

libparanut Unittest

Generated by Doxygen 1.8.13



# Contents

- 1 libparanut Unittest Documentation 1**
  - 1.1 Description 1
  - 1.2 Copyright 1
  - 1.3 HOWTO 2
  - 1.4 Also see ... 2
  
- 2 Todo List 3**
  
- 3 Module Index 5**
  - 3.1 Modules 5
  
- 4 File Index 7**
  - 4.1 File List 7
  
- 5 Module Documentation 9**
  - 5.1 Architecture Defines 9
    - 5.1.1 Detailed Description 9
    - 5.1.2 Macro Definition Documentation 9
      - 5.1.2.1 M2CAP\_MSK 9
      - 5.1.2.2 M3CAP\_MSK 9
  - 5.2 Test Case Return Values 10
    - 5.2.1 Detailed Description 10

<b>6 File Documentation</b>	<b>11</b>
6.1 libparanut_unittest.h File Reference	11
6.1.1 Detailed Description	12
6.1.2 Function Documentation	13
6.1.2.1 test_cache()	13
6.1.2.2 test_cap()	14
6.1.2.3 test_exception()	14
6.1.2.4 test_halt_CoPU()	14
6.1.2.5 test_link()	15
6.1.2.6 test_numcores()	15
6.1.2.7 test_thread()	15
6.1.2.8 test_time()	15
6.2 libparanut_unittest_main.c File Reference	16
6.2.1 Detailed Description	16
6.2.2 Macro Definition Documentation	18
6.2.2.1 TEST	18
6.3 libparanut_unittest_testcases.c File Reference	19
6.3.1 Detailed Description	20
6.3.2 Macro Definition Documentation	32
6.3.2.1 CPU_MSK	32
6.3.2.2 CPU_MSK_CHECK	32
6.3.2.3 LOOPS	33
6.3.2.4 NUMCORE_MIN	33
6.3.2.5 NUMCORES_CHECK	33
6.3.2.6 PLAUSIBLE_TIME	33
6.3.3 Function Documentation	33
6.3.3.1 test_cache()	34
6.3.3.2 test_cap()	34
6.3.3.3 test_exception()	34
6.3.3.4 test_halt_CoPU()	34
6.3.3.5 test_link()	35
6.3.3.6 test_numcores()	35
6.3.3.7 test_thread()	35
6.3.3.8 test_time()	35
<b>Index</b>	<b>37</b>

# Chapter 1

## libparanut Unittest Documentation

### 1.1 Description

This is a Unittest for the libparanut, a hardware abstraction layer for ParaNut architectures.

### 1.2 Copyright

Copyright 2019-2020 Anna Pfuetzner ([annakerstin.pfuetzner@gmail.com](mailto: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.

## 1.3 HOWTO

First, check this Unittests Makefile. In there, you will see a section called "System Configuration":

```
#System Configuration#####  
# Call clean when changing these!  
  
# System Parameters  
NUMCORES = 4  
M2CAP_MSK = 0x0000000F  
M3CAP_MSK = 0x00000001
```

It is necessary to chose the right parameters for your ParaNut configuration here, else the Unittest cannot check if the ParaNut gives correct data about itself. NUMCORES is about how many cores your system has. M2CAP\_MSK is a bitmask with the bits turned on that represent the cores which are able of running in Threaded Mode. M3CAP\_MSK is a bit mask with the bits turned on that represent the cores which are able of handling their own exceptions.

The parameters in these section should already be the same as the default configuration of the ParaNut, so if you didn't change anything in the config File of the ParaNut, you don't need to worry about this.

To run in SystemC simulation, execute:

```
make sim
```

for just the execution. To produce more debug information, execute:

```
make sim_dbg
```

This produces additional information, like a very full binary dump, a reduced dump, and a Waveform which you can open with GTKWave. All of that is available in the directory Debugging\_Aid (it aids debugging). For viewing the waveform, I have already prepared a standard view which proved to be very useful for debugging the libparanut. It can be found under Debugging\_Aid/waveview.gtkw. The file paranut.cfg in the same directory can be used for connecting GDB to simulation (see ParaNut Manual Apendix for instructions on how to do that).

For running on Zybo Z7020, execute:

```
make flash-z20-bit
```

Further explanations on this can be found in the documentation of module [Architecture Defines](#).

## 1.4 Also see ...

For further information on what exactly is being tested here, check the documentation of the libparanut itself and the ParaNut Manual.

**Todo** Test \_g functions too when they are actually implemented in libparanut.

## Chapter 2

# Todo List

### page [libparanut Unittest Documentation](#)

Test `_g` functions too when they are actually implemented in libparanut.

### Member [NUMCORES\\_CHECK](#)

If there's enough cores for `pn_numcores()` to be negative some day, this needs to be changed.

### Member [PLAUSIBLE\\_TIME](#)

If the ParaNut is getting faster in the future, this might need to change.

### Member [test\\_cache](#) (void)

I have no idea how I am supposed to test `pn_interrupt_enable()` and `pn_interrupt_disable()` at the current ParaNut implementation, since we do not have a working `mtimecmp` and `mtime` register yet. This may change in the future, though.

### Member [test\\_cap](#) (void)

Test `pn_m2cap_g()` when it is available.

Test `pn_m3cap_g()` when it is available.

### Member [test\\_exception](#) (void)

This needs changes in case there's more than one group of CPUs.

### Member [test\\_halt\\_CoPU](#) (void)

Test group function when it is available.

### Member [test\\_link](#) (void)

Group function test (as soon as implemented in libparanut).

### Member [test\\_thread](#) (void)

Test group functions when they are available.

Group function test (as soon as implemented in libparanut).

POSIX Threads



# Chapter 3

## Module Index

### 3.1 Modules

Here is a list of all modules:

Architecture Defines . . . . .	9
Test Case Return Values . . . . .	10



# Chapter 4

## File Index

### 4.1 File List

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

- [libparanut\\_unittest.h](#)  
Contains helpers and function prototypes of testcases . . . . . 11
- [libparanut\\_unittest\\_main.c](#)  
Contains main function which calls all the testcases . . . . . 16
- [libparanut\\_unittest\\_testcases.c](#)  
Contains testcase implementations . . . . . 19



# Chapter 5

## Module Documentation

### 5.1 Architecture Defines

Defines that give information about your ParaNut architecture.

- `#define NUMCORES`  
*Number of cores on your system (includes CePU).*
- `#define M2CAP_MSK`  
*Mask representing which cores are capable of Mode 2.*
- `#define M3CAP_MSK`  
*Mask representing which cores are capable of Mode 3.*

#### 5.1.1 Detailed Description

Defines that give information about your ParaNut architecture.

Since this Unittest is designed to be run on many different ParaNut implementations, it needs some information on your exact architecture. You have to set these things explicitly while compiling the test. If you don't, errors are thrown.

For learning how to set the defines during compilation, check the manual of your preprocessor/compiler.

#### 5.1.2 Macro Definition Documentation

##### 5.1.2.1 M2CAP\_MSK

```
#define M2CAP_MSK
```

Mask representing which cores are capable of Mode 2.

Make this as wide as your native register width. Only represent the first group (group number 0).

##### 5.1.2.2 M3CAP\_MSK

```
#define M3CAP_MSK
```

Mask representing which cores are capable of Mode 3.

Make this as wide as your native register width. Only represent the first group (group number 0).

## 5.2 Test Case Return Values

Defines and Typedef for Test Case Return Values.

- typedef int8\_t **TEST\_RET**  
*Renaming of int8\_t to mark clearly where a test return value is expected.*
- #define **TEST\_SUCCESS** ( 0)  
*Return value if test succeeded.*
- #define **TEST\_FAIL** (-1)  
*Return value if test failed.*
- #define **TEST\_SKIPPED** (-2)  
*Return value if test was not executed.*

### 5.2.1 Detailed Description

Defines and Typedef for Test Case Return Values.

## Chapter 6

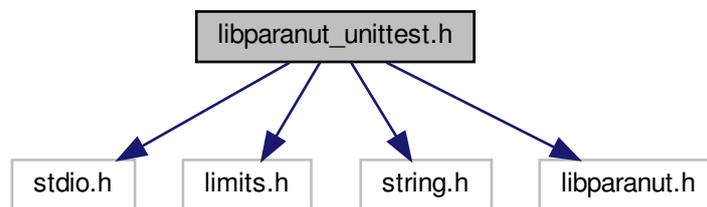
# File Documentation

### 6.1 libparanut\_unittest.h File Reference

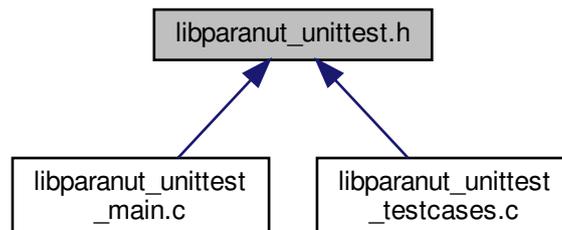
Contains helpers and function prototypes of testcases.

```
#include <stdio.h>
#include <limits.h>
#include <string.h>
#include "libparanut.h"
```

Include dependency graph for libparanut\_unittest.h:



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



## Macros

- `#define` `TERMNL` `"\n\r"`  
*Terminal newline, works on several platforms.*
- `#define` `NUMCORES`  
*Number of cores on your system (includes CePU).*
- `#define` `M2CAP_MSK`  
*Mask representing which cores are capable of Mode 2.*
- `#define` `M3CAP_MSK`  
*Mask representing which cores are capable of Mode 3.*

## Functions

- `TEST_RET` `test_time` (void)
- `TEST_RET` `test_numcores` (void)
- `TEST_RET` `test_cap` (void)
- `TEST_RET` `test_link` (void)
- `TEST_RET` `test_thread` (void)
- `TEST_RET` `test_halt_CoPU` (void)
- `TEST_RET` `test_cache` (void)
- `TEST_RET` `test_exception` (void)
- `TEST_RET` `test_spinlock` (void)
- `#define` `TEST_SUCCESS` ( 0)  
*Return value if test succeeded.*
- `#define` `TEST_FAIL` (-1)  
*Return value if test failed.*
- `#define` `TEST_SKIPPED` (-2)  
*Return value if test was not executed.*
- `typedef` `int8_t` `TEST_RET`  
*Renaming of `int8_t` to mark clearly where a test return value is expected.*

### 6.1.1 Detailed Description

Contains helpers and function prototypes of testcases.

```

1
103 /*Includes*****
104
105 #include <stdio.h>
106 #include <limits.h>
107 #include <string.h>
108 #include "libparanut.h"
109
110 /*Architecture Defines*****
111
112
113
114 /*
115 * The weird #if DOXYGEN is done because Doxygen won't document it otherwise.
116 * Sorry about that.
117 * If you find a more elegant solution, do not hesitate to put it in :)
118 */
119

```

```

144 #if DOXYGEN
145
146     #define NUMCORES
147
148 #endif
149
150 #ifndef NUMCORES
151
152     #define NUMCORES
153     #error NUMCORES undefined! Check "Architecture Defines" Documentation!
154
155 #endif
156
164 #if DOXYGEN
165
166     #define M2CAP_MSK
167
168 #endif
169
170 #ifndef M2CAP_MSK
171
172     #define M2CAP_MSK
173     #error M2CAP_MSK undefined! Check "Architecture Defines" Documentation!
174
175 #endif
176
184 #if DOXYGEN
185
186     #define M3CAP_MSK
187
188 #endif
189
190 #ifndef M3CAP_MSK
191
192     #define M3CAP_MSK
193     #error M3CAP_MSK undefined! Check "Architecture Defines" Documentation!
194
195 #endif
196
205 /*Helpers*****/
206
211 #define TERMINL          "\n\r"
212
232 typedef int8_t TEST_RET;
233
238 #define TEST_SUCCESS      ( 0)
239
244 #define TEST_FAIL        (-1)
245
250 #define TEST_SKIPPED     (-2)
251
260 /*Test Case Prototypes*****/
261
262 TEST_RET test_time(void);
263 TEST_RET test_numcores(void);
264 TEST_RET test_cap(void);
265 TEST_RET test_link(void);
266 TEST_RET test_thread(void);
267 TEST_RET test_halt_CoPU(void);
268 TEST_RET test_cache(void);
269 TEST_RET test_exception(void);
270 TEST_RET test_spinlock(void);
271
272 /*EOF*****/
273

```

## 6.1.2 Function Documentation

### 6.1.2.1 test\_cache()

```

TEST_RET test_cache (
    void )

```

Tests all functions in exception module.

Assumes exception module to have been initialized before.

**Todo** I have no idea how I am supposed to test `pn_interrupt_enable()` and `pn_interrupt_disable()` at the current ParaNut implementation, since we do not have a working `mtimecmp` and `mtime` register yet. This may change in the future, though.

#### 6.1.2.2 test\_cap()

```
TEST_RET test_cap (  
    void )
```

Tests all functions in link module.

Uses `pn_numcores()`.

**Todo** Test `pn_m2cap_g()` when it is available.

**Todo** Test `pn_m3cap_g()` when it is available.

#### 6.1.2.3 test\_exception()

```
TEST_RET test_exception (  
    void )
```

Tests all functions in spinlock module.

Implicitly tests `pn_begin_threaded()` and `pn_end_threaded()`.

**Todo** This needs changes in case there's more than one group of CPUs.

#### 6.1.2.4 test\_halt\_CoPU()

```
TEST_RET test_halt_CoPU (  
    void )
```

Tests all functions in cache module. Also implicitly tests `pn_simulation()`.

Assumes cache module to have been initialized before.

Testing the cache is skipped in ParaNut simulation since it is excruciatingly slow. Also tests `pn_simulation()`. This means, if you're not in a simulation and this testcase is skipped, something is wrong with `pn_simulation()`.

**Todo** Test group function when it is available.

### 6.1.2.5 test\_link()

```
TEST_RET test_link (  
    void )
```

Tests all functions in thread module.

Assumes that entry point for CoPUs is set correctly in the startup code.

Uses pn\_m2cap().

**Todo** Group function test (as soon as implemented in libparanut).

### 6.1.2.6 test\_numcores()

```
TEST_RET test_numcores (  
    void )
```

Tests functions pn\_m2cap() and pn\_m3cap().

### 6.1.2.7 test\_thread()

```
TEST_RET test_thread (  
    void )
```

Tests function pn\_halt\_CoPU().

This test was being put in here because threaded mode has to work properly before this can be tested.

**Todo** Test group functions when they are available.

**Todo** Group function test (as soon as implemented in libparanut).

**Todo** POSIX Threads

### 6.1.2.8 test\_time()

```
TEST_RET test_time (  
    void )
```

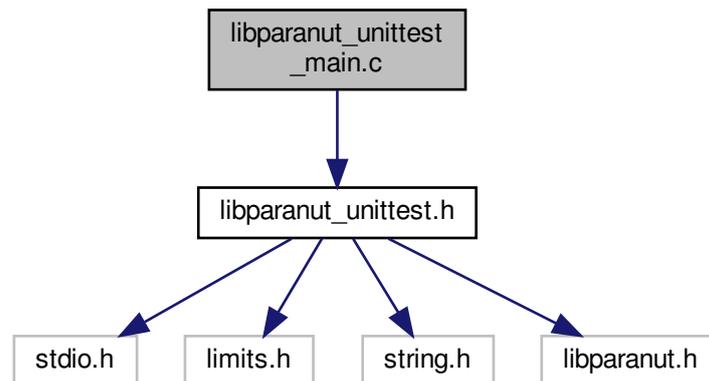
Tests function pn\_numcores().

## 6.2 libparanut\_unittest\_main.c File Reference

Contains main function which calls all the testcases.

```
#include "libparanut_unittest.h"
```

Include dependency graph for libparanut\_unittest\_main.c:



### Macros

- `#define TEST(x)`  
*Helper for ending the test when execution failed.*

### 6.2.1 Detailed Description

Contains main function which calls all the testcases.

Execution is ended automatically when a testcase fails. This is because some testcases need other functionality to work perfectly before testing the actual function.

```

1 /*
2  * Copyright 2019-2020 Anna Pfuetzner (<annakerstin.pfuetzner@gmail.com>)
3  *
4  * Redistribution and use in source and binary forms, with or without
5  * modification, are permitted provided that the following conditions are met:
6  *
7  * 1. Redistributions of source code must retain the above copyright notice,
8  * this list of conditions and the following disclaimer.
9  *
10 * 2. Redistributions in binary form must reproduce the above copyright notice,
11 * this list of conditions and the following disclaimer in the documentation
12 * and/or other materials provided with the distribution.
13 *
14 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
15 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
16 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
17 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
18 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
19 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
20 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
21 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN

```

```

22 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
23 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
24 * POSSIBILITY OF SUCH DAMAGE.
25 */
26
40 /*Includes*****
41
42 #include "libparanut_unittest.h"
43
44 /*Helpers*****
45
50 #define TEST(x)      printf("###STARTING %s###" TERMNL, #x);          \
51                    if ((ret = x()) == TEST_FAIL)                  \
52                    {                                              \
53                        printf("###TESTCASE FAILED###" TERMNL TERMNL); \
54                        printf("###Unsuccessful End of Test :(###"    \
55                            TERMNL TERMNL); \
56                        return -1;                                    \
57                    }                                              \
58                    else if (ret == TEST_SKIPPED)                  \
59                    {                                              \
60                        printf("###TESTCASE SKIPPED###" TERMNL TERMNL); \
61                    }                                              \
62                    else                                           \
63                    {                                              \
64                        printf("###TESTCASE SUCCESS###" TERMNL TERMNL); \
65                    }                                              \
66
67 /*Main Function*****
68
73 int main()
74 {
75
76 #if !(defined PN_WITH_BASE)          \
77     && !(defined PN_WITH_CACHE)      \
78     && !(defined PN_WITH_LINK)       \
79     && !(defined PN_WITH_THREAD)     \
80     && !(defined PN_WITH_EXCEPTION)  \
81     && !(defined PN_WITH_SPINLOCK)   \
82
83     printf("###No Modules were compiled in libparanut, cannot start test :(###"
84           TERMNL);
85     return TEST_FAIL;
86 #else
87 /*
88  * locals
89  */
90     TEST_RET    ret;          /* saves return value - see helper TEST() */
91
92 #endif
93
94 #endif
95
96 #ifdef PN_WITH_BASE
97     long long int start, end; /* start and end time of test */
98 #endif /* PN_WITH_BASE */
99
100     printf("###Welcome to libparanut Unittest###" TERMNL TERMNL);
101
102     /*
103     * Initialize all of libparanut Modules that need initializing. This is sorta
104     * untestable by itself. Things will go wrong in the unit test itself if
105     * something's wrong here, though.
106     */
107
108 #ifdef PN_WITH_EXCEPTION
109     printf("###Initializing exception module ...###" TERMNL TERMNL);
110     pn_exception_init();
111 #endif /* PN_WITH_EXCEPTION */
112
113 #ifdef PN_WITH_CACHE
114     printf("###Initializing cache module ...###" TERMNL TERMNL);
115     if (pn_cache_init() != PN_SUCCESS)
116     {
117         printf("Error in pn_cache_init(). We can not proceed with this test."
118               TERMNL
119               TERMNL);
120         printf("###Unsuccessful End of Test :(###" TERMNL TERMNL);
121         return -1;
122     }
123 #endif /* PN_WITH_CACHE */
124 #ifdef PN_WITH_BASE
125     TEST(test_time)
126     start = pn_time_ns();

```

```

135
140     TEST(test_numcores)
141
142
146     TEST(test_cap)
147
148 #endif /* PN_WITH_BASE */
149
150 #if defined PN_WITH_LINK && defined PN_WITH_BASE
151     TEST(test_link)
158
159
160 #endif /* defined PN_WITH_LINK && defined PN_WITH_BASE */
161
162 #if defined PN_WITH_THREAD && defined PN_WITH_BASE
163     TEST(test_thread)
172
173
174 #endif /* defined PN_WITH_THREAD && defined PN_WITH_BASE */
175
176 #if defined PN_WITH_BASE && defined PN_WITH_THREAD
177     TEST(test_halt_CoPU)
185
186
187 #endif /* defined PN_WITH_BASE && defined PN_WITH_THREAD */
188
189 #ifdef PN_WITH_CACHE
190     TEST(test_cache)
203
204
205 #endif /* PN_WITH_CACHE */
206
207 #ifdef PN_WITH_EXCEPTION
208     TEST(test_exception)
220
221
222 #endif /* PN_WITH_EXCEPTION */
223
224 #if defined PN_WITH_SPINLOCK && defined PN_WITH_THREAD
225     TEST(test_spinlock)
234
235
236 #endif /* defined PN_WITH_SPINLOCK && defined PN_WITH_THREAD */
237
238 #ifdef PN_WITH_BASE
239     end = pn_time_ns();
240     printf("###Execution time of test: %lld ms###" TERMNL TERMNL,
241           (end - start) / 1000000);
242 #endif /* PN_WITH_BASE */
243
244     printf("###Successfull End of Test :###" TERMNL TERMNL);
245
246     return TEST_SUCCESS;
247 }
248
249 /*EOF*****

```

## 6.2.2 Macro Definition Documentation

### 6.2.2.1 TEST

```
#define TEST(
    x )
```

#### Value:

```

printf("###STARTING %s###" TERMNL, #x);
    if ((ret = x()) == TEST_FAIL)
    {
        printf("###TESTCASE FAILED###" TERMNL TERMNL);
        printf("###Unsuccessful End of Test :(###"
              TERMNL TERMNL);
        return -1;
    }

```

```

    }
    else if (ret == TEST_SKIPPED)
    {
        printf("###TESTCASE SKIPPED###" TERMNL TERMNL);
    }
    else
    {
        printf("###TESTCASE SUCCESS###" TERMNL TERMNL);
    }
}

```

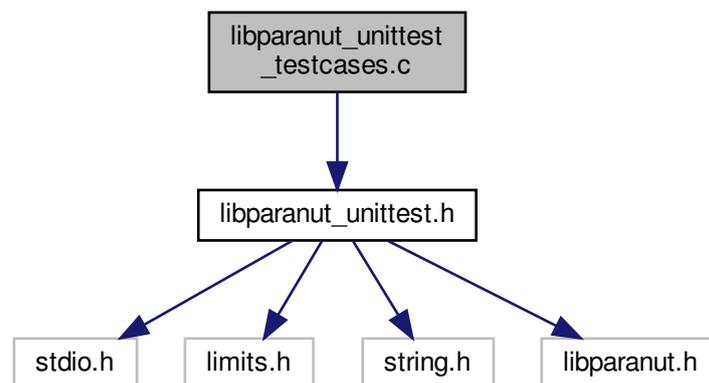
Helper for ending the test when execution failed.

## 6.3 libparanut\_unittest\_testcases.c File Reference

Contains testcase implementations.

```
#include "libparanut_unittest.h"
```

Include dependency graph for libparanut\_unittest\_testcases.c:



### Macros

- `#define NUMCORE_MIN 2`  
*Minimal number of cores that shall be linked/threaded together.*
- `#define CPU_MSK 0b1`  
*Bitmask of cores that shall be linked/threaded together.*
- `#define LOOPS 4`  
*Number of loops for testing linked/threaded Mode.*
- `#define PLAUSIBLE_TIME 30000`  
*Number of ns that are considered plausible between two timer gets.*
- `#define ARRAYLENGTH 100`  
*Length of the global test array (s\_testarray). Must be divisible by 10 and by NUMCORE\_MIN.*
- `#define NUMCORES_CHECK`  
*Checks if minimum number of cores is available.*
- `#define CPU_MSK_CHECK`  
*Checks if at least two Mode 2 capable cores are available.*

## Functions

- [TEST\\_RET test\\_time](#) (void)
- [TEST\\_RET test\\_numcores](#) (void)
- [TEST\\_RET test\\_cap](#) (void)
- [TEST\\_RET test\\_link](#) (void)
- [TEST\\_RET test\\_thread](#) (void)
- [TEST\\_RET test\\_halt\\_CoPU](#) (void)
- [TEST\\_RET test\\_cache](#) (void)
- [TEST\\_RET test\\_exception](#) (void)
- [TEST\\_RET test\\_spinlock](#) (void)

### 6.3.1 Detailed Description

Contains testcase implementations.

```

1 /*
2  * Copyright 2019-2020 Anna Pfuetzner (<annakerstin.pfuetzner@gmail.com>)
3  *
4  * Redistribution and use in source and binary forms, with or without
5  * modification, are permitted provided that the following conditions are met:
6  *
7  * 1. Redistributions of source code must retain the above copyright notice,
8  * this list of conditions and the following disclaimer.
9  *
10 * 2. Redistributions in binary form must reproduce the above copyright notice,
11 * this list of conditions and the following disclaimer in the documentation
12 * and/or other materials provided with the distribution.
13 *
14 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
15 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
16 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
17 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
18 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
19 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
20 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
21 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
22 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
23 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
24 * POSSIBILITY OF SUCH DAMAGE.
25 */
26
27 /*Includes*****
28 #include "libparanut_unittest.h"
29
30 /*Local Defines*****
31
32 /* TODO Documentation */
33
34 #define NUMCORE_MIN      2
35
36 #define CPU_MSK          0b11
37
38 #define LOOPS            4
39
40 #define PLAUSIBLE_TIME  30000
41
42 #define ARRAYLENGTH     100
43
44 #define NUMCORES_CHECK  if (pn_numcores() < NUMCORE_MIN)
45 {
46     printf("  This Testcase demands at least 2 cores."
47           TERMNL);
48     return TEST_SKIPPED;
49 }
50
51 #define CPU_MSK_CHECK   if ((pn_m2cap() & CPU_MSK) != CPU_MSK)
52 {
53     printf("  This Testcase demands core 0 and 1 to "
54           "be capable of Mode 2." TERMNL);
55     return TEST_SKIPPED;
56 }
57
58 */

```

```

115 * Weak definitions of functions called in linked_threaded_test().
116 */
117 #if !(defined DOXYGEN)
118 #if !(defined PN_WITH_LINK)
119 PN_CID pn_begin_linked(PN_NUMC numcores) { return 0; }
120 PN_CID pn_begin_linked_m(PN_CMSK coremask) { return 0; }
121 int pn_end_linked(void) { return 0; }
122 #endif /* !(defined PN_WITH_LINK) */
123 #if !(defined PN_WITH_THREAD)
124 PN_CID pn_begin_threaded(PN_NUMC numcores) { return 0; }
125 PN_CID pn_begin_threaded_m(PN_CMSK coremask) { return 0; }
126 int pn_end_threaded(void) { return 0; }
127 #endif /* !(defined PN_WITH_THREAD) */
128 #endif /* !(defined DOXYGEN) */
129
130 /*Variables*****
131
132 #if defined PN_WITH_SPINLOCK && defined PN_WITH_THREAD
133
134 static int s_testarray[ARRAYLENGTH];
135 #endif /* defined PN_WITH_SPINLOCK && defined PN_WITH_THREAD */
136
137 #ifdef PN_WITH_EXCEPTION
138
139 static int s_exc_var;
140 #endif /* PN_WITH_EXCEPTION */
141
142 /*Static Functions*****
143
144 #if ((defined PN_WITH_LINK) || (defined PN_WITH_THREAD)) && defined PN_WITH_BASE
145
146 static void set_arrays(int *sum, int *a, int *b)
147 {
148     /*
149     * locals
150     */
151     int i; /* loop counting variable */
152
153     for (i = 0; i < LOOPS; i++)
154     {
155         sum[i] = 0;
156
157         /* also change check_sum() when changing this */
158         a[i] = 1;
159         b[i] = 1;
160     }
161
162     return;
163 }
164
165 /*-----*/
166
167 static void calc_sum(PN_CID cid, int *sum, int *a, int *b)
168 {
169     /*
170     * locals
171     */
172     int i; /* loop counting variable */
173
174     for (i = cid; i < LOOPS; i += NUMCORE_MIN)
175         sum[i] = a[i] + b[i];
176
177     return;
178 }
179
180 /*-----*/
181
182 static TEST_RET check_sum(int *sum)
183 {
184     /*
185     * locals
186     */
187     int i; /* loop counting variable */
188
189     for (i = 0; i < LOOPS; i++)
190     {
191         /* sum should be two since a and b are filled with 1s */
192         if (sum[i] != 2)
193             return TEST_FAIL;
194     }
195
196     return TEST_SUCCESS;
197 }
198
199 /*-----*/
200
201 static TEST_RET linked_threaded_test(char *funcname, PN_CID (*funcp)())
202 {

```

```

217 /*
218  * locals
219  */
220 static int    sum[LOOPS], a[LOOPS], b[LOOPS];
221              /* sum is sum of a and b          */
222 PN_CID       cid;          /* core ID          */
223 int          err;         /* error            */
224 int          i;           /* loop counter     */
225
226 printf("  Test %s." TERMNL, funcname);
227
228 /* fill in the arrays */
229 set_arrays(sum, a, b);
230
231 /* print some debug information */
232 printf(TERMNL);
233 printf("    Sum array before calculation:" TERMNL);
234 for (i = 0; i < LOOPS; i++)
235 {
236     printf("        sum[%d] = %d" TERMNL, i, sum[i]);
237 }
238 printf(TERMNL);
239
240 /* begin linked or threaded mode */
241 if ((funcp == &pn_begin_linked) || (funcp == &pn_begin_threaded))
242 {
243     cid = funcp(NUMCORE_MIN);
244 }
245 else if ((funcp == &pn_begin_linked_m) || (funcp == &pn_begin_threaded_m))
246 {
247     cid = funcp(CPU_MSK);
248 }
249 else
250 {
251     printf("  You passed a not yet implemented function to subtest "
252           "linked_threaded_test()" TERMNL);
253     return TEST_FAIL;
254 }
255
256 /* conditional jump doesn't matter if we didn't even go into linked mode */
257 if (cid < 0)
258 {
259     printf("  Failure of function %s." TERMNL, funcname);
260     return TEST_FAIL;
261 }
262
263 /* set sum to sum of a and b */
264 calc_sum(cid, sum, a, b);
265
266
267 /* end linked or threaded mode */
268 if ((funcp == &pn_begin_linked) || (funcp == &pn_begin_linked_m))
269 {
270     err = pn_end_linked();
271     if (err)
272     {
273         printf("  Failure of function pn_end_linked()." TERMNL);
274         return TEST_FAIL;
275     }
276 }
277 else
278 {
279     err = pn_end_threaded();
280     if (err)
281     {
282         printf("  Failure of function pn_end_threaded()." TERMNL);
283         return TEST_FAIL;
284     }
285 }
286
287 /* print some debug information */
288 if (cid == 0)
289 {
290     printf("    Sum array after calculation:" TERMNL);
291     for (i = 0; i < LOOPS; i++)
292     {
293         printf("        sum[%d] = %d" TERMNL, i, sum[i]);
294     }
295     printf(TERMNL);
296 }
297
298 /* check the sum array */
299 if (check_sum(sum) == TEST_FAIL)
300 {
301     printf("  Failure of calculation in chosen mode." TERMNL);
302     return TEST_FAIL;
303 }

```

```

304
305     return TEST_SUCCESS;
306 }
307
308 #endif /* ((defined PN_WITH_LINK) || (defined PN_WITH_THREAD)) */
309 /*      && defined PN_WITH_BASE          */
310
311 /*-----*/
312
313 #if defined PN_WITH_SPINLOCK && defined PN_WITH_THREAD
314
315 static void print_testarray(void)
316 {
317     /*
318     * locals
319     */
320     int i;
321
322     for (i = 0; i < (ARRAYLENGTH / 10); i++)
323     {
324         printf("      %i %i %i %i %i %i %i %i %i %i" TERMNL,
325                s_testarray[(i * 10) + 0],
326                s_testarray[(i * 10) + 1],
327                s_testarray[(i * 10) + 2],
328                s_testarray[(i * 10) + 3],
329                s_testarray[(i * 10) + 4],
330                s_testarray[(i * 10) + 5],
331                s_testarray[(i * 10) + 6],
332                s_testarray[(i * 10) + 7],
333                s_testarray[(i * 10) + 8],
334                s_testarray[(i * 10) + 9]);
335     }
336
337     return;
338 }
339
340 #endif /* defined PN_WITH_SPINLOCK && defined PN_WITH_THREAD */
341
342 /*-----*/
343
344 #ifndef PN_WITH_EXCEPTION
345
346 static void handler(unsigned int cause,
347                   unsigned int program_counter,
348                   unsigned int mtval)
349 {
350     printf("      Hello, this is the exception handler!" TERMNL);
351
352     if ((cause < 8) || (cause > 11))
353     {
354         printf("      Cause was not correctly passed to the handler." TERMNL);
355         return;
356     }
357
358     printf("      Changing variable now." TERMNL);
359     s_exc_var = 1;
360     printf("      Setting exception program counter to next instruction."
361            TERMNL);
362     pn_progress_mepc();
363     printf("      Returning ..." TERMNL);
364
365     return;
366 }
367
368 #endif /* PN_WITH_EXCEPTION */
369
370 /*-----*/
371
372 #ifdef PN_WITH_CACHE
373
374 static volatile TEST_RET invalidate(int (*invalidate_function)())
375 {
376     /*
377     * locals
378     */
379     static int testvar = 3;
380
381     /* disable cache */
382     printf("      Disable Cache." TERMNL);
383     pn_cache_disable();
384
385     /* value 1 stands in memory now */
386     printf("      Give test variable a value of 1." TERMNL);
387     testvar = 1;
388     printf("      Value of variable is now %i." TERMNL, testvar);
389
390     /* enable cache again */

```

```

391 printf("    Enable Cache." TERMNL);
392 pn_cache_enable();
393
394 /* value 2 stands in cache now */
395 printf("    Give test variable a value of 2." TERMNL);
396 testvar = 2;
397 printf("    Value of variable is now %i." TERMNL, testvar);
398
399 /* cache invalidate */
400 printf("    Invalidate cache." TERMNL);
401 if (*invalidate_function == pn_cache_invalidate)
402     invalidate_function(&testvar, 0);
403 else if (*invalidate_function == pn_cache_invalidate_all)
404     invalidate_function();
405 else
406 {
407     printf("    The sub test invalidate() was given a wrong function."
408           TERMNL);
409 }
410
411 /* read variable -> should be old value */
412 printf("    Variable now has a value of %d, should have value 1." TERMNL,
413       testvar);
414 if (testvar != 1)
415     return TEST_FAIL;
416 return TEST_SUCCESS;
417 }
418 }
419
420 /*-----*/
421
422 static TEST_RET writeback(int (*writeback_function)())
423 {
424     /*
425     * locals
426     */
427     static int testvar = 0;
428
429     /* disable cache */
430     printf("    Disable Cache." TERMNL);
431     pn_cache_disable();
432
433     /* value 1 stands in memory now */
434     printf("    Give test variable a value of 1." TERMNL);
435     testvar = 1;
436
437     /* enable cache again */
438     printf("    Enable Cache." TERMNL);
439     pn_cache_enable();
440
441     /* value 2 stands in cache now */
442     printf("    Give test variable a value of 2." TERMNL);
443     testvar = 2;
444
445     /* cache writeback */
446     printf("    Write back cache." TERMNL);
447     if (*writeback_function == pn_cache_writeback)
448         writeback_function(&testvar, 0);
449     else if (*writeback_function == pn_cache_writeback_all)
450         writeback_function();
451     else
452     {
453         printf("    The sub test writeback() was given a wrong function."
454               TERMNL);
455     }
456
457     /* disable cache */
458     printf("    Disable Cache." TERMNL);
459     pn_cache_disable();
460
461     /* read variable -> should be new value */
462     printf("    Variable now has a value of %d, should have value 2." TERMNL,
463       testvar);
464     if (testvar != 2)
465         return TEST_FAIL;
466     return TEST_SUCCESS;
467 }
468 }
469
470 /*-----*/
471
472 static TEST_RET flush(int (*flush_function)())
473 {
474     /*
475     * locals
476     */
477     static int testvar = 0;

```

```

478
479 /* disable cache */
480 printf("      Disable Cache." TERMNL);
481 pn_cache_disable();
482
483 /* value 1 stands in memory now */
484 printf("      Give test variable a value of 1." TERMNL);
485 testvar = 1;
486
487 /* enable cache again */
488 printf("      Enable Cache." TERMNL);
489 pn_cache_enable();
490
491 /* value 2 stands in cache now */
492 printf("      Give test variable a value of 2." TERMNL);
493 testvar = 2;
494
495 /* cache flush */
496 printf("      Flush cache." TERMNL);
497 if (*flush_function == pn_cache_flush)
498     flush_function(&testvar, 0);
499 else if (*flush_function == pn_cache_flush_all)
500     flush_function();
501 else
502 {
503     printf("      The sub test flush() was given a wrong function."
504           TERMNL);
505 }
506
507 /* disable cache */
508 printf("      Disable Cache." TERMNL);
509 pn_cache_disable();
510
511 /* read variable -> should be new value */
512 printf("      Variable now has a value of %d, should have value 2." TERMNL,
513        testvar);
514 if (testvar != 2)
515     return TEST_FAIL;
516 return TEST_SUCCESS;
517 }
518 }
519
520 #endif /* PN_WITH_CACHE */
521
522 /*Test Cases*****
523
524 #ifdef PN_WITH_BASE
525
526 TEST_RET test_time(void)
527 {
528     /*
529     * locals
530     */
531     long long int start, end;          /* start and end time          */
532
533     /*
534     * Read one time before actual measurement because first time takes the
535     * longest and is therefore not representantive.
536     */
537
538     start = pn_time_ns();
539
540     /*
541     * Actual measurement starts here.
542     */
543
544     start = pn_time_ns();
545     end   = pn_time_ns();
546
547     printf("      Start time: %lli" TERMNL, start);
548     printf("      End time:   %lli" TERMNL, end);
549
550     if (end == start)
551         goto _implausible;
552
553     if ((start > end) && ((LLONG_MAX - (start - end)) > PLAUSIBLE_TIME))
554         goto _implausible;
555
556     if ((start < end) && ((end - start) > PLAUSIBLE_TIME))
557         goto _implausible;
558
559     return TEST_SUCCESS;
560
561 _implausible:
562     printf("      Implausible." TERMNL);
563     return TEST_FAIL;
564 }

```

```

565
566 /*-----*/
567
568 TEST_RET test_numcores(void)
569 {
570     /*
571     * locals
572     */
573     PN_NUMC numc;                /* number of cores */
574
575     printf(" Test pn_numcores()." TERMNL);
576     numc = pn_numcores();
577     if (pn_numcores() != NUMCORES)
578     {
579         printf(" NUMCORES was %i, but pn_numcores() returned %i." TERMNL,
580                NUMCORES, (int) numc);
581         return TEST_FAIL;
582     }
583     return TEST_SUCCESS;
584 }
585
586 /*-----*/
587
588 TEST_RET test_cap(void)
589 {
590     /*
591     * locals
592     */
593     PN_CMSK cmsk;                /* core mask */
594
595     printf(" Test pn_m2cap()." TERMNL);
596     cmsk = pn_m2cap();
597     if (cmsk != M2CAP_MSK)
598     {
599         printf(" M2CAP_MSK was %u, but pn_m2cap() returned %u." TERMNL,
600                M2CAP_MSK, (unsigned int) cmsk);
601         return TEST_FAIL;
602     }
603
604     printf(" Test pn_m3cap()." TERMNL);
605     cmsk = pn_m3cap();
606     if (cmsk != M3CAP_MSK)
607     {
608         printf(" M3CAP_MSK was %u, but pn_m3cap() returned %u." TERMNL,
609                M3CAP_MSK, (unsigned int) cmsk);
610         return TEST_FAIL;
611     }
612
613     return TEST_SUCCESS;
614 }
615
616
617 #endif /* PN_WITH_BASE */
618
619 /*-----*/
620
621 #if defined PN_WITH_LINK && defined PN_WITH_BASE
622
623 TEST_RET test_link(void)
624 {
625     /*
626     * locals
627     */
628     int result;                /* result of subtest */
629
630     /* check if the test case is actually doable on current architecture */
631     NUMCORES_CHECK;
632
633     /*
634     * linked mode, method 1
635     */
636
637     printf(TERMNL);
638     result = linked_threaded_test("pn_begin_linked()", &pn_begin_linked);
639
640     if (result == TEST_FAIL)
641         return TEST_FAIL;
642
643     /*
644     * linked mode, method 2
645     */
646
647     result = linked_threaded_test("pn_begin_linked_m()", &pn_begin_linked_m);
648
649     if (result == TEST_FAIL)
650         return TEST_FAIL;
651
652     /*
653     */
654 }
655
656 #endif

```

```

660  * linked mode, method 3
661  */
662
663  return TEST_SUCCESS;
664 }
665
666 #endif /* defined PN_WITH_LINK && defined PN_WITH_BASE */
667
668 /*-----*/
669
670 #if defined PN_WITH_THREAD && defined PN_WITH_BASE
671
672 TEST_RET test_thread(void)
673 {
674     /*
675     * locals
676     */
677     int result;                /* result of subtest */
678
679     /* check if the test case is actually doable on current architecture */
680     CPU_MSK_CHECK;
681
682     /*
683     * threaded mode, method 1
684     */
685     printf(TERMNL);
686     result = linked_threaded_test("pn_begin_threaded()", &pn_begin_threaded);
687
688     if (result == TEST_FAIL)
689         return TEST_FAIL;
690
691     /*
692     * threaded mode, method 2
693     */
694     result = linked_threaded_test("pn_begin_threaded_m()", &pn_begin_threaded_m);
695
696     if (result == TEST_FAIL)
697         return TEST_FAIL;
698
699     /*
700     * threaded mode, method 3
701     */
702     return TEST_SUCCESS;
703 }
704
705 #endif /* defined PN_WITH_THREAD && defined PN_WITH_BASE */
706
707 /*-----*/
708
709 #if defined PN_WITH_BASE && defined PN_WITH_THREAD
710
711 TEST_RET test_halt_CoPU(void)
712 {
713     /*
714     * locals
715     */
716     PN_CID    cid;                /* core ID */
717     static int s_counter = 0;     /* counter touched by CoPUs */
718     int       counter_copy_1, counter_copy_2; /* counter copies */
719     int       i;                 /* loop counter */
720     int       err;               /* error value */
721
722     /* check if the test case is actually doable on current architecture */
723     CPU_MSK_CHECK;
724
725     /*
726     * pn_halt_CoPU()
727     */
728     printf("    Test pn_halt_CoPU()." TERMNL);
729
730     cid = pn_begin_threaded(NUMCORE_MIN);
731
732     if (cid == 0)
733         printf("        Threaded Mode started successfully." TERMNL);
734
735     if (cid != 0)
736         while (1)
737             s_counter++;
738     printf("        CoPUs are counting a static counter now." TERMNL);
739
740     for (i = 1; i < NUMCORE_MIN; i++)
741     {
742         if ((err = pn_halt_CoPU((PN_CID)i)) != PN_SUCCESS)

```

```

767     {
768         printf("    pn_halt_CoPU() returned error %d." TERMNL, err);
769         return TEST_FAIL;
770     }
771 }
772 printf("    Tried to halt them. Check if they are still counting." TERMNL);
773
774 /* since all CoPUs should be disabled, the counter should not change */
775 counter_copy_1 = s_counter;
776 pn_time_ns();
777 counter_copy_2 = s_counter;
778
779 if (counter_copy_1 != counter_copy_2)
780 {
781     printf("    The CoPUs have not been disabled!." TERMNL);
782     return TEST_FAIL;
783 }
784 printf("    They aren't. Good." TERMNL);
785
786 /*
787 * pn_halt_CoPU_m()
788 */
789
790 printf("    Test pn_halt_CoPU_m()." TERMNL);
791
792 cid = pn_begin_threaded_m(CPU_MSK);
793 if (cid == 0)
794     printf("    Threaded Mode started successfully." TERMNL);
795
796 if (cid != 0)
797     while (1)
798         s_counter++;
799 printf("    CoPUs are counting a static counter now." TERMNL);
800
801 if ((err = pn_halt_CoPU_m((CPU_MSK & 0xFFFFFFFF))) != PN_SUCCESS)
802 {
803     printf("    pn_halt_CoPU_m() returned error %d." TERMNL, err);
804     return TEST_FAIL;
805 }
806 printf("    Tried to halt them. Check if they are still counting." TERMNL);
807
808 /* since all CoPUs should be disabled, the counter should not change */
809 counter_copy_1 = s_counter;
810 pn_time_ns();
811 counter_copy_2 = s_counter;
812
813 if (counter_copy_1 != counter_copy_2)
814 {
815     printf("    The CoPUs have not been disabled!." TERMNL);
816     return TEST_FAIL;
817 }
818 printf("    They aren't. Good." TERMNL);
819
820 return TEST_SUCCESS;
821 }
822
823 #endif /* defined PN_WITH_BASE && defined PN_WITH_THREAD */
824
825 /*-----*/
826
827 #ifdef PN_WITH_CACHE
828 TEST_RET test_cache(void)
829 {
830     /*
831     * locals
832     */
833     TEST_RET ret;
834
835     /* note about skipping some parts on simulation */
836     if (pn_simulation())
837     {
838         printf(TERMNL);
839         printf("    Testing the pn_cache_...() functions is skipped in ParaNut"
840             " simulation since it is exruciatingly slow." TERMNL);
841         printf("    Not in simulation? Then pn_simulation() failed." TERMNL);
842         printf(TERMNL);
843         return TEST_SKIPPED;
844     }
845 }
846
847 /*
848 * Test pn_cache_invalidate() and pn_cache_invalidate_all().
849 * pn_cache_enable() and pn_cache_disable() are implicitly tested.
850 */
851
852 printf("    Test pn_cache_invalidate()." TERMNL);
853

```

```

854     if ((ret = invalidate(&pn_cache_invalidate)) != TEST_SUCCESS)
855     {
856         printf("      Failure of function pn_cache_invalidate()." TERMNL);
857         printf(TERMNL);
858         return ret;
859     }
860
861     printf("    pn_cache_invalidate_all() is not testible due to invalidation of "
862           "stack." TERMNL
TERMNL);
863
864     /*
865     * Test pn_cache_writeback() and pn_cache_writeback_all().
866     * pn_cache_enable() and pn_cache_disable() are implicitly tested.
867     */
868
869     printf("    Test pn_cache_writeback()." TERMNL);
870
871     if ((ret = writeback(&pn_cache_writeback)) != TEST_SUCCESS)
872     {
873         printf("      Failure of function pn_cache_writeback()." TERMNL);
874         printf(TERMNL);
875         return ret;
876     }
877
878     printf("    Test pn_cache_writeback_all()." TERMNL);
879
880     if ((ret = writeback(&pn_cache_writeback_all)) != TEST_SUCCESS)
881     {
882         printf("      Failure of function pn_cache_writeback_all()." TERMNL);
883         printf(TERMNL);
884         return ret;
885     }
886
887     printf(TERMNL);
888
889     /*
890     * Test pn_cache_flush() and pn_cache_flush_all().
891     * pn_cache_enable() and pn_cache_disable() are implicitly tested.
892     */
893
894     printf("    Test pn_cache_flush()." TERMNL);
895
896     if ((ret = flush(&pn_cache_flush)) != TEST_SUCCESS)
897     {
898         printf("      Failure of function pn_cache_flush()." TERMNL);
899         printf(TERMNL);
900         return ret;
901     }
902
903     printf("    Test pn_cache_flush_all()." TERMNL);
904
905     if ((ret = flush(&pn_cache_flush_all)) != TEST_SUCCESS)
906     {
907         printf("      Failure of function pn_cache_flush_all()." TERMNL);
908         printf(TERMNL);
909         return ret;
910     }
911
912     printf(TERMNL);
913
914     /* enable cache since it was disabled in last test */
915     pn_cache_enable();
916
917     return TEST_SUCCESS;
918 }
919
920 #endif /* PN_WITH_CACHE */
921
922 /*-----*/
923
924 #ifdef PN_WITH_EXCEPTION
925
926 TEST_RET test_exception(void)
927 {
928     /*
929     * locals
930     */
931     int ret;                /* return value */
932     int i;                  /* loop variable */
933
934     printf("    Test pn_exception_set_handler() and pn_ecall()." TERMNL);
935     printf("      Hang in an exception handler for all environment calls."
TERMNL);
936
937
938     /* hang in the handler TODO */
939     for (i = 8; i <= 11; i++)

```

```

940 {
941     if ((ret = pn_exception_set_handler(&handler, i)) != PN_SUCCESS)
942     {
943         printf("      Error in pn_exception_set_handler()." TERMNL);
944         return TEST_FAIL;
945     }
946 }
947
948 /* set the variable to unchanged */
949 s_exc_var = 0;
950
951 /*
952 * Cause an environment call exception.
953 * In the handler, the static variable s_exc_var should be changed.
954 */
955
956 pn_ecall();
957
958 /* check if the variable was changed */
959 if (s_exc_var == 0)
960 {
961     printf("      The test variable was not changed." TERMNL);
962     return TEST_FAIL;
963 }
964
965 printf("      The test variable was changed. Good." TERMNL);
966
967 return TEST_SUCCESS;
968 }
969
970 #endif /* PN_WITH_EXCEPTION */
971
972 /*-----*/
973
974 #if defined PN_WITH_SPINLOCK && defined PN_WITH_THREAD
975
976 TEST_RET test_spinlock(void)
977 {
978     /*
979     * locals
980     */
981     PN_CID          coreid;
982     static _pn_spinlock lock;
983     int             i;
984     int             count_CPU[NUMCORE_MIN];
985     static int      *testarrayp = s_testarray;
986                                     /* pointer to position in test array */
987
988     /* check if the test case is actually doable on current architecture */
989     CPU_MSK_CHECK;
990
991     printf(TERMNL);
992
993     /* initialize the lock */
994     if (pn_spinlock_init(&lock) != PN_SUCCESS)
995     {
996         printf("      Failure of function pn_spinlock_init()." TERMNL);
997         return TEST_FAIL;
998     }
999
1000     printf("      Lock was initialized." TERMNL);
1001
1002     /* since no one else is in the game yet, we should be able to lock it */
1003     if (pn_spinlock_trylock(&lock) != PN_SUCCESS)
1004     {
1005         printf("      Failure of function pn_spinlock_trylock()." TERMNL);
1006         return TEST_FAIL;
1007     }
1008
1009     printf("      Locked successfully." TERMNL);
1010
1011     /* locking twice should fail */
1012     if (pn_spinlock_trylock(&lock) == PN_SUCCESS)
1013     {
1014         printf("      Failure of function pn_spinlock_trylock()." TERMNL);
1015         return TEST_FAIL;
1016     }
1017
1018     printf("      Locking twice failed as expected." TERMNL);
1019
1020     /* unlocking should work */
1021     if (pn_spinlock_unlock(&lock) != PN_SUCCESS)
1022     {
1023         printf("      Failure of function pn_spinlock_unlock()." TERMNL);
1024         return TEST_FAIL;
1025     }
1026

```

```

1027     printf("    Unlocked successfully." TERMNL);
1028
1029     /* unlocking twice should fail */
1030     if (pn_spinlock_unlock(&lock) == PN_SUCCESS)
1031     {
1032         printf("    Failure of function pn_spinlock_unlock()." TERMNL);
1033         return TEST_FAIL;
1034     }
1035
1036     printf("    Unlocking twice failed as expected." TERMNL);
1037
1038     /* since the lock is unlocked now, locking it should work */
1039     if (pn_spinlock_lock(&lock) != PN_SUCCESS)
1040     {
1041         printf("    Failure of function pn_spinlock_lock()." TERMNL);
1042         return TEST_FAIL;
1043     }
1044
1045     printf("    Locked successfully." TERMNL);
1046
1047     /* destroying the lock should work */
1048     if (pn_spinlock_destroy(&lock) != PN_SUCCESS)
1049     {
1050         printf("    Failure of function pn_spinlock_destroy()." TERMNL);
1051         return TEST_FAIL;
1052     }
1053
1054     printf("    Destroyed lock successfully." TERMNL);
1055
1056     /* re-initializing the lock should work */
1057     if (pn_spinlock_init(&lock) != PN_SUCCESS)
1058     {
1059         printf("    Failure of function pn_spinlock_init()." TERMNL);
1060         return TEST_FAIL;
1061     }
1062
1063     printf("    Lock was initialized." TERMNL TERMNL);
1064     printf("    Opening up 2 threads now. Immediately fill an array with the IDs"
1065            " of the cores plus 1." TERMNL);
1066
1067     /* open up two threads */
1068     if ((coreid = pn_begin_threaded(2)) < PN_SUCCESS)
1069     {
1070         printf("    Failure of function pn_begin_threaded()." TERMNL);
1071         return TEST_FAIL;
1072     }
1073
1074     /* fill the test array */
1075     for (i = 0; i < (ARRAYLENGTH / 2); i++)
1076     {
1077         /* get the lock */
1078         pn_spinlock_lock(&lock);
1079
1080         /* put something into array */
1081         *testarrayp = pn_coreid() + 1;
1082
1083         /* set testarrayp */
1084         testarrayp++;
1085
1086         /* unlock the lock */
1087         pn_spinlock_unlock(&lock);
1088     }
1089
1090     if (coreid == 0)
1091         printf("    Test array was filled in, end threaded mode and destroy lock."
1092                TERMNL);
1093
1094     /* end threaded mode */
1095     if (pn_end_threaded() != PN_SUCCESS)
1096     {
1097         printf("    Failure of function pn_end_threaded()." TERMNL);
1098         return TEST_FAIL;
1099     }
1100
1101     printf("    Ended threaded mode successfully." TERMNL);
1102
1103     /* destroying the lock should work */
1104     if (pn_spinlock_destroy(&lock) != PN_SUCCESS)
1105     {
1106         if (coreid == 0)
1107             printf("    Failure of function pn_spinlock_destroy()." TERMNL);
1108         return TEST_FAIL;
1109     }
1110
1111     printf("    Destroyed lock successfully." TERMNL);
1112

```

```

1113  /* check the testarray */
1114  memset(count_CPU, 0, (sizeof (int)) * NUMCORE_MIN);
1115  for (i = 0; i < ARRAYLENGTH; i++)
1116  {
1117      if ((s_testarray[i] > NUMCORE_MIN) || (s_testarray[i] < 1))
1118      {
1119          printf("  The test array contained wrong values:" TERMNL);
1120          print_testarray();
1121          return TEST_FAIL;
1122      }
1123      else
1124      {
1125          count_CPU[s_testarray[i] - 1]++;
1126      }
1127  }
1128
1129  /* check the counters */
1130  for (i = 0; i < NUMCORE_MIN; i++)
1131  {
1132      if (count_CPU[i] != (ARRAYLENGTH/NUMCORE_MIN))
1133      {
1134          printf("  The test array core distribution is wrong." TERMNL);
1135          printf("  Counted %d entries by core with ID %d. Array:" TERMNL,
1136                count_CPU[i], i);
1137          print_testarray();
1138          return TEST_FAIL;
1139      }
1140  }
1141
1142  /* test was successful, print array */
1143  printf("  Test array was filled in correctly! Array:" TERMNL);
1144  print_testarray();
1145
1146  printf(TERMNL);
1147
1148  return TEST_SUCCESS;
1149 }
1150
1151 #endif /* defined PN_WITH_SPINLOCK && defined PN_WITH_THREAD */
1152
1153 /*EOF*****
1154

```

## 6.3.2 Macro Definition Documentation

### 6.3.2.1 CPU\_MSK

```
#define CPU_MSK 0b11
```

Bitmask of cores that shall be linked/threaded together.

Number of cores shall be equal to NUMCORE\_MIN.

### 6.3.2.2 CPU\_MSK\_CHECK

```
#define CPU_MSK_CHECK
```

**Value:**

```

if ((pn_m2cap() & CPU_MSK) != CPU_MSK)
{
    printf("  This Testcase demands core 0 and 1 to "
          "be capable of Mode 2." TERMNL);
    return TEST_SKIPPED;
}

```

Checks if at least two Mode 2 capable cores are available.

### 6.3.2.3 LOOPS

```
#define LOOPS 4
```

Number of loops for testing linked/threaded Mode.

Must be dividable by [NUMCORE\\_MIN](#).

### 6.3.2.4 NUMCORE\_MIN

```
#define NUMCORE_MIN 2
```

Minimal number of cores that shall be linked/threaded together.

Also check [CPU\\_MSK](#) when touching this value.

### 6.3.2.5 NUMCORES\_CHECK

```
#define NUMCORES_CHECK
```

**Value:**

```
if (pn_numcores() < NUMCORE_MIN) \
    { \
        printf("  This Testcase demands at least 2 cores." \
              TERMNL); \
        return TEST_SKIPPED; \
    }
```

Checks if minimum number of cores is available.

**Todo** If there's enough cores for `pn_numcores()` to be negative some day, this needs to be changed.

### 6.3.2.6 PLAUSIBLE\_TIME

```
#define PLAUSIBLE_TIME 30000
```

Number of ns that are considered plausible between two timer gets.

This depends on your ParaNut configuration. If your frequency is lower than 25MHz and you saw that the timer values actually made sense, you can crank this up. On a faster ParaNut, this value should not be cranked up.

**Todo** If the ParaNut is getting faster in the future, this might need to change.

## 6.3.3 Function Documentation

### 6.3.3.1 test\_cache()

```
TEST_RET test_cache (  
    void )
```

Tests all functions in exception module.

Assumes exception module to have been initialized before.

**Todo** I have no idea how I am supposed to test `pn_interrupt_enable()` and `pn_interrupt_disable()` at the current ParaNut implementation, since we do not have a working `mtimecmp` and `mtime` register yet. This may change in the future, though.

### 6.3.3.2 test\_cap()

```
TEST_RET test_cap (  
    void )
```

**Todo** Test `pn_m2cap_g()` when it is available.

**Todo** Test `pn_m3cap_g()` when it is available.

### 6.3.3.3 test\_exception()

```
TEST_RET test_exception (  
    void )
```

Tests all functions in spinlock module.

Implicitly tests `pn_begin_threaded()` and `pn_end_threaded()`.

**Todo** This needs changes in case there's more than one group of CPUs.

### 6.3.3.4 test\_halt\_CoPU()

```
TEST_RET test_halt_CoPU (  
    void )
```

**Todo** Test group function when it is available.

### 6.3.3.5 test\_link()

```
TEST_RET test_link (  
    void )
```

**Todo** Group function test (as soon as implemented in libparanut).

### 6.3.3.6 test\_numcores()

```
TEST_RET test_numcores (  
    void )
```

Tests functions pn\_m2cap() and pn\_m3cap().

### 6.3.3.7 test\_thread()

```
TEST_RET test_thread (  
    void )
```

**Todo** Test group functions when they are available.

**Todo** Group function test (as soon as implemented in libparanut).

**Todo** POSIX Threads

### 6.3.3.8 test\_time()

```
TEST_RET test_time (  
    void )
```

Tests function pn\_numcores().



# Index

Architecture Defines, [9](#)

    M2CAP\_MSK, [9](#)

    M3CAP\_MSK, [9](#)

CPU\_MSK\_CHECK

    libparanut\_unittest\_testcases.c, [32](#)

CPU\_MSK

    libparanut\_unittest\_testcases.c, [32](#)

LOOPS

    libparanut\_unittest\_testcases.c, [32](#)

libparanut\_unittest.h, [11](#)

    test\_cache, [13](#)

    test\_cap, [14](#)

    test\_exception, [14](#)

    test\_halt\_CoPU, [14](#)

    test\_link, [14](#)

    test\_numcores, [15](#)

    test\_thread, [15](#)

    test\_time, [15](#)

libparanut\_unittest\_main.c, [16](#)

    TEST, [18](#)

libparanut\_unittest\_testcases.c, [19](#)

    CPU\_MSK\_CHECK, [32](#)

    CPU\_MSK, [32](#)

    LOOPS, [32](#)

    NUMCORE\_MIN, [33](#)

    NUMCORES\_CHECK, [33](#)

    PLAUSIBLE\_TIME, [33](#)

    test\_cache, [33](#)

    test\_cap, [34](#)

    test\_exception, [34](#)

    test\_halt\_CoPU, [34](#)

    test\_link, [34](#)

    test\_numcores, [35](#)

    test\_thread, [35](#)

    test\_time, [35](#)

M2CAP\_MSK

    Architecture Defines, [9](#)

M3CAP\_MSK

    Architecture Defines, [9](#)

NUMCORE\_MIN

    libparanut\_unittest\_testcases.c, [33](#)

NUMCORES\_CHECK

    libparanut\_unittest\_testcases.c, [33](#)

PLAUSIBLE\_TIME

    libparanut\_unittest\_testcases.c, [33](#)

TEST

    libparanut\_unittest\_main.c, [18](#)

Test Case Return Values, [10](#)

test\_cache

    libparanut\_unittest.h, [13](#)

    libparanut\_unittest\_testcases.c, [33](#)

test\_cap

    libparanut\_unittest.h, [14](#)

    libparanut\_unittest\_testcases.c, [34](#)

test\_exception

    libparanut\_unittest.h, [14](#)

    libparanut\_unittest\_testcases.c, [34](#)

test\_halt\_CoPU

    libparanut\_unittest.h, [14](#)

    libparanut\_unittest\_testcases.c, [34](#)

test\_link

    libparanut\_unittest.h, [14](#)

    libparanut\_unittest\_testcases.c, [34](#)

test\_numcores

    libparanut\_unittest.h, [15](#)

    libparanut\_unittest\_testcases.c, [35](#)

test\_thread

    libparanut\_unittest.h, [15](#)

    libparanut\_unittest\_testcases.c, [35](#)

test\_time

    libparanut\_unittest.h, [15](#)

    libparanut\_unittest\_testcases.c, [35](#)