`
CoPU error.
- `#define PN_ERR_MATCH (-4)`
Mode begin and end matching error.
- `#define PN_ERR_LOCKOCC (-5)`
Lock occupied error.
- `#define PN_ERR_CACHE_LINESIZE (-6)`
Weird cache line size error.
- `#define PN_ERR_EXC (-8)`
Exception code not implemented error.

6.3.1 Detailed Description

Error codes returned by the functions in this library.

6.3.2 Macro Definition Documentation

6.3.2.1 PN_ERR_CACHE_LINESIZE

```
#define PN_ERR_CACHE_LINESIZE (-6)
```

Weird cache line size error.

Can occur if libparanut is supposed to run on an architecture that has a cache line size which is not either 32, 64, 128, 256, 512, 1024 or 2048 bit. Should be more of a development error instead of a normal usage error.

In other words, it should not occur if you are just developing middle end stuff while using the libparanut on a deployed ParaNut. Contact the maintainers about this if it still does.

6.3.2.2 PN_ERR_COPU

```
#define PN_ERR_COPU (-3)
```

CoPU error.

Function that isn't allowed on CoPU was being executed on CoPU.

6.3.2.3 PN_ERR_EXC

```
#define PN_ERR_EXC (-8)
```

Exception code not implemented error.

You tried calling [pn_exception_set_handler\(\)](#) with an invalid exception code.

6.3.2.4 PN_ERR_LOCKOCC

```
#define PN_ERR_LOCKOCC (-5)
```

Lock occupied error.

Can occur if you tried destroying an occupied lock or used the trylock function on an occupied lock.

6.3.2.5 PN_ERR_MATCH

```
#define PN_ERR_MATCH (-4)
```

Mode begin and end matching error.

Functions for beginning and ending linked and threaded mode have to be matched. Linked and threaded mode shall not be mixed.

6.3.2.6 PN_ERR_NOIMP

```
#define PN_ERR_NOIMP (-2)
```

Function not implemented.

The libparanut function is not yet implemented.

6.3.2.7 PN_ERR_PARAM

```
#define PN_ERR_PARAM (-1)
```

Parameter error.

The parameters given to a function were wrong (i.e. out of range).

6.3.2.8 PN_SUCCESS

```
#define PN_SUCCESS 0
```

Successful execution.

Implies that a function finished successfully.

6.4 Modes

Modes of the ParaNut Cores.

Collaboration diagram for Modes:



- `#define PN_M0 0x0U`
Mode 0 (halted Mode).
- `#define PN_M1 0x1U`
Mode 1 (linked Mode).
- `#define PN_M2 0x2U`
Mode 2 (unlinked or threaded Mode).
- `#define PN_M3 0x3U`
Mode 3 (autonomous Mode).

6.4.1 Detailed Description

Modes of the ParaNut Cores.

The CePU can only ever operate in Mode 3 (autonomous). It is still shown as capable of Mode 2 (threaded Mode) because Mode 3 is an extension in functionality in comparison to Mode 2. Mode 2 cores do not handle their own interrupts/exceptions, which a Mode 3 core does.

It can also be set into Mode 0, which does not break hardware debugging support.

The CePU is the only core capable of changing other cores Modes.

The CoPUs are never capable of Mode 3. They may be capable of Mode 2, which means they are able to fetch their own instructions and are therefore able to do different work in parallel to the CePU. They are, at minimum, capable of Mode 1 (linked Mode), which means it will execute the same instructions as the CePU on different data. This does not start until the CePU is also told to now start executing in linked mode, though.

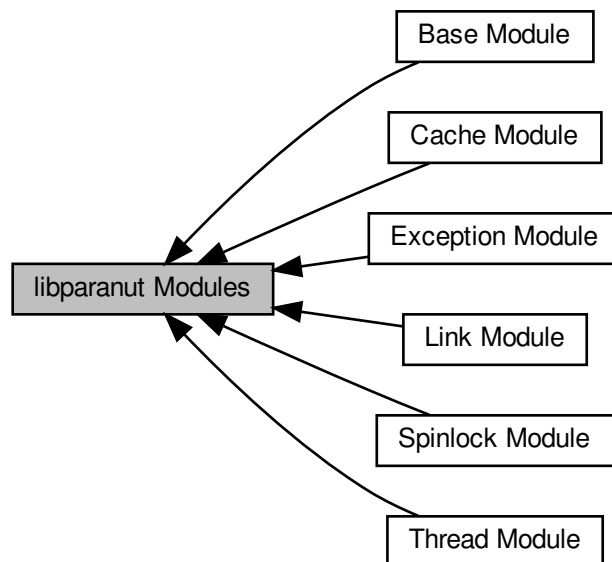
Which Mode the CoPUs are in after system reset is an implementation detail of the ParaNut itself and the startup code.

For further information, check ParaNut Manual.

6.5 libparanut Modules

Modules of libparanut.

Collaboration diagram for libparanut Modules:



Modules

- [Base Module](#)
Functions for getting the status of your ParaNut and halting cores.
- [Link Module](#)
Functions for using the linked mode.
- [Thread Module](#)
Functions for using the threaded mode.
- [Cache Module](#)
Special functions for controlling the shared ParaNut cache.
- [Exception Module](#)
Functions for controlling the handling of interrupts/exceptions.
- [Spinlock Module](#)
Functions and structure used for synchronizing memory access.

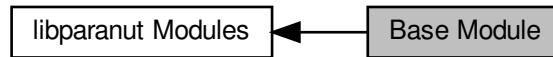
6.5.1 Detailed Description

Modules of libparanut.

6.6 Base Module

Functions for getting the status of your ParaNut and halting cores.

Collaboration diagram for Base Module:



- [PN_NUMC pn_numcores](#) (void)
Get the number of cores in your system.
- [PN_CMSK pn_m2cap](#) (void)
Check which cores are capable of Mode 2 operation.
- [PN_CMSK pn_m2cap_g](#) (PN_NUMG groupnum)
Check which cores are capable of Mode 2 operation.
- [PN_CMSK pn_m3cap](#) (void)
Check which cores are capable of Mode 3 operation.
- [PN_CMSK pn_m3cap_g](#) (PN_NUMG groupnum)
Check which cores are capable of Mode 3 operation.
- [PN_CID pn_coreid](#) (void)
Get the ID of the core that this function is executed on.
- [PN_CID pn_coreid_g](#) (PN_NUMG *groupnum)
Get the ID and group number of the core that this code is running on.
- void [pn_halt](#) (void)
Halt whatever core the function is executed on.
- int [pn_halt_CoPU](#) (PN_CID coreid)
Halts a CoPU.
- int [pn_halt_CoPU_m](#) (PN_CMSK coremask)
Halts one or more CoPUs.
- int [pn_halt_CoPU_gm](#) (PN_CMSK *coremask_array, PN_NUMG array_size)
Halts the CoPUs specified in the coremask_array.
- long long int [pn_time_ns](#) (void)
Returns system time in ns. Does not care for overflow.
- int [pn_simulation](#) (void)
Checks if we run in simulation instead of real hardware.

6.6.1 Detailed Description

Functions for getting the status of your ParaNut and halting cores.

Concerning the `_g` functions: If your ParaNut implementation has more cores than what is the standard register size on your system (i.e. register size is 32, but you got more than 32 cores including the CePU), you have to chose the group of cores that you want to talk to. For that, most functions in this section have a group version (suffix `_g`) which has an additional group parameter.

This works the following way: If your register size is 32, then the CePU is in group 0 (0x00000000) and is always represented by the first bit in the core masks (0x00000001). The first CoPU is also in group 0 and represented by the second bit (0x00000002). After that comes the second CoPU, represented by 0x00000004. And so on until CoPU 30. The 31st CoPU is then represented by group 1 (0x00000001) and the first bit (0x00000001). The 63rd CoPU is represented by group 2 (0x00000002) and the first bit (0x00000001). The 95th CoPU is represented by group 3 (0x00000003) and the first bit (0x00000001). This information may become untrue if a big-endian version of the ParaNut is ever released.

For further information on this topic, you should check the ParaNut Manual itself. Also, the documentation on the ParaNut [Modes](#) that is included in here could clear some things up.

6.6.2 Function Documentation

6.6.2.1 pn_coreid()

```
PN_CID pn_coreid (
    void )
```

Get the ID of the core that this function is executed on.

Returns

The core ID. Starts with 0 for CePU. Can not return an error.

6.6.2.2 pn_coreid_g()

```
PN_CID pn_coreid_g (
    PN_NUMG * groupnum )
```

Get the ID and group number of the core that this code is running on.

Todo Currently only a stub. Will therefore always return [PN_ERR_NOIMP](#).

Parameters

out	<i>groupnum</i>	is a reference to be filled with the group number of the core.
-----	-----------------	--

Returns

The core ID. Starts with 0 for CePU. Does not start again when in another group than group 0. Can not return an error.

Todo Group function implementation.

6.6.2.3 pn_halt()

```
void pn_halt (
    void )
```

Halt whatever core the function is executed on.

If executed on a core, it will be set to Mode 0 and stop operation. Causes reset of program counter to reset address on a Mode 2 capable CPU.

If executed on CePU, also halts all other CoPUs on system.

See documentation of [Modes](#) for more information.

This function returns nothing because it should not be possible for any core to leave this state on its own.

6.6.2.4 pn_halt_CoPU()

```
int pn_halt_CoPU (
    PN_CID coreid )
```

Halts a CoPU.

Sets the CoPU with the given ID into a halted state (Mode 0).

See documentation of [Modes](#) for more information.

Cannot be executed on CoPU.

Halting an already halted core results in failure ([PN_ERR_PARAM](#)) to aid debugging.

Todo Not yet implemented for given core IDs outside of group 0. Will return [PN_ERR_NOIMP](#) in this case.

Parameters

in	<i>coreid</i>	is the ID of the CoPUs you want to halt. Since ID 0 represents the CePU, this function will throw an error when given 0 to aid debugging. If you want the CePU to halt, use function pn_halt() .
----	---------------	--

Returns

Either [PN_SUCCESS](#), [PN_ERR_PARAM](#) or [PN_ERR_COPU](#).

6.6.2.5 pn_halt_CoPU_gm()

```
int pn_halt_CoPU_gm (
    PN_CMSK * coremask_array,
    PN_NUMG array_size )
```

Halts the CoPUs specified in the `coremask_array`.

Todo Currently only a stub. Will therefore always return either [PN_ERR_COPU](#) or [PN_ERR_NOIMP](#) if executed on CePU.

Sets the CoPUs represented by bitmask and their position in the array (= their group number) into a halted state (Mode 0).

See documentation of [Modes](#) for more information.

Cannot be executed on CoPU.

Halting an already halted core results in failure ([PN_ERR_PARAM](#)) to aid debugging.

Parameters

in	<i>coremask_array</i>	is a pointer to the start of a coremask array. The position of the mask in the array represents the group number of the cores. When all bits are set to 0, this function will return an error (PN_ERR_PARAM) to aid debugging. Also, setting the first bit to 1 will return an error (PN_ERR_PARAM) since it represents the CePU, which should be halted with pn_halt() .
in	<i>array_size</i>	is the number of entries in the coremask_array.

Returns

Either [PN_SUCCESS](#), [PN_ERR_PARAM](#) or [PN_ERR_COPU](#).

6.6.2.6 pn_halt_CoPU_m()

```
int pn_halt_CoPU_m (
    PN\_CMSK coremask )
```

Halts one or more CoPUs.

Sets the CoPUs represented in the bitmask into a halted state (Mode 0).

See documentation of [Modes](#) for more information.

Cannot be executed on CoPU.

Halting an already halted core results in failure ([PN_ERR_PARAM](#)) to aid debugging.

Parameters

in	<i>coremask</i>	is the bitmask representing the CoPUs you want to halt. When all bits are set to 0, this function will return an error (PN_ERR_PARAM) to aid debugging. Also, setting the first bit to 1 will return an error (PN_ERR_PARAM) since it represents the CePU, which should be halted with pn_halt() .
----	-----------------	--

Returns

Either [PN_SUCCESS](#), [PN_ERR_PARAM](#) or [PN_ERR_COPU](#).

6.6.2.7 pn_m2cap()

```
PN_CMSK pn_m2cap (
    void )
```

Check which cores are capable of Mode 2 operation.

See documentation of [Modes](#) for more information.

Cannot be called from CoPU.

Returns

A bitmask representing your ParaNut cores or [PN_ERR_COPU](#). If a bit is set to 1, it means that the core is capable of operating in Mode 2.

6.6.2.8 pn_m2cap_g()

```
PN_CMSK pn_m2cap_g (
    PN_NUMG groupnum )
```

Check which cores are capable of Mode 2 operation.

Todo Currently only a stub. Will therefore always return either [PN_ERR_COPU](#) or [PN_ERR_NOIMP](#) if executed on CePU.

See documentation of [Modes](#) for more information.

Cannot be called from CoPU.

Parameters

in	<i>groupnum</i>	is the group of cores you want to know about.
----	-----------------	---

Returns

A bitmask representing your ParaNut cores or [PN_ERR_COPU](#). If a bit is set to 1, it means that the core is capable of operating in Mode 2.

Todo Group function implementation.

6.6.2.9 pn_m3cap()

```
PN_CMSK pn_m3cap (
    void )
```

Check which cores are capable of Mode 3 operation.

Attention

This function will, in the current ParaNut implementation, return a hard coded 1 or [PN_ERR_COPU](#) if executed on CoPU. The reason for this is that only the CePU is capable of Mode 3.

See documentation of [Modes](#) for more information.

Cannot be called from CoPU.

Returns

A bitmask representing your ParaNut cores or [PN_ERR_COPU](#). If a bit is set to 1, it means that the core is capable of operating in Mode 3.

Todo If other cores are ever capable of Mode 3 (and if there ever is a register to get the information from), implement this properly.

6.6.2.10 pn_m3cap_g()

```
PN_CMSK pn_m3cap_g (
    PN_NUMG groupnum )
```

Check which cores are capable of Mode 3 operation.

Todo Currently only a stub. Will therefore always return either [PN_ERR_COPU](#) or [PN_ERR_NOIMP](#) if executed on CePU.

See documentation of [Modes](#) for more information.

Cannot be called from CoPU.

Parameters

in	<i>groupnum</i>	is the group of cores you want to know about.
----	-----------------	---

Returns

A bitmask representing your ParaNut cores or [PN_ERR_COPU](#). If a bit is set to 1, it means that the core is capable of operating in Mode 3.

Todo Group function implementation.

6.6.2.11 pn_numcores()

```
PN_NUMC pn_numcores (  
    void )
```

Get the number of cores in your system.

Cannot be called from CoPU.

Returns

The number of cores in your ParaNut implementation or [PN_ERR_COPU](#).

6.6.2.12 pn_simulation()

```
int pn_simulation (  
    void )
```

Checks if we run in simulation instead of real hardware.

Warning

This function is a thing that only works with our specific ParaNut simulation. If the simulation changes, this needs to be changed, too.

Returns

Zero if we run on hardware, non-zero if we run in simulation.

6.6.2.13 pn_time_ns()

```
long long int pn_time_ns (  
    void )
```

Returns system time in ns. Does not care for overflow.

Cannot be executed on CoPU.

The first time executing this function takes the longest time since it has to initialize the frequency and an internal conversion factor. So if you want to use it for time measurement, you can call the function once before actual measurement to make the values more comparable.

When testing on my ParaNut, this made a difference of around 2000 ticks.

Returns

System time in ns or [PN_ERR_COPU](#).

6.7 Link Module

Functions for using the linked mode.

Collaboration diagram for Link Module:



- [PN_CID pn_begin_linked](#) ([PN_NUMC](#) numcores)
Links a given number of CoPUs to the CePU.
- [PN_CID pn_begin_linked_m](#) ([PN_CMSK](#) coremask)
Links the CPUs specified in the coremask.
- [PN_CID pn_begin_linked_gm](#) ([PN_CMSK](#) *coremask_array, [PN_NUMG](#) array_size)
Links the CPUs specified in the coremask_array.
- `int pn_end_linked` (void)
Ends linked execution.

6.7.1 Detailed Description

Functions for using the linked mode.

Also see [Modes](#).

Warning

If you want to use the Linked Module on RISC-V ParaNut, your ParaNut has to support the M Extension and you have to compile your application with the flag `mabi=rv32im`. The [libparanut Makefile](#) sets this flag automatically when you chose the [Link Module](#) or one of the Modules that has [Link Module](#) as a dependency.

6.7.2 Function Documentation

6.7.2.1 `pn_begin_linked()`

```
PN_CID pn_begin_linked (
    PN_NUMC numcores )
```

Links a given number of CoPUs to the CePU.

Todo Not yet implemented for more cores than what is available in one group. Will return `PN_ERR_NOIMP` when called with more cores.

Sets numcores-1 CoPUs to Mode 1 (linked Mode) so they start executing the instruction stream fetched by the CePU. This function will return an error (`PN_ERR_MATCH`) if the CoPUs are not all halted.

See documentation of [Modes](#) for more information.

Cannot be executed on CoPU. Cannot be mixed with threaded mode or other linked mode functions, until `pn_end_linked()` is called.

Attention

All data that you want to preserve after `pn_end_linked()` was called can not be stored on stack. You can make it static or global.

Warning

There is no guarantee that the execution of code actually happens at the same time on CePUs and CoPUs.

Parameters

in	<code>numcores</code>	is the number of cores that shall be linked together. A value of 0 or 1 will return an error (<code>PN_ERR_PARAM</code>) to aid debugging.
----	-----------------------	--

Returns

The ID of the core, or `PN_ERR_MATCH`, `PN_ERR_PARAM`, or `PN_ERR_COPU`.

6.7.2.2 `pn_begin_linked_gm()`

```
PN_CID pn_begin_linked_gm (
    PN_CMSK * coremask_array,
    PN_NUMG array_size )
```

Links the CPUs specified in the `coremask_array`.

Todo Currently only a stub. Will therefore always return `PN_ERR_NOIMP`.

Sets the CoPUs represented by bitmask and their position in the array (= their group number) to Mode 1 (linked Mode) so they start executing the instruction stream fetched by the CePU. This function will return an error ([PN_ERR_MATCH](#)) if the CoPUs are not all halted.

See documentation of [Modes](#) for more information.

Cannot be executed on CoPU. Cannot be mixed with threaded mode or other linked mode functions, until [pn_end_linked\(\)](#) is called.

Attention

All data that you want to preserve after [pn_end_linked\(\)](#) was called can not be stored on stack. You can make it static or global.

Warning

There is no guarantee that the execution of code actually happens at the same time on CePUs and CoPUs.

Parameters

in	<i>coremask_array</i>	is a pointer to the start of a coremask array. The position of the mask in the array represents the group number of the cores. When all bits are set to 0, this function will return an error (PN_ERR_PARAM) to aid debugging. Also, not setting the first bit to 1 will return an error (PN_ERR_PARAM) since it represents the CePU (which needs to be linked, too).
in	<i>array_size</i>	is the number of entries in the coremask_array.

Returns

The ID of the core, or [PN_ERR_MATCH](#), [PN_ERR_PARAM](#), or [PN_ERR_COPU](#).

6.7.2.3 pn_begin_linked_m()

```
PN_CID pn_begin_linked_m (
    PN_CMSK coremask )
```

Links the CPUs specified in the coremask.

Sets the CoPUs represented by the bitmask to Mode 1 (linked Mode) so they start executing the instruction stream fetched by the CePU. This function will return an error ([PN_ERR_MATCH](#)) if the CoPUs are not all halted.

See documentation of [Modes](#) for more information.

Cannot be executed on CoPU. Cannot be mixed with threaded mode or other linked mode functions, until [pn_end_linked\(\)](#) is called.

Attention

All data that you want to preserve after [pn_end_linked\(\)](#) was called can not be stored on stack. You can make it static or global.

Warning

There is no guarantee that the execution of code actually happens at the same time on CePUs and CoPUs.

Parameters

in	<i>coremask</i>	is the bitmask representing the CoPUs you want to link. When all bits are set to 0, this function will return an error (PN_ERR_PARAM) to aid debugging. Also, not setting the first bit to 1 will return an error (PN_ERR_PARAM) since it represents the CePU (which needs to be linked, too).
----	-----------------	--

Returns

The ID of the core, or [PN_ERR_MATCH](#), [PN_ERR_PARAM](#), or [PN_ERR_COPU](#).

6.7.2.4 pn_end_linked()

```
int pn_end_linked (
    void )
```

Ends linked execution.

Halts all CoPUs that are currently linked together, effectively ending the linked execution. Will fail if there are no cores linked together.

See documentation of [Modes](#) for more information.

Can be executed on CoPU, but will do nothing then.

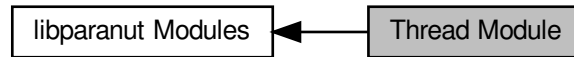
Returns

Either [PN_SUCCESS](#) or [PN_ERR_MATCH](#).

6.8 Thread Module

Functions for using the threaded mode.

Collaboration diagram for Thread Module:



- void [pn_thread_entry](#) (void)
Function that has to be called for CoPUs at the end of the startup code.
- [PN_CID pn_begin_threaded](#) ([PN_NUMC](#) numcores)
Puts numcores CPUs in threaded mode.
- [PN_CID pn_begin_threaded_m](#) ([PN_CMSK](#) coremask)
Puts the CPUs specified in the coremask in threaded mode.
- [PN_CID pn_begin_threaded_gm](#) ([PN_CMSK](#) *coremask_array, [PN_NUMG](#) array_size)
Puts the CPUs specified in the coremask_array in threaded mode.
- int [pn_end_threaded](#) (void)
Ends threaded execution.

6.8.1 Detailed Description

Functions for using the threaded mode.

Also see [Modes](#).

6.8.2 Function Documentation

6.8.2.1 pn_begin_threaded()

```
PN_CID pn_begin_threaded (
    PN_NUMC numcores )
```

Puts numcores CPUs in threaded mode.

Todo Not yet implemented for more cores than what is available in one group. Will return [PN_ERR_NOIMP](#) when called with more cores.

Sets numcores-1 CoPUs to Mode 2 (unlinked Mode) so they start executing the following code in parallel. This function will return an error ([PN_ERR_MATCH](#)) if the CoPUs are not all halted.

See documentation of [Modes](#) for more information.

Cannot be executed on CoPU. Cannot be mixed with linked mode or other begin threaded functions, until [pn_end_threaded\(\)](#) is called.

Attention

All data that you want to preserve after `pn_end_threaded()` was called can not be stored on stack. You can make it static or global.

Warning

There is no guarantee that the execution of code actually happens at the same time on CePUs and CoPUs.

Parameters

in	<i>numcores</i>	is the number of cores that shall run threaded. A value of 0 or 1 will return an error (<code>PN_ERR_PARAM</code>) to aid debugging.
----	-----------------	--

Returns

The ID of the core, or `PN_ERR_MATCH`, `PN_ERR_PARAM`, or `PN_ERR_COPU`.

6.8.2.2 pn_begin_threaded_gm()

```
PN_CID pn_begin_threaded_gm (
    PN_CMSK * coremask_array,
    PN_NUMG array_size )
```

Puts the CPUs specified in the `coremask_array` in threaded mode.

Todo Currently only a stub. Will therefore always return `PN_ERR_NOIMP`.

Sets the CoPUs represented by bitmask and their position in the array (= their group number) to Mode 2 (unlinked Mode) so they start executing the following code in parallel. This function will return an error (`PN_ERR_MATCH`) if the CoPUs are not all halted.

See documentation of [Modes](#) for more information.

Cannot be executed on CoPU. Cannot be mixed with linked mode or other begin threaded functions, until `pn_end_threaded()` is called.

Attention

All data that you want to preserve after `pn_end_threaded()` was called can not be stored on stack. You can make it static or global.

Warning

There is no guarantee that the execution of code actually happens at the same time on CePUs and CoPUs.

Parameters

in	<i>coremask_array</i>	is a pointer to the start of a coremask array. The position of the mask in the array represents the group number of the cores. When all bits are set to 0, this function will return an error (PN_ERR_PARAM) to aid debugging. Also, not setting the first bit to 1 will return an error (PN_ERR_PARAM) since it represents the CePU (which needs to run threaded, too).
in	<i>array_size</i>	is the number of entries in the coremask_array.

Returns

The ID of the core, or [PN_ERR_MATCH](#), [PN_ERR_PARAM](#), or [PN_ERR_COPU](#).

6.8.2.3 `pn_begin_threaded_m()`

```
PN_CID pn_begin_threaded_m (
    PN_CMSK coremask )
```

Puts the CPUs specified in the coremask in threaded mode.

Sets the CoPUs represented by the bitmask to Mode 2 (unlinked Mode) so they start executing the following code in parallel. This function will return an error ([PN_ERR_MATCH](#)) if the CoPUs are not all halted.

See documentation of [Modes](#) for more information.

Cannot be executed on CoPU. Cannot be mixed with linked mode or other begin threaded functions, until [pn_end_threaded\(\)](#) is called.

Attention

All data that you want to preserve after [pn_end_threaded\(\)](#) was called can not be stored on stack. You can make it static or global.

Warning

There is no guarantee that the execution of code actually happens at the same time on CePUs and CoPUs.

Parameters

in	<i>coremask</i>	is the bitmask representing the CoPUs you want to run threaded. When all bits are set to 0, this function will return an error (PN_ERR_PARAM) to aid debugging. Also, not setting the first bit to 1 will return an error (PN_ERR_PARAM) since it represents the CePU (which needs to run threaded, too).
----	-----------------	---

Returns

The ID of the core, or [PN_ERR_MATCH](#), [PN_ERR_PARAM](#), or [PN_ERR_COPU](#).

6.8.2.4 `pn_end_threaded()`

```
int pn_end_threaded (  
    void )
```

Ends threaded execution.

On CoPU, halts the core. On CePU, waits until all other cores are halted. This ends the threaded mode.

See documentation of [Modes](#) for more information.

Returns

Either [PN_SUCCESS](#) or [PN_ERR_MATCH](#).

6.8.2.5 `pn_thread_entry()`

```
void pn_thread_entry (  
    void )
```

Function that has to be called for CoPUs at the end of the startup code.

Marks the entry point of CoPUs into the [Thread Module](#). Necessary for threaded Mode.

The CoPUs are set up to work correctly in threaded Mode.

Execution is only effective on CoPU. Can be executed on CePU, will do nothing then.

Should not be called in a normal application at all. Left in here for startup code writers convenience.

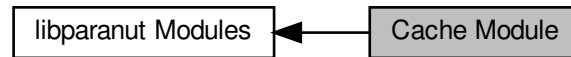
Attention

What comes after this part in the startup code is irrelevant, since the CoPUs that landed there are either put into threaded mode or halted. The function can therefore not return any errors.

6.9 Cache Module

Special functions for controlling the shared ParaNut cache.

Collaboration diagram for Cache Module:



- int [pn_cache_init](#) (void)
Function that has to be called in the main function before any of the functions in the [Cache Module](#) can be called. Initializes some internal data and enables the cache.
- int [pn_cache_enable](#) (void)
Enables instruction and data cache. When changing this, make sure that the [pn_cache_init\(\)](#) function is still correct!
- int [pn_cache_disable](#) (void)
Disables instruction and data cache.
- unsigned long [pn_cache_linesize](#) (void)
Returns the cache line size in bit.
- unsigned long [pn_cache_size](#) (void)
Returns the cache size in Byte.
- int [pn_cache_invalidate](#) (void *addr, unsigned long size)
Invalidates the cache entries containing the given address range.
- int [pn_cache_invalidate_all](#) (void)
Invalidates the whole cache. When changing this, make sure that the [pn_cache_init\(\)](#) function is still correct!
- int [pn_cache_writeback](#) (void *addr, unsigned long size)
Writes back the cache lines that cached the given address range.
- int [pn_cache_writeback_all](#) (void)
Writes whole cache back.
- int [pn_cache_flush](#) (void *addr, unsigned long size)
Combination of [pn_cache_invalidate\(\)](#) and [pn_cache_writeback\(\)](#).
- int [pn_cache_flush_all](#) (void)
Flushes the whole cache.

6.9.1 Detailed Description

Special functions for controlling the shared ParaNut cache.

6.9.2 Function Documentation

6.9.2.1 `pn_cache_disable()`

```
int pn_cache_disable (
    void )
```

Disables instruction and data cache.

Attention

Careful here: ParaNut Cache is flushed completely before disabling.

Warning

Atomic memory operations ([Spinlock Module](#)) are not possible on a disabled cache.

Can only be used on CePU.

Returns

Either [PN_SUCCESS](#) or [PN_ERR_COPU](#).

6.9.2.2 `pn_cache_enable()`

```
int pn_cache_enable (
    void )
```

Enables instruction and data cache. When changing this, make sure that the [pn_cache_init\(\)](#) function is still correct!

Attention

Careful here: ParaNut Cache is flushed completely before disabling.

Can only be used on CePU.

Returns

Either [PN_SUCCESS](#) or [PN_ERR_COPU](#).

6.9.2.3 `pn_cache_flush()`

```
int pn_cache_flush (
    void * addr,
    unsigned long size )
```

Combination of [pn_cache_invalidate\(\)](#) and [pn_cache_writeback\(\)](#).

Parameters

in	<i>addr</i>	is the (virtual)start address of the memory you want flushed.
in	<i>size</i>	is the size of the address range you want flushed in byte. The size will always be aligned to the cache line size.

Returns

Either [PN_SUCCESS](#) or [PN_ERR_PARAM](#) if given size is bigger than memory size.

6.9.2.4 pn_cache_flush_all()

```
int pn_cache_flush_all (
    void )
```

Flushes the whole cache.

Returns

Can only return [PN_SUCCESS](#), is not made void for the sake of making the internal implementation more similar for cache functions.

6.9.2.5 pn_cache_init()

```
int pn_cache_init (
    void )
```

Function that has to be called in the main function before any of the functions in the [Cache Module](#) can be called. Initializes some internal data and enables the cache.

Can only be used on CePU.

Returns

Either [PN_SUCCESS](#), [PN_ERR_COPU](#), or [PN_CACHE_LINESIZE](#).

6.9.2.6 pn_cache_invalidate()

```
int pn_cache_invalidate (
    void * addr,
    unsigned long size )
```

Invalidates the cache entries containing the given address range.

Parameters

in	<i>addr</i>	is the (virtual) start address of the memory you want to invalidate.
in	<i>size</i>	is the size of the address range you want to invalidate in byte. The size will always be aligned to the cache line size.

Returns

Either [PN_SUCCESS](#) or [PN_ERR_PARAM](#) if given size is bigger than memory size.

6.9.2.7 pn_cache_invalidate_all()

```
int pn_cache_invalidate_all (
    void )
```

Invalidates the whole cache. When changing this, make sure that the [pn_cache_init\(\)](#) function is still correct!

Returns

Can only return [PN_SUCCESS](#), is not made void for the sake of making the internal implementation more similar for cache functions.

6.9.2.8 pn_cache_linesize()

```
unsigned long pn_cache_linesize (
    void )
```

Returns the cache line size in bit.

Returns

The cache line size in bit. Can not return an error.

6.9.2.9 pn_cache_size()

```
unsigned long pn_cache_size (
    void )
```

Returns the cache size in Byte.

Returns

The cache size in byte. Can not return an error.

6.9.2.10 pn_cache_writeback()

```
int pn_cache_writeback (
    void * addr,
    unsigned long size )
```

Writes back the cache lines that cached the given address range.

Parameters

in	<i>addr</i>	is the (virtual) start address of the memory you want written back.
in	<i>size</i>	is the size of the address range you want written back in byte. The size will always be aligned to the cache line size.

Returns

Either [PN_SUCCESS](#) or [PN_ERR_PARAM](#) if given size is bigger than memory size.

6.9.2.11 `pn_cache_writeback_all()`

```
int pn_cache_writeback_all (  
    void )
```

Writes whole cache back.

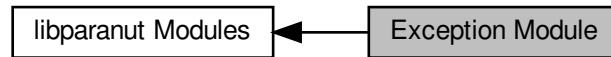
Returns

Can only return [PN_SUCCESS](#), is not made void for the sake of making the internal implementation more similar for cache functions.

6.10 Exception Module

Functions for controlling the handling of interrupts/exceptions.

Collaboration diagram for Exception Module:



- void [pn_exception_init](#) (void)
Initializes libparanut internal exception handling. Interrupts (not exceptions in general!) are disabled after. Should be called before using [pn_exception_set_handler](#)).
- int [pn_exception_set_handler](#) (void(*handler)(unsigned int cause, unsigned int program_counter, unsigned int mtval), unsigned int exception_code)
Set your own exception handler.
- void [pn_ecall](#) (void)
Raises an environment call exception.
- void [pn_interrupt_enable](#) (void)
Enables interrupts only.
- void [pn_interrupt_disable](#) (void)
Disables interrupts only.
- void [pn_progress_mepc](#) (void)
Sets program counter of the register which keeps the exception return adress to next instruction.

6.10.1 Detailed Description

Functions for controlling the handling of interrupts/exceptions.

Why are we calling this exceptions, even though most people would call this an interrupt? Historic reasons. The libparanut was first written for the RISCv implementation of the ParaNut, and the RISCv specification refers to both interrupts and exceptions as exceptions.

6.10.2 Function Documentation

6.10.2.1 [pn_ecall\(\)](#)

```
void pn_ecall (
    void )
```

Raises an environment call exception.

Can be called without using [pn_exception_init\(\)](#) first.

6.10.2.2 `pn_exception_set_handler()`

```
int pn_exception_set_handler (
    void(*) (unsigned int cause, unsigned int program_counter, unsigned int mtval)
    handler,
    unsigned int exception_code )
```

Set your own exception handler.

Can be called without using `pn_exception_init()` first, will not work though.

Already does the work of saving away registers, setting program counter etc. for you. You can just hang in what you want to do.

Attention

For exceptions, the register that contains the address where execution resumes is set to the faulty instruction that threw the exception. For interrupts, it already points to the instruction where execution should resume. Consider this in your handler. If you need to, use `pn_progress_mepc`.

Parameters

in	<i>handler</i>	is a function pointer to your exception handler. Will return an error (PN_ERR_PARAM) if NULL is given.
in	<i>exception_code</i>	is the number that your exception has. You can look up the exception codes in the ParaNut Manual. For interrupts, the value of the most significant bit of the exception code has to be 1. For synchronous exceptions, it has to be 0. This function will return an error (PN_ERR_EXC) if a non implemented value for cause is given.

Returns

Either [PN_SUCCESS](#), [PN_ERR_EXC](#), or [PN_ERR_PARAM](#).

6.10.2.3 `pn_interrupt_disable()`

```
void pn_interrupt_disable (
    void )
```

Disables interrupts only.

Can be called without using `pn_exception_init()` first.

6.10.2.4 `pn_interrupt_enable()`

```
void pn_interrupt_enable (
    void )
```

Enables interrupts only.

Can be called without using `pn_exception_init()` first.

6.10.2.5 `pn_progress_mepc()`

```
void pn_progress_mepc (  
    void )
```

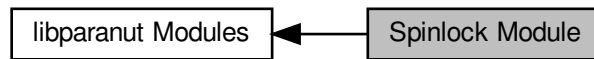
Sets program counter of the register which keeps the exception return adress to next instruction.

Can be called without using [pn_exception_init\(\)](#) first.

6.11 Spinlock Module

Functions and structure used for synchronizing memory access.

Collaboration diagram for Spinlock Module:



Classes

- struct [__pn_spinlock](#)
A synchronization primitive. Use [__pn_spinlock](#) instead of this.
- int [pn_spinlock_init](#) ([__pn_spinlock](#) *spinlock)
Creates a lock.
- int [pn_spinlock_lock](#) ([__pn_spinlock](#) *spinlock)
Waits for a lock. Forever, if it must. Use with caution.
- int [pn_spinlock_trylock](#) ([__pn_spinlock](#) *spinlock)
Tries to acquire a lock. Nonblocking.
- int [pn_spinlock_unlock](#) ([__pn_spinlock](#) *spinlock)
Unlocks a lock.
- int [pn_spinlock_destroy](#) ([__pn_spinlock](#) *spinlock)
Destroys a lock.

6.11.1 Detailed Description

Functions and structure used for synchronizing memory access.

Warning

The functions in here are really kinda performance critical, since it is always important to do as little as possible when you have reserved a memory area. This means that the functions will do extremely little security checks, which means you have to use them the way they are described. Read the detailed descriptions of the functions carefully. Or don't. I'm not the coding police.

Using the spinlock functions in linked Mode (see [Link Module](#) and [Modes](#)) results in undefined behaviour.

If you want to use the Spinlock Module on RISC-V ParaNut, your ParaNut has to support the A Extension and you have to compile your application with the flag mabi=rv32ia. The libparanut [Makefile](#) sets this flag automatically when you chose the [Spinlock Module](#) or one of the Modules that has [Spinlock Module](#) as a dependency.

Return value of functions is always error code, except when stated otherwise in the description of the function.

6.11.2 Function Documentation

6.11.2.1 `pn_spinlock_destroy()`

```
int pn_spinlock_destroy (
    __pn_spinlock * spinlock )
```

Destroys a lock.

Behaviour of this function is undefined if the lock wasn't initialized by `pn_spinlock_init()`.

A destroyed lock can be re-initialized by using `pn_spinlock_init()`.

The lock can either be owned by current hart or unlocked, else the function will fail.

Parameters

<i>spinlock</i>	is a pointer to a lock that we want to destroy. The function will return <code>PN_ERR_PARAM</code> if NULL is passed.
-----------------	---

Returns

Either `PN_SUCCESS`, `PN_ERR_LOCKOCC` or `PN_ERR_PARAM`. If something internally went very wrong, the function is also theoretically able to return `PN_ERR_NOIMP`.

6.11.2.2 `pn_spinlock_init()`

```
int pn_spinlock_init (
    __pn_spinlock * spinlock )
```

Creates a lock.

You allocate the space for the spinlock. Really don't care where you get it from (check `__pn_spinlock` for a recommendation). You pass a reference to this function, and the function initializes it for you. And you shall never touch what's in it.

Afterwards, you can use the other functions in this module on the same lock. Behaviour will always be undefined if you don't call this function first.

The function does not care if your lock was already initialized. It will fail if the CPU did not get the memory reservation (`PN_ERR_LOCKOCC`), which should never happen. This is a very good indicator that something is very wrong with your program.

After initialization, the lock is free. It will not be automatically owned by the hart that initialized it.

Parameters

<i>spinlock</i>	is a pointer to the lock. The function will return <code>PN_ERR_PARAM</code> if NULL is passed.
-----------------	---

Returns

Either [PN_SUCCESS](#), [PN_ERR_PARAM](#), or [PN_ERR_LOCKOCC](#). If something internally went very wrong, the function is also theoretically able to return [PN_ERR_NOIMP](#).

6.11.2.3 pn_spinlock_lock()

```
int pn_spinlock_lock (
    _pn_spinlock * spinlock )
```

Waits for a lock. Forever, if it must. Use with caution.

Behaviour of this function is undefined if the lock wasn't initialized by [pn_spinlock_init\(\)](#).

Warning

The function will be stuck in eternity if the lock is already in the current harts possession, or if someone else owns the lock and forgot to unlock it, or if the lock was destroyed.

Parameters

<i>spinlock</i>	is a pointer to a lock that we want to aquire. The function will return PN_ERR_PARAM if NULL is passed.
-----------------	---

Returns

Either [PN_SUCCESS](#) or [PN_ERR_PARAM](#).

6.11.2.4 pn_spinlock_trylock()

```
int pn_spinlock_trylock (
    _pn_spinlock * spinlock )
```

Tries to acquire a lock. Nonblocking.

Behaviour of this function is undefined if the lock wasn't initialized by [pn_spinlock_init\(\)](#).

Will fail if lock is already owned (no matter by whom), another CPU got the memory reservation, or if lock was destroyed.

Parameters

<i>spinlock</i>	is a pointer to a lock that we want to aquire. The function will return PN_ERR_PARAM if NULL is passed.
-----------------	---

Returns

Either [PN_SUCCESS](#), [PN_ERR_LOCKOCC](#) or [PN_ERR_PARAM](#). If something internally went very wrong, the function is also theoretically able to return [PN_ERR_NOIMP](#).

6.11.2.5 `pn_spinlock_unlock()`

```
int pn_spinlock_unlock (
    _pn_spinlock * spinlock )
```

Unlocks a lock.

Behaviour of this function is undefined if the lock wasn't initialized by [pn_spinlock_init\(\)](#).

Will fail is lock is not owned by current hart ([PN_ERR_PARAM](#)).

Parameters

<i>spinlock</i>	is a pointer to a lock that we want to unlock. The function will return PN_ERR_PARAM if NULL is passed.
-----------------	---

Returns

Either [PN_SUCCESS](#) or [PN_ERR_PARAM](#). If something internally went very wrong, the function is also theoretically able to return [PN_ERR_NOIMP](#).

6.12 libparanut Compile Time Parameters

Group contains defines that inform the application writer how the libparanut was compiled.

Macros

- `#define PN_CACHE_LINESIZE`
Size of a cache line in bit.
- `#define PN_RWIDTH 32`
Register width in bit.
- `#define PN_COMPILE_RAW`
All security checks in libparanut are dropped if this is set to 1.
- `#define PN_WITH_BASE`
libparanut was compiled with [Base Module](#). Also check [Modules of the libparanut](#) for more information.
- `#define PN_WITH_CACHE`
libparanut was compiled with [Cache Module](#). Also check [Modules of the libparanut](#) for more information.
- `#define PN_WITH_LINK`
libparanut was compiled with [Link Module](#). Also check [Modules of the libparanut](#) for more information.
- `#define PN_WITH_THREAD`
libparanut was compiled with [Thread Module](#). Also check [Modules of the libparanut](#) for more information.
- `#define PN_WITH_EXCEPTION`
libparanut was compiled with [Exception Module](#). Also check [Modules of the libparanut](#) for more information.
- `#define PN_WITH_SPINLOCK`
libparanut was compiled with [Spinlock Module](#). Also check [Modules of the libparanut](#) for more information.

6.12.1 Detailed Description

Group contains defines that inform the application writer how the libparanut was compiled.

The libparanut is a very flexible piece of software. Some modules may have been compiled in, others may not. The cache line size could have a fixed value to improve speed, or it could be set on auto which is more compatible. Find out by including [pn_config.h](#) in your application and checking the defines listed in here!

6.12.2 Macro Definition Documentation

6.12.2.1 PN_CACHE_LINESIZE

```
#define PN_CACHE_LINESIZE
```

Size of a cache line in bit.

This decides which assembly file was included during compilation of the [Cache Module](#). If "auto" was chosen, the file that contains functions for all possible cache line sizes is included. This means great binary compatibility, but terribly big code size. When your application is deployed, you should definitely compile and link a version of libparanut with this parameter set to the cache line size you want to use eventually.

Also check documentation of [pn_cache_RV32I_buildscript.py](#).

6.12.2.2 PN_COMPILE_RAW

```
#define PN_COMPILE_RAW
```

All security checks in libparanut are dropped if this is set to 1.

Since functions in this library may be timing critical, you can compile the libparanut with this parameter and disable all the security checks.

While you are developing, it is recommended you don't do this, as the security checks will tell you when you are giving input that does not make sense. You can enable it when you properly tested your system to get optimal performance. Or don't. I mean, it's not like I'm the code police.

6.12.2.3 PN_RWIDTH

```
#define PN_RWIDTH 32
```

Register width in bit.

Was set to 32 bit in this documentation to enable Doxygen to properly write down the typedefs in the Typedefs section of [libparanut.h](#). This should not be of interest at the moment since there is only a 32 bit version of the ParaNut, but it may become relevant in the future.

6.13 Comm_def

6.13.1 Detailed Description

6.14 Comm_glo

6.14.1 Detailed Description

Chapter 7

Class Documentation

7.1 `__pn_spinlock` Struct Reference

A synchronization primitive. Use [_pn_spinlock](#) instead of this.

```
#include <libparanut.h>
```

7.1.1 Detailed Description

A synchronization primitive. Use [_pn_spinlock](#) instead of this.

A simple implementation of a synchronization primitive. Might be used for implementing POSIX Threads later on.

Warning

You are not supposed to touch anything in this struct, which is why you can't see anything about the members in this documentation.

Attention

I highly recommend to not put locks on stack. The reason is that the stack (or at least a part of it) will be copied when putting the ParaNut into threaded Mode (see [pn_begin_threaded\(\)](#)). In other words, if you put a lock on stack, it will be copied, and the new instances of the lock will be available per core. You should make it static, global or get the memory by allocation.

Warning

Atomic memory operations are not possible on a disabled cache.

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

- [libparanut.h](#)

Chapter 8

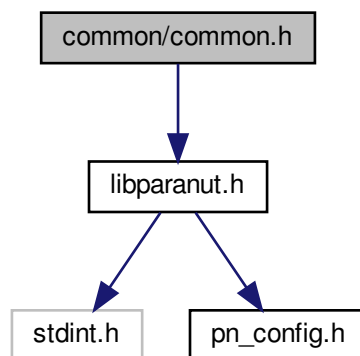
File Documentation

8.1 common/common.h File Reference

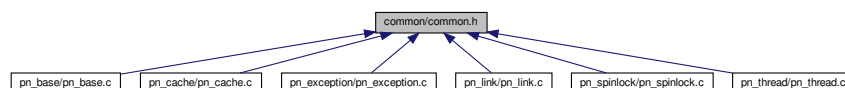
Contains architecture independent internal prototypes and defines needed in [libparanut Modules](#).

```
#include "libparanut.h"
```

Include dependency graph for common.h:



This graph shows which files directly or indirectly include this file:



8.1.1 Detailed Description

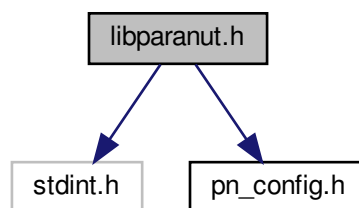
Contains architecture independent internal prototypes and defines needed in [libparanut Modules](#).

Is included by all modules and includes the [libparanut.h](#) itself, thereby is a "common layer" for all modules.

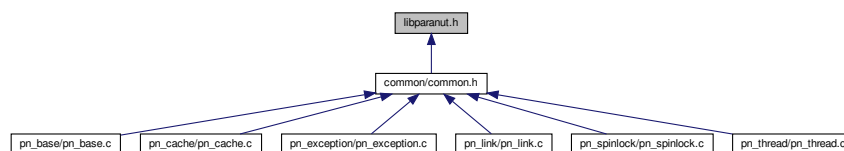
8.2 libparanut.h File Reference

API of the libparanut.

```
#include <stdint.h>
#include "pn_config.h"
Include dependency graph for libparanut.h:
```



This graph shows which files directly or indirectly include this file:



Classes

- struct [__pn_spinlock](#)

A synchronization primitive. Use [_pn_spinlock](#) instead of this.

Macros

- `#define PN_SUCCESS 0`
Successful execution.
 - `#define PN_ERR_PARAM (-1)`
Parameter error.
 - `#define PN_ERR_NOIMP (-2)`
Function not implemented.
 - `#define PN_ERR_COPU (-3)`
CoPU error.
 - `#define PN_ERR_MATCH (-4)`
Mode begin and end matching error.
 - `#define PN_ERR_LOCKOCC (-5)`
Lock occupied error.
 - `#define PN_ERR_CACHE_LINESIZE (-6)`
Weird cache line size error.
 - `#define PN_ERR_EXC (-8)`
Exception code not implemented error.
-
- `#define PN_M0 0x0U`
Mode 0 (halted Mode).
 - `#define PN_M1 0x1U`
Mode 1 (linked Mode).
 - `#define PN_M2 0x2U`
Mode 2 (unlinked or threaded Mode).
 - `#define PN_M3 0x3U`
Mode 3 (autonomous Mode).

Typedefs

- `typedef struct __pn_spinlock _pn_spinlock`
Renaming of struct `__pn_spinlock` for your convenience.
-
- `typedef int32_t PN_CID`
Signed type that can be used to address any core in this architecture.
 - `typedef int32_t PN_NUMC`
Signed type that can hold the maximum number of cores in this architecture.
 - `typedef uint32_t PN_CMSK`
Unsigned type that can act as a core mask.
 - `typedef int32_t PN_NUMG`
Signed type that can be used to address any group in this architecture.

Functions

- [PN_NUMC pn_numcores](#) (void)
Get the number of cores in your system.
 - [PN_CMSK pn_m2cap](#) (void)
Check which cores are capable of Mode 2 operation.
 - [PN_CMSK pn_m2cap_g](#) (PN_NUMG groupnum)
Check which cores are capable of Mode 2 operation.
 - [PN_CMSK pn_m3cap](#) (void)
Check which cores are capable of Mode 3 operation.
 - [PN_CMSK pn_m3cap_g](#) (PN_NUMG groupnum)
Check which cores are capable of Mode 3 operation.
 - [PN_CID pn_coreid](#) (void)
Get the ID of the core that this function is executed on.
 - [PN_CID pn_coreid_g](#) (PN_NUMG *groupnum)
Get the ID and group number of the core that this code is running on.
 - void [pn_halt](#) (void)
Halt whatever core the function is executed on.
 - int [pn_halt_CoPU](#) (PN_CID coreid)
Halts a CoPU.
 - int [pn_halt_CoPU_m](#) (PN_CMSK coremask)
Halts one or more CoPUs.
 - int [pn_halt_CoPU_gm](#) (PN_CMSK *coremask_array, PN_NUMG array_size)
Halts the CoPUs specified in the coremask_array.
 - long long int [pn_time_ns](#) (void)
Returns system time in ns. Does not care for overflow.
 - int [pn_simulation](#) (void)
Checks if we run in simulation instead of real hardware.
-
- [PN_CID pn_begin_linked](#) (PN_NUMC numcores)
Links a given number of CoPUs to the CePU.
 - [PN_CID pn_begin_linked_m](#) (PN_CMSK coremask)
Links the CPUs specified in the coremask.
 - [PN_CID pn_begin_linked_gm](#) (PN_CMSK *coremask_array, PN_NUMG array_size)
Links the CPUs specified in the coremask_array.
 - int [pn_end_linked](#) (void)
Ends linked execution.
-
- void [pn_thread_entry](#) (void)
Function that has to be called for CoPUs at the end of the startup code.
 - [PN_CID pn_begin_threaded](#) (PN_NUMC numcores)
Puts numcores CPUs in threaded mode.

- [PN_CID pn_begin_threaded_m](#) ([PN_CMSK](#) coremask)
Puts the CPUs specified in the coremask in threaded mode.
- [PN_CID pn_begin_threaded_gm](#) ([PN_CMSK](#) *coremask_array, [PN_NUMG](#) array_size)
Puts the CPUs specified in the coremask_array in threaded mode.
- int [pn_end_threaded](#) (void)
Ends threaded execution.

- int [pn_cache_init](#) (void)
Function that has to be called in the main function before any of the functions in the [Cache Module](#) can be called. Initializes some internal data and enables the cache.
- int [pn_cache_enable](#) (void)
Enables instruction and data cache. When changing this, make sure that the [pn_cache_init\(\)](#) function is still correct!
- int [pn_cache_disable](#) (void)
Disables instruction and data cache.
- unsigned long [pn_cache_linesize](#) (void)
Returns the cache line size in bit.
- unsigned long [pn_cache_size](#) (void)
Returns the cache size in Byte.
- int [pn_cache_invalidate](#) (void *addr, unsigned long size)
Invalidates the cache entries containing the given address range.
- int [pn_cache_invalidate_all](#) (void)
Invalidates the whole cache. When changing this, make sure that the [pn_cache_init\(\)](#) function is still correct!
- int [pn_cache_writeback](#) (void *addr, unsigned long size)
Writes back the cache lines that cached the given address range.
- int [pn_cache_writeback_all](#) (void)
Writes whole cache back.
- int [pn_cache_flush](#) (void *addr, unsigned long size)
Combination of [pn_cache_invalidate\(\)](#) and [pn_cache_writeback\(\)](#).
- int [pn_cache_flush_all](#) (void)
Flushes the whole cache.

- void [pn_exception_init](#) (void)
Initializes libparanut internal exception handling. Interrupts (not exceptions in general!) are disabled after. Should be called before using [pn_exception_set_handler\(\)](#).
- int [pn_exception_set_handler](#) (void(*handler)(unsigned int cause, unsigned int program_counter, unsigned int mtval), unsigned int exception_code)
Set your own exception handler.
- void [pn_ecall](#) (void)
Raises an environment call exception.
- void [pn_interrupt_enable](#) (void)
Enables interrupts only.
- void [pn_interrupt_disable](#) (void)
Disables interrupts only.
- void [pn_progress_mepc](#) (void)
Sets program counter of the register which keeps the exception return adress to next instruction.

- `int pn_spinlock_init (_pn_spinlock *spinlock)`
Creates a lock.
- `int pn_spinlock_lock (_pn_spinlock *spinlock)`
Waits for a lock. Forever, if it must. Use with caution.
- `int pn_spinlock_trylock (_pn_spinlock *spinlock)`
Tries to acquire a lock. Nonblocking.
- `int pn_spinlock_unlock (_pn_spinlock *spinlock)`
Unlocks a lock.
- `int pn_spinlock_destroy (_pn_spinlock *spinlock)`
Destroys a lock.

8.2.1 Detailed Description

API of the libparanut.

8.3 Makefile File Reference

Makefile of the libparanut.

8.3.1 Detailed Description

Makefile of the libparanut.

This Makefile is supposed to make (ha!) the compilation of libparanut more handy. To check out exactly how this works, see the section [HOWTO](#) in the mainpage!

```

1 #####
2 # Copyright 2019-2020 Anna Pfuetzner (<annakerstin.pfuetzner@gmail.com>)
3 #
4 # Redistribution and use in source and binary forms, with or without modification, are permitted
5 # provided that the following conditions are met:
6 #
7 # 1. Redistributions of source code must retain the above copyright notice, this list of conditions
8 # and the following disclaimer.
9 #
10 # 2. Redistributions in binary form must reproduce the above copyright notice, this list of
11 # conditions and the following disclaimer in the documentation and/or other materials provided with
12 # the distribution.
13 #
14 # THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR
15 # IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
16 # FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR
17 # CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
18 # DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
19 # DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER
20 # IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
21 # OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
22 #####
23
24 #System Configuration#####
25
26 # Call clean when changing these!
27
28 # System Parameters
29 PN_CACHE_LINESIZE      = auto          # Cache Linesize in Bits, Default: auto
30                          # auto means all power of two linesizes from 32 to 2048
31 PN_RWIDTH              = 32             # Register Width in Bits, Default: 32
32

```

```

33 # Switch for raw compilation (optimized for performance, will drop security checks)
34 # Default: 0
35 PN_COMPILE_RAW                = 0
36
37 # Chose modules - Set to 0 to switch off
38 # Default: 1
39 PN_WITH_BASE                  = 1
40 PN_WITH_CACHE                  = 1
41 PN_WITH_LINK                   = 1
42 PN_WITH_THREAD                 = 1
43 PN_WITH_EXCEPTION              = 1
44 PN_WITH_SPINLOCK               = 1
45
46 #Compiler Configuration#####
47
48 # Instruction Set Architecture
49 # Currently available:
50 #   1. RISC-V 32 bit - Set to RV32I
51 # Don't forget to change compiler when touching this!
52 PN_ISA                         = RV32I
53
54 # Compiler
55 # Currently available:
56 #   1. GCC - Available GCC for chosen ISA
57 PN_COMPILER                    = GCC
58
59 # Compile with Debug Symbols
60 # Default: 1
61 PN_DEBUG                       = 1
62
63 #Compiler and Assembler Flags#####
64
65 ifeq ($(PN_ISA),RV32I)
66
67     ifeq ($(PN_COMPILER),GCC)
68
69         # Set the list of system parameters together
70         PN_CONFIG_DEFINES = -D PN_CACHE_LINESIZE=$(PN_CACHE_LINESIZE) -D PN_RWIDTH=$(PN_RWIDTH)
71         PN_CONFIG_DEFINES += -D PN_JOBQUEUE_SIZE=$(PN_JOBQUEUE_SIZE)
72
73         # Put in define for raw compilation if it was set before
74         ifeq ($(PN_COMPILE_RAW),1)
75             PN_CONFIG_DEFINES += -D PN_COMPILE_RAW
76         endif
77
78         # Actual Compiler and Assembler
79         CC = riscv32-unknown-elf-gcc
80         ASM = riscv32-unknown-elf-as
81
82         # Compiler Flags
83         CFLAGS = -c -ansi -O3 -Wall -Werror -I./ -I./common $(PN_CONFIG_DEFINES)
84         CFLAGS += -mabi=ilp32
85
86         ifeq ($(PN_WITH_SPINLOCK),0)
87             ifeq ($(PN_WITH_LINK),0)
88                 CFLAGS += -march=rv32i
89             else
90                 CFLAGS += -march=rv32im
91             endif
92         else
93             ifeq ($(PN_WITH_LINK),0)
94                 CFLAGS += -march=rv32ia
95             else
96                 CFLAGS += -march=rv32ima
97             endif
98         endif
99
100         ifeq ($(PN_DEBUG),1)
101             CFLAGS += -g
102         endif
103
104         CFLAGS += -o
105
106         # Assembler Flags
107         AFLAGS = --fatal-warnings -mabi=ilp32 -I./common
108
109         ifeq ($(PN_WITH_SPINLOCK),0)
110             ifeq ($(PN_WITH_LINK),0)
111                 AFLAGS += -march=rv32i
112             else
113                 AFLAGS += -march=rv32im
114             endif
115         else
116             ifeq ($(PN_WITH_LINK),0)
117                 AFLAGS += -march=rv32ia
118             else
119                 AFLAGS += -march=rv32ima

```

```

120         endif
121     endif
122
123     ifeq ($(PN_DEBUG),1)
124         AFLAGS          += -g
125     endif
126
127     AFLAGS          += -o
128
129     else
130
131         $(error No valid compiler set for the ISA that you chose.)
132     endif
133
134
135 else
136
137     $(error No valid ISA set.)
138
139 endif
140
141 #Lists#####
142
143 # Assemble object list so we know what objects to build
144 OBJECT_LIST =
145
146 ifeq ($(PN_WITH_BASE),1)
147     PN_CONFIG_DEFINES    += -D PN_WITH_BASE
148     BPATH                = ./pn_base/pn_base
149     OBJECT_LIST          += $(BPATH).o $(BPATH)_$(PN_ISA).o
150 endif
151
152 ifeq ($(PN_WITH_LINK),1)
153     PN_CONFIG_DEFINES    += -D PN_WITH_LINK
154     LPATH                = ./pn_link/pn_link
155     OBJECT_LIST          += $(LPATH).o $(LPATH)_$(PN_ISA).o
156 endif
157
158 ifeq ($(PN_WITH_THREAD),1)
159     PN_CONFIG_DEFINES    += -D PN_WITH_THREAD
160     TPATH                = ./pn_thread/pn_thread
161     OBJECT_LIST          += $(TPATH).o $(TPATH)_$(PN_ISA).o
162 endif
163
164 ifeq ($(PN_WITH_CACHE),1)
165     PN_CONFIG_DEFINES    += -D PN_WITH_CACHE
166     CPATH                = ./pn_cache/pn_cache
167     OBJECT_LIST          += $(CPATH).o $(CPATH)_$(PN_ISA)_$(strip $(PN_CACHE_LINESIZE)).o
168 endif
169
170 ifeq ($(PN_WITH_EXCEPTION),1)
171     PN_CONFIG_DEFINES    += -D PN_WITH_EXCEPTION
172     EPATH                = ./pn_exception/pn_exception
173     OBJECT_LIST          += $(EPATH).o $(EPATH)_$(PN_ISA).o
174 endif
175
176 ifeq ($(PN_WITH_SPINLOCK),1)
177     PN_CONFIG_DEFINES    += -D PN_WITH_SPINLOCK
178     SPATH                = ./pn_spinlock/pn_spinlock
179     OBJECT_LIST          += $(SPATH).o $(SPATH)_$(PN_ISA).o
180 endif
181
182 # Check if object list is empty because it would not make any sense to build the library then
183 ifeq ($(strip $(OBJECT_LIST)),)
184     $(error No modules enabled.)
185 endif
186
187 # Add the objects for the common part
188 OBJECT_LIST          += ./common/common_$(PN_ISA).o
189
190
191 # Everything in this list has to be put into INSTALL directory
192 INSTALL_LIST          = ./INSTALL/libparanut.a ./INSTALL/libparanut.h
193 INSTALL_LIST          += ./INSTALL/pn_config.h
194
195
196 #Target Magic#####
197
198 # Creates everything for usage of libparanut
199 all: $(INSTALL_LIST)
200
201 # Creates all cache files for RV32I anew in case of changes in buildscript.
202 # Important for development only.
203 cache: ./pn_cache/pn_cache_RV32I_buildscript.py
204     python $< 32
205     python $< 64
206     python $< 128

```

```

207     python $< 256
208     python $< 512
209     python $< 1024
210     python $< 2048
211     python $< auto
212
213 # Generates library from object files
214 ./INSTALL/libparanut.a: $(OBJECT_LIST)
215     mkdir -p INSTALL
216     riscv32-unknown-elf-ar cr $@ $(OBJECT_LIST)
217
218 # Builds objects from C source code
219 %.o: %.c libparanut.h pn_config.h
220     $(CC) $(CFLAGS) $@ $<;
221
222 # Cache Module Assembly Code for RV32I is generated at compile time
223 ./pn_cache/pn_cache_RV32I_*.S: ./pn_cache/pn_cache_RV32I_buildscript.py
224     python $< $(strip $(PN_CACHE_LINESIZE))
225
226 # Put libparanut.h (API) into INSTALL directory
227 ./INSTALL/libparanut.h: libparanut.h pn_config.h
228     mkdir -p INSTALL
229     cp libparanut.h ./INSTALL/
230
231 # Generate pn_config header from system parameters - Only dependent on Makefile itself
232 SHELL = /bin/bash
233 ./INSTALL/pn_config.h:
234     mkdir -p INSTALL
235     touch ./INSTALL/pn_config.h
236     echo "/* Automatically generated file. See Makefile. No edits here! */" > ./INSTALL/pn_config.h
237     echo "#define PN_CACHE_LINESIZE $(PN_CACHE_LINESIZE)" >> ./INSTALL/pn_config.h
238     echo "#define PN_RWIDTH $(PN_RWIDTH)" >> ./INSTALL/pn_config.h
239     if ((${PN_COMPILE_RAW} == 1)); \
240     then echo "#define PN_COMPILE_RAW" >> ./INSTALL/pn_config.h; fi
241     if ((${PN_WITH_BASE} == 1)); \
242     then echo "#define PN_WITH_BASE" >> ./INSTALL/pn_config.h; fi
243     if ((${PN_WITH_CACHE} == 1)); \
244     then echo "#define PN_WITH_CACHE" >> ./INSTALL/pn_config.h; fi
245     if ((${PN_WITH_EXCEPTION} == 1)); \
246     then echo "#define PN_WITH_EXCEPTION" >> ./INSTALL/pn_config.h; fi
247     if ((${PN_WITH_LINK} == 1)); \
248     then echo "#define PN_WITH_LINK" >> ./INSTALL/pn_config.h; fi
249     if ((${PN_WITH_THREAD} == 1)); \
250     then echo "#define PN_WITH_THREAD" >> ./INSTALL/pn_config.h; fi
251     if ((${PN_WITH_SPINLOCK} == 1)); \
252     then echo "#define PN_WITH_SPINLOCK" >> ./INSTALL/pn_config.h; fi
253
254 # The other INSTALL targets just have to be copied from common directory
255 ./INSTALL/%: ./common/%
256     mkdir -p INSTALL
257     cp $< $@ ./INSTALL/
258
259 # Removes library, object files, and auto generated files
260 clean:
261     find . -name '*.o' -delete
262     if [ -d INSTALL ]; then rm -r INSTALL; fi
263
264 #EOF#####

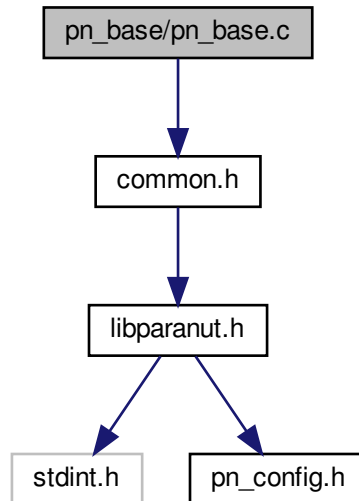
```

8.4 pn_base/pn_base.c File Reference

Contains architecture independent implementations of the [Base Module](#) functions.

```
#include "common.h"
```

Include dependency graph for pn_base.c:



Functions

- [PN_NUMC pn_numcores](#) (void)
Get the number of cores in your system.
- [PN_CMSK pn_m2cap](#) (void)
Check which cores are capable of Mode 2 operation.
- [PN_CMSK pn_m2cap_g](#) (PN_NUMG groupnum)
Check which cores are capable of Mode 2 operation.
- [PN_CMSK pn_m3cap](#) (void)
Check which cores are capable of Mode 3 operation.
- [PN_CMSK pn_m3cap_g](#) (PN_NUMG groupnum)
Check which cores are capable of Mode 3 operation.
- [PN_CID pn_coreid](#) (void)
Get the ID of the core that this function is executed on.
- [PN_CID pn_coreid_g](#) (PN_NUMG *groupnum)
Get the ID and group number of the core that this code is running on.
- void [pn_halt](#) (void)
Halt whatever core the function is executed on.
- int [pn_halt_CoPU](#) (PN_CID coreid)
Halts a CoPU.
- int [pn_halt_CoPU_m](#) (PN_CMSK coremask)
Halts one or more CoPUs.
- int [pn_halt_CoPU_gm](#) (PN_CMSK *coremask_array, PN_NUMG array_size)
Halts the CoPUs specified in the coremask_array.
- long long int [pn_time_ns](#) (void)
Returns system time in ns. Does not care for overflow.
- int [pn_simulation](#) (void)
Checks if we run in simulation instead of real hardware.

8.4.1 Detailed Description

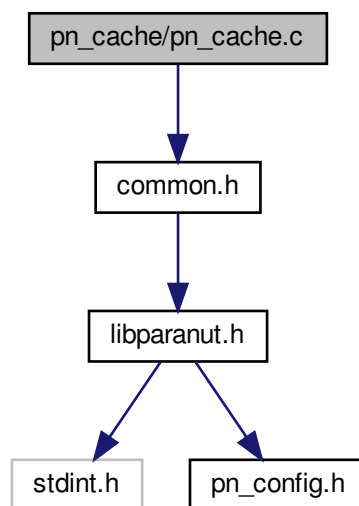
Contains architecture independent implementations of the [Base Module](#) functions.

8.5 pn_cache/pn_cache.c File Reference

Contains architecture independent implementations of the [Cache Module](#) functions.

```
#include "common.h"
```

Include dependency graph for pn_cache.c:



Functions

- int [pn_cache_init](#) (void)
Function that has to be called in the main function before any of the functions in the [Cache Module](#) can be called. Initializes some internal data and enables the cache.
- int [pn_cache_enable](#) (void)
Enables instruction and data cache. When changing this, make sure that the [pn_cache_init\(\)](#) function is still correct!
- int [pn_cache_disable](#) (void)
Disables instruction and data cache.
- unsigned long [pn_cache_linesize](#) (void)
Returns the cache line size in bit.
- unsigned long [pn_cache_size](#) (void)
Returns the cache size in Byte.
- int [pn_cache_invalidate](#) (void *addr, unsigned long size)
Invalidates the cache entries containing the given address range.
- int [pn_cache_invalidate_all](#) (void)

- Invalidates the whole cache. When changing this, make sure that the `pn_cache_init()` function is still correct!*
 - int `pn_cache_writeback` (void *addr, unsigned long size)
Writes back the cache lines that cached the given address range.
 - int `pn_cache_writeback_all` (void)
Writes whole cache back.
 - int `pn_cache_flush` (void *addr, unsigned long size)
Combination of `pn_cache_invalidate()` and `pn_cache_writeback()`.
 - int `pn_cache_flush_all` (void)
Flushes the whole cache.

8.5.1 Detailed Description

Contains architecture independent implementations of the [Cache Module](#) functions.

8.6 pn_cache/pn_cache_RV32I_buildscript.py File Reference

Builds the architecture dependent assembly files of the [Cache Module](#) since they are performance critical, very dependent on cache line size, and should not take up to much space.

8.6.1 Detailed Description

Builds the architecture dependent assembly files of the [Cache Module](#) since they are performance critical, very dependent on cache line size, and should not take up to much space.

This means we can neither make the cache line size a variable, since that would mean a performance trade off, nor can we put an implementation for every possible cache line size, since that would take a giant amount of space in the binary. This Python Script seems the perfect solution for that, except for the fact that it reduces binary compatibility of the libparanut to ParaNuts with different cache line sizes. Having to re-compile the libparanut is, of of course, a minor inconvenience, but the only other solution, which would be code that modifies itself on startup, is extremely complex and could be a possible subject for another Bachelor Thesis.

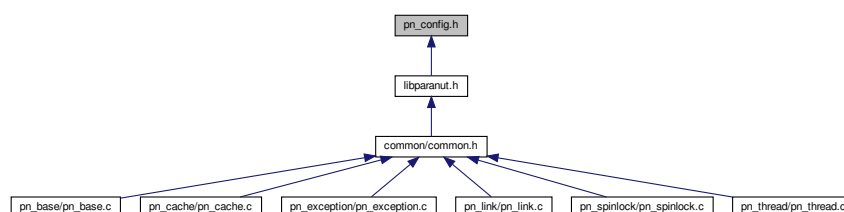
For all those who hate having to recompile stuff, it is also possible to start this script with the option "auto", which generates a file that does contain all of the cache line size functions from 32 bit to 2048 in powers of two. Since every other linesize is deemed to be unrealistic, this version has the most binary compatibility ... but also the biggest code size of them all. Make of that what you want.

Also check documentation of [PN_CACHE_LINESIZE](#).

8.7 pn_config.h File Reference

See Documentaion of [libparanut Compile Time Parameters](#).

This graph shows which files directly or indirectly include this file:



Macros

- `#define PN_CACHE_LINESIZE`
Size of a cache line in bit.
- `#define PN_RWIDTH 32`
Register width in bit.
- `#define PN_COMPILE_RAW`
All security checks in libparanut are dropped if this is set to 1.
- `#define PN_WITH_BASE`
libparanut was compiled with [Base Module](#). Also check [Modules of the libparanut](#) for more information.
- `#define PN_WITH_CACHE`
libparanut was compiled with [Cache Module](#). Also check [Modules of the libparanut](#) for more information.
- `#define PN_WITH_LINK`
libparanut was compiled with [Link Module](#). Also check [Modules of the libparanut](#) for more information.
- `#define PN_WITH_THREAD`
libparanut was compiled with [Thread Module](#). Also check [Modules of the libparanut](#) for more information.
- `#define PN_WITH_EXCEPTION`
libparanut was compiled with [Exception Module](#). Also check [Modules of the libparanut](#) for more information.
- `#define PN_WITH_SPINLOCK`
libparanut was compiled with [Spinlock Module](#). Also check [Modules of the libparanut](#) for more information.

8.7.1 Detailed Description

See Documentaion of [libparanut Compile Time Parameters](#).

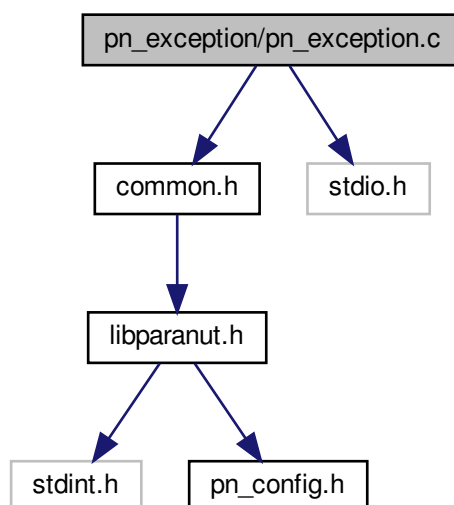
8.8 pn_exception/pn_exception.c File Reference

Contains (somewhat) architecture independent implementations of the [Exception Module](#) functions.

```
#include "common.h"
```

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

Include dependency graph for pn_exception.c:



Functions

- void [pn_exception_init](#) (void)
Initializes libparanut internal exception handling. Interrupts (not exceptions in general!) are disabled after. Should be called before using [pn_exception_set_handler](#)).
- int [pn_exception_set_handler](#) (void(*handler)(unsigned int cause, unsigned int program_counter, unsigned int mtval), unsigned int exception_code)
Set your own exception handler.
- void [pn_ecall](#) (void)
Raises an environment call exception.
- void [pn_interrupt_enable](#) (void)
Enables interrupts only.
- void [pn_interrupt_disable](#) (void)
Disables interrupts only.
- void [pn_progress_mepc](#) (void)
Sets program counter of the register which keeps the exception return adress to next instruction.

8.8.1 Detailed Description

Contains (somewhat) architecture independent implementations of the [Exception Module](#) functions.

The exception module is not exactly architecture independent since it implements a RISC-V exception table.

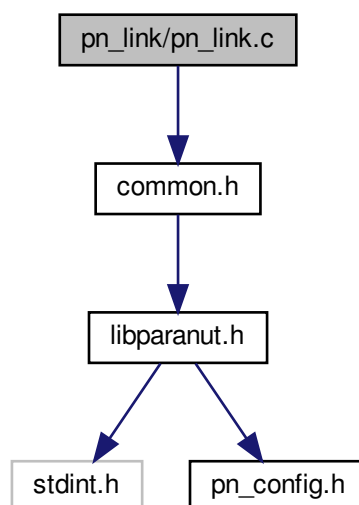
Todo Layer this better in later versions of libparanut.

8.9 pn_link/pn_link.c File Reference

Contains architecture independent implementations of the [Link Module](#) functions.

```
#include "common.h"
```

Include dependency graph for pn_link.c:



Functions

- [PN_CID pn_begin_linked](#) ([PN_NUMC](#) numcores)
Links a given number of CoPUs to the CePU.
- [PN_CID pn_begin_linked_m](#) ([PN_CMSK](#) coremask)
Links the CPUs specified in the coremask.
- [PN_CID pn_begin_linked_gm](#) ([PN_CMSK](#) *coremask_array, [PN_NUMG](#) array_size)
Links the CPUs specified in the coremask_array.
- [int pn_end_linked](#) (void)
Ends linked execution.

8.9.1 Detailed Description

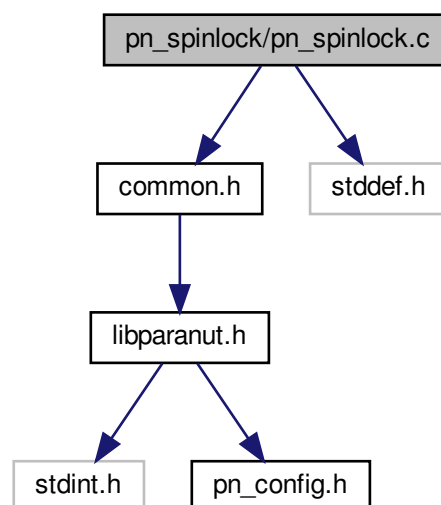
Contains architecture independent implementations of the [Link Module](#) functions.

8.10 pn_spinlock/pn_spinlock.c File Reference

Contains architecture independent implementations of the [Spinlock Module](#) functions.

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

Include dependency graph for pn_spinlock.c:



Functions

- int [pn_spinlock_init](#) ([_pn_spinlock](#) *spinlock)
Creates a lock.
- int [pn_spinlock_lock](#) ([_pn_spinlock](#) *spinlock)
Waits for a lock. Forever, if it must. Use with caution.
- int [pn_spinlock_trylock](#) ([_pn_spinlock](#) *spinlock)
Tries to acquire a lock. Nonblocking.
- int [pn_spinlock_unlock](#) ([_pn_spinlock](#) *spinlock)
Unlocks a lock.
- int [pn_spinlock_destroy](#) ([_pn_spinlock](#) *spinlock)
Destroys a lock.

8.10.1 Detailed Description

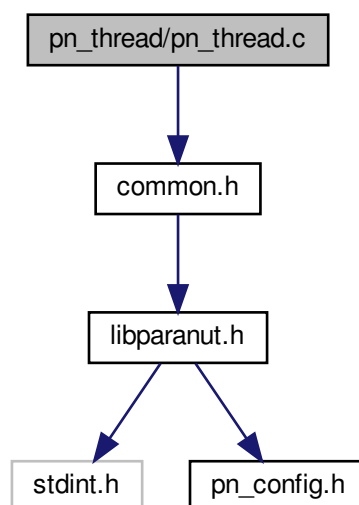
Contains architecture independent implementations of the [Spinlock Module](#) functions.

8.11 pn_thread/pn_thread.c File Reference

Contains architecture independent implementations of the [Thread Module](#) functions.

```
#include "common.h"
```

Include dependency graph for pn_thread.c:



Functions

- void [pn_thread_entry](#) ()
Function that has to be called for CoPUs at the end of the startup code.
- [PN_CID pn_begin_threaded](#) ([PN_NUMC](#) numcores)
Puts numcores CPUs in threaded mode.
- [PN_CID pn_begin_threaded_m](#) ([PN_CMSK](#) coremask)
Puts the CPUs specified in the coremask in threaded mode.
- [PN_CID pn_begin_threaded_gm](#) ([PN_CMSK](#) *coremask_array, [PN_NUMG](#) array_size)
Puts the CPUs specified in the coremask_array in threaded mode.
- int [pn_end_threaded](#) (void)
Ends threaded execution.

8.11.1 Detailed Description

Contains architecture independent implementations of the [Thread Module](#) functions.

Index

[__pn_spinlock](#), [53](#)

[_pn_spinlock](#)

[Typedefs](#), [16](#)

[Base Module](#), [22](#)

[pn_coreid](#), [23](#)

[pn_coreid_g](#), [23](#)

[pn_halt](#), [23](#)

[pn_halt_CoPU_gm](#), [24](#)

[pn_halt_CoPU_m](#), [25](#)

[pn_halt_CoPU](#), [24](#)

[pn_m2cap](#), [26](#)

[pn_m2cap_g](#), [26](#)

[pn_m3cap](#), [26](#)

[pn_m3cap_g](#), [27](#)

[pn_numcores](#), [28](#)

[pn_simulation](#), [28](#)

[pn_time_ns](#), [28](#)

[Cache Module](#), [37](#)

[pn_cache_disable](#), [37](#)

[pn_cache_enable](#), [38](#)

[pn_cache_flush](#), [38](#)

[pn_cache_flush_all](#), [39](#)

[pn_cache_init](#), [39](#)

[pn_cache_invalidate](#), [39](#)

[pn_cache_invalidate_all](#), [40](#)

[pn_cache_linesize](#), [40](#)

[pn_cache_size](#), [40](#)

[pn_cache_writeback](#), [40](#)

[pn_cache_writeback_all](#), [41](#)

[Comm_def](#), [51](#)

[Comm_glo](#), [52](#)

[common/common.h](#), [55](#)

[Error Codes](#), [18](#)

[PN_ERR_CACHE_LINESIZE](#), [18](#)

[PN_ERR_COPU](#), [18](#)

[PN_ERR_EXC](#), [19](#)

[PN_ERR_LOCKOCC](#), [19](#)

[PN_ERR_MATCH](#), [19](#)

[PN_ERR_NOIMP](#), [19](#)

[PN_ERR_PARAM](#), [19](#)

[PN_SUCCESS](#), [19](#)

[Exception Module](#), [42](#)

[pn_ecall](#), [42](#)

[pn_exception_set_handler](#), [42](#)

[pn_interrupt_disable](#), [43](#)

[pn_interrupt_enable](#), [43](#)

[pn_progress_mepc](#), [43](#)

[libparanut Compile Time Parameters](#), [49](#)

[PN_CACHE_LINESIZE](#), [49](#)

[PN_COMPILE_RAW](#), [49](#)

[PN_RWIDTH](#), [50](#)

[libparanut Helpers](#), [15](#)

[libparanut Modules](#), [21](#)

[libparanut.h](#), [56](#)

[Link Module](#), [29](#)

[pn_begin_linked](#), [29](#)

[pn_begin_linked_gm](#), [30](#)

[pn_begin_linked_m](#), [31](#)

[pn_end_linked](#), [32](#)

[Makefile](#), [60](#)

[Modes](#), [20](#)

[PN_CACHE_LINESIZE](#)

[libparanut Compile Time Parameters](#), [49](#)

[PN_CID](#)

[Typedefs](#), [16](#)

[PN_CMSK](#)

[Typedefs](#), [17](#)

[PN_COMPILE_RAW](#)

[libparanut Compile Time Parameters](#), [49](#)

[PN_ERR_CACHE_LINESIZE](#)

[Error Codes](#), [18](#)

[PN_ERR_COPU](#)

[Error Codes](#), [18](#)

[PN_ERR_EXC](#)

[Error Codes](#), [19](#)

[PN_ERR_LOCKOCC](#)

[Error Codes](#), [19](#)

[PN_ERR_MATCH](#)

[Error Codes](#), [19](#)

[PN_ERR_NOIMP](#)

[Error Codes](#), [19](#)

[PN_ERR_PARAM](#)

[Error Codes](#), [19](#)

[PN_NUMC](#)

[Typedefs](#), [17](#)

[PN_NUMG](#)

[Typedefs](#), [17](#)

[PN_RWIDTH](#)

[libparanut Compile Time Parameters](#), [50](#)

[PN_SUCCESS](#)

[Error Codes](#), [19](#)

[pn_base/pn_base.c](#), [63](#)

[pn_begin_linked](#)

[Link Module](#), [29](#)

[pn_begin_linked_gm](#)

- Link Module, 30
- pn_begin_linked_m
 - Link Module, 31
- pn_begin_threaded
 - Thread Module, 33
- pn_begin_threaded_gm
 - Thread Module, 34
- pn_begin_threaded_m
 - Thread Module, 35
- pn_cache/pn_cache.c, 65
- pn_cache/pn_cache_RV32I_buildscript.py, 66
- pn_cache_disable
 - Cache Module, 37
- pn_cache_enable
 - Cache Module, 38
- pn_cache_flush
 - Cache Module, 38
- pn_cache_flush_all
 - Cache Module, 39
- pn_cache_init
 - Cache Module, 39
- pn_cache_invalidate
 - Cache Module, 39
- pn_cache_invalidate_all
 - Cache Module, 40
- pn_cache_linesize
 - Cache Module, 40
- pn_cache_size
 - Cache Module, 40
- pn_cache_writeback
 - Cache Module, 40
- pn_cache_writeback_all
 - Cache Module, 41
- pn_config.h, 66
- pn_coreid
 - Base Module, 23
- pn_coreid_g
 - Base Module, 23
- pn_ecall
 - Exception Module, 42
- pn_end_linked
 - Link Module, 32
- pn_end_threaded
 - Thread Module, 35
- pn_exception/pn_exception.c, 67
- pn_exception_set_handler
 - Exception Module, 42
- pn_halt
 - Base Module, 23
- pn_halt_CoPU_gm
 - Base Module, 24
- pn_halt_CoPU_m
 - Base Module, 25
- pn_halt_CoPU
 - Base Module, 24
- pn_interrupt_disable
 - Exception Module, 43
- pn_interrupt_enable
 - Exception Module, 43
- pn_link/pn_link.c, 68
- pn_m2cap
 - Base Module, 26
- pn_m2cap_g
 - Base Module, 26
- pn_m3cap
 - Base Module, 26
- pn_m3cap_g
 - Base Module, 27
- pn_numcores
 - Base Module, 28
- pn_progress_mepc
 - Exception Module, 43
- pn_simulation
 - Base Module, 28
- pn_spinlock/pn_spinlock.c, 69
- pn_spinlock_destroy
 - Spinlock Module, 46
- pn_spinlock_init
 - Spinlock Module, 46
- pn_spinlock_lock
 - Spinlock Module, 47
- pn_spinlock_trylock
 - Spinlock Module, 47
- pn_spinlock_unlock
 - Spinlock Module, 48
- pn_thread/pn_thread.c, 70
- pn_thread_entry
 - Thread Module, 36
- pn_time_ns
 - Base Module, 28
- Spinlock Module, 45
 - pn_spinlock_destroy, 46
 - pn_spinlock_init, 46
 - pn_spinlock_lock, 47
 - pn_spinlock_trylock, 47
 - pn_spinlock_unlock, 48
- Thread Module, 33
 - pn_begin_threaded, 33
 - pn_begin_threaded_gm, 34
 - pn_begin_threaded_m, 35
 - pn_end_threaded, 35
 - pn_thread_entry, 36
- Typedefs, 16
 - _pn_spinlock, 16
 - PN_CID, 16
 - PN_CMSK, 17
 - PN_NUMC, 17
 - PN_NUMG, 17