libparanut Unittest

Generated by Doxygen 1.8.13

Contents

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

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 explainations 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 Para↩ Nut 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:

Chapter 4

File Index

4.1 File List

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

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 explicitely 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 succeded.
- #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

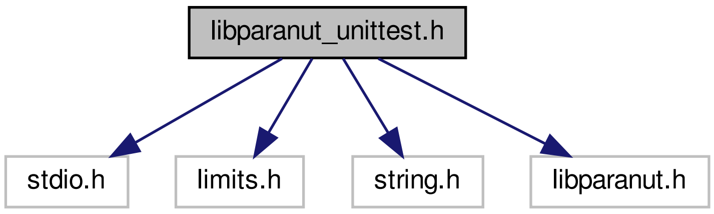
File Documentation

6.1 libparanut_unittest.h File Reference

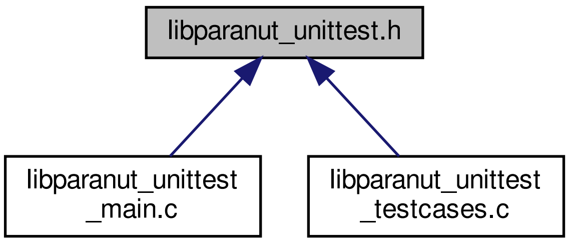
Contains helpers and function prototypes of testcases.

```
#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 succeded.
- #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.

```

/*Includes*********************************************************************/

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

/*Architecture Defines*********************************************************/

/*
 * The weird #if DOXYGEN is done because Doxygen won’t document it otherwise.
 * Sorry about that.
 * If you find a more elegant solution, do not hesitate to put it in :)
 */

```

```
#if DOXYGEN

   #define NUMCORES

#endif

#ifndef NUMCORES

   #define NUMCORES
   #error NUMCORES undefined! Check "Architecture Defines" Documentation!

#endif

#if DOXYGEN

   #define M2CAP_MSK

#endif

#ifndef M2CAP_MSK

   #define M2CAP_MSK
   #error M2CAP_MSK undefined! Check "Architecture Defines" Documentation!

#endif

#if DOXYGEN

   #define M3CAP_MSK

#endif

#ifndef M3CAP_MSK

   #define M3CAP_MSK
   #error M3CAP_MSK undefined! Check "Architecture Defines" Documentation!

#endif

/*Helpers**********************************************************************/

#define TERMNL              "\n\r"

typedef int8_t TEST_RET;

#define TEST_SUCCESS        ( 0)

#define TEST_FAIL           (-1)

#define TEST_SKIPPED        (-2)

/*Test Case Prototypes*********************************************************/

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);

/*EOF**************************************************************************/

```

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.

Implicitely 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 implicitely 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  )
```

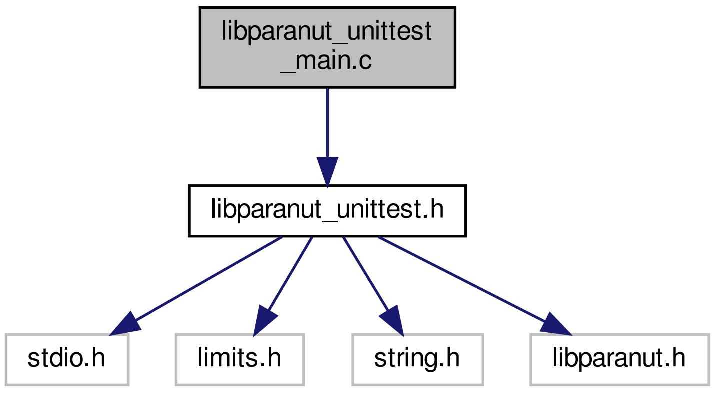
Tests function pn_numcores().

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.

```
/*
 * 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.
 */

/*Includes*****************************************************************/

#include "libparanut_unittest.h"

/*Helpers******************************************************************/

#define TEST(x)      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);     \
                     }

/*Main Function************************************************************/

int main()
{

#if !(defined PN_WITH_BASE)            \
      && !(defined PN_WITH_CACHE)      \
      && !(defined PN_WITH_LINK)       \
      && !(defined PN_WITH_THREAD)     \
      && !(defined PN_WITH_EXCEPTION)  \
      && !(defined PN_WITH_SPINLOCK)

   printf("###No Modules were compiled in libparanut, cannot start test :(###"
                                                                        TERMNL);
   return TEST_FAIL;

#else

  /*
   * locals
   */
   TEST_RET       ret;              /* saves return value - see helper TEST() */

#endif

#ifdef PN_WITH_BASE
   long long int  start, end;       /* start and end time of test              */
#endif /* PN_WITH_BASE */

   printf("###Welcome to libparanut Unittest###" TERMNL TERMNL);

  /*
   * Initialize all of libparanut Modules that need initializing. This is sorta
   * untestable by itself. Things will go wrong in the unit test itself if
   * something’s wrong here, though.
   */

#ifdef PN_WITH_EXCEPTION
   printf("###Initializing exception module ...###" TERMNL TERMNL);
   pn_exception_init();
#endif /* PN_WITH_EXCEPTION */

#ifdef PN_WITH_CACHE
   printf("###Initializing cache module ...###" TERMNL TERMNL);
   if (pn_cache_init() != PN_SUCCESS)
   {
      printf("Error in pn_cache_init(). We can not proceed with this test."
                                                                 TERMNL
  TERMNL);
      printf("###Unsuccessful End of Test :(###" TERMNL TERMNL);
      return -1;
   }
#endif /* PN_WITH_CACHE */

#ifdef PN_WITH_BASE

   TEST(test_time)

   start = pn_time_ns();
   TEST(test_numcores)


   TEST(test_cap)

#endif /* PN_WITH_BASE */

#if defined PN_WITH_LINK && defined PN_WITH_BASE

   TEST(test_link)

#endif /* defined PN_WITH_LINK && defined PN_WITH_BASE */

#if defined PN_WITH_THREAD && defined PN_WITH_BASE

   TEST(test_thread)

#endif /* defined PN_WITH_THREAD && defined PN_WITH_BASE */

#if defined PN_WITH_BASE && defined PN_WITH_THREAD

   TEST(test_halt_CoPU)

#endif /* defined PN_WITH_BASE && defined PN_WITH_THREAD */

#ifdef PN_WITH_CACHE

   TEST(test_cache)

#endif /* PN_WITH_CACHE */

#ifdef PN_WITH_EXCEPTION

   TEST(test_exception)

#endif /* PN_WITH_EXCEPTION */

#if defined PN_WITH_SPINLOCK && defined PN_WITH_THREAD

   TEST(test_spinlock)

#endif /* defined PN_WITH_SPINLOCK && defined PN_WITH_THREAD */

#ifdef PN_WITH_BASE
   end = pn_time_ns();
   printf("###Execution time of test: %lld ms###" TERMNL TERMNL,
                                                    (end - start) / 1000000);
#endif /* PN_WITH_BASE */

   printf("###Successfull End of Test :)###" TERMNL TERMNL);

   return TEST_SUCCESS;
}

/*EOF**************************************************************************/
```

6.2.2 Macro Definition Documentation

6.2.2.1 TEST

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

Value:

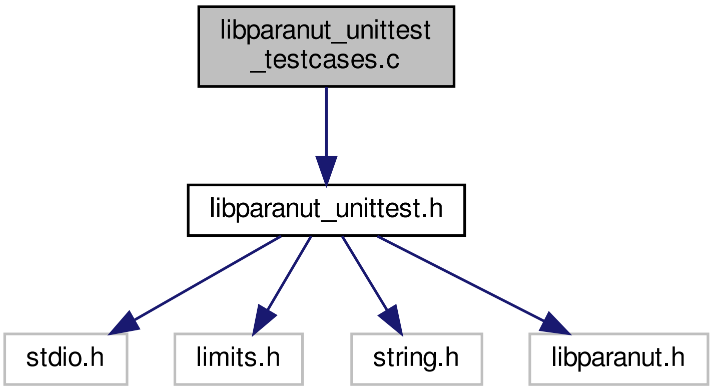
```
printf("###STARTING %s###" TERMNL, #x);                    \
                     if ((ret = x()) == TEST_FAIL)                          \
                     {                                                      \
                        printf("###TESTCASE FAILED###" TERMNL TERMNL);      \
                        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 0b11

  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.

```
/*
 * 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.
 */

/*Includes*********************************************************************/

#include "libparanut_unittest.h"

/*Local Defines****************************************************************/

/* TODO Documentation */

#define NUMCORE_MIN     2

#define CPU_MSK         0b11

#define LOOPS           4

#define PLAUSIBLE_TIME 30000

#define ARRAYLENGTH     100

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

#define CPU_MSK_CHECK   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;                                \
                        }

/*
```

```
 * Weak definitions of functions called in linked_threaded_test().
 */
#if !(defined DOXYGEN)
#if !(defined PN_WITH_LINK)
PN_CID pn_begin_linked(PN_NUMC numcores)      { return 0; }
PN_CID pn_begin_linked_m(PN_CMSK coremask)    { return 0; }
int    pn_end_linked(void)                    { return 0; }
#endif /* !(defined PN_WITH_LINK) */
#if !(defined PN_WITH_THREAD)
PN_CID pn_begin_threaded(PN_NUMC numcores)    { return 0; }
PN_CID pn_begin_threaded_m(PN_CMSK coremask) { return 0; }
int    pn_end_threaded(void)                  { return 0; }
#endif /* !(defined PN_WITH_THREAD) */
#endif /* !(defined DOXYGEN) */

/*Variables*****************************************************************/

#if defined PN_WITH_SPINLOCK && defined PN_WITH_THREAD

static int s_testarray[ARRAYLENGTH];
#endif /* defined PN_WITH_SPINLOCK && defined PN_WITH_THREAD */

#ifdef PN_WITH_EXCEPTION

static int s_exc_var;
#endif /* PN_WITH_EXCEPTION */

/*Static Functions**********************************************************/

#if ((defined PN_WITH_LINK) || (defined PN_WITH_THREAD)) && defined PN_WITH_BASE

static void set_arrays(int *sum, int *a, int *b)
{
   /*
    * locals
    */
    int i;                              /* loop counting variable              */

    for (i = 0; i < LOOPS; i++)
    {
       sum[i] = 0;

       /* also change check_sum() when changing this */
       a[i] = 1;
       b[i] = 1;
    }

    return;
}

/*-------------------------------------------------------------------------*/

static void calc_sum(PN_CID cid, int *sum, int *a, int *b)
{
   /*
    * locals
    */
    int i;                              /* loop counting variable              */

    for (i = cid; i < LOOPS; i += NUMCORE_MIN)
       sum[i] = a[i] + b[i];

    return;
}

/*-------------------------------------------------------------------------*/

static TEST_RET check_sum(int *sum)
{
   /*
    * locals
    */
    int i;                              /* loop counting variable              */

    for (i = 0; i < LOOPS; i++)

       /* sum should be two since a and b are filled with 1s */
       if (sum[i] != 2)
          return TEST_FAIL;

    return TEST_SUCCESS;
}

/*-------------------------------------------------------------------------*/

static TEST_RET linked_threaded_test(char *funcname, PN_CID (*funcp)())
{
   /*
    * locals
    */
    static int     sum[LOOPS], a[LOOPS], b[LOOPS];
                                        /* sum is sum of a and b               */
    PN_CID         cid;                 /* core ID                             */
    int            err;                 /* error                               */
    int            i;                   /* loop counter                        */

    printf("   Test %s." TERMNL, funcname);

    /* fill in the arrays */
    set_arrays(sum, a, b);

    /* print some debug information */
    printf(TERMNL);
    printf("      Sum array before calculation:" TERMNL);
    for (i = 0; i < LOOPS; i++)
    {
       printf("         sum[%d] = %d" TERMNL, i, sum[i]);
    }
    printf(TERMNL);

    /* begin linked or threaded mode */
    if ((funcp == &pn_begin_linked) || (funcp == &pn_begin_threaded))
    {
       cid = funcp(NUMCORE_MIN);
    }
    else if ((funcp == &pn_begin_linked_m) || (funcp == &pn_begin_threaded_m))
    {
       cid = funcp(CPU_MSK);
    }
    else
    {
       printf("   You passed a not yet implemented function to subtest "
                                              "linked_threaded_test()" TERMNL);
       return TEST_FAIL;
    }

    /* conditional jump doesn't matter if we didn't even go into linked mode */
    if (cid < 0)
    {
       printf("   Failure of function %s." TERMNL, funcname);
       return TEST_FAIL;
    }

    /* set sum to sum of a and b */
    calc_sum(cid, sum, a, b);


    /* end linked or threaded mode */
    if ((funcp == &pn_begin_linked) || (funcp == &pn_begin_linked_m))
    {
       err = pn_end_linked();
       if (err)
       {
          printf("   Failure of function pn_end_linked()." TERMNL);
          return TEST_FAIL;
       }
    }
    else
    {
       err = pn_end_threaded();
       if (err)
       {
          printf("   Failure of function pn_end_threaded()." TERMNL);
          return TEST_FAIL;
       }
    }

    /*  print some debug information */
    if (cid == 0)
    {
       printf("      Sum array after calculation:" TERMNL);
       for (i = 0; i < LOOPS; i++)
       {
          printf("         sum[%d] = %d" TERMNL, i, sum[i]);
       }
       printf(TERMNL);
    }

    /* check the sum array */
    if (check_sum(sum) == TEST_FAIL)
    {
       printf("   Failure of calculation in chosen mode." TERMNL);
       return TEST_FAIL;
    }
```

```

   return TEST_SUCCESS;
}

#endif /* ((defined PN_WITH_LINK) || (defined PN_WITH_THREAD)) */
       /*    && defined PN_WITH_BASE                           */

/*---------------------------------------------------------------------------*/

#if defined PN_WITH_SPINLOCK && defined PN_WITH_THREAD

static void print_testarray(void)
{
  /*
   * locals
   */
   int i;

   for (i = 0; i < (ARRAYLENGTH / 10); i++)
   {
      printf("      %i  %i  %i  %i  %i  %i  %i  %i  %i  %i" TERMNL,
                                                     s_testarray[(i * 10) + 0],
                                                     s_testarray[(i * 10) + 1],
                                                     s_testarray[(i * 10) + 2],
                                                     s_testarray[(i * 10) + 3],
                                                     s_testarray[(i * 10) + 4],
                                                     s_testarray[(i * 10) + 5],
                                                     s_testarray[(i * 10) + 6],
                                                     s_testarray[(i * 10) + 7],
                                                     s_testarray[(i * 10) + 8],
                                                     s_testarray[(i * 10) + 9]);
   }

   return;
}

#endif /* defined PN_WITH_SPINLOCK && defined PN_WITH_THREAD */

/*---------------------------------------------------------------------------*/

#ifdef PN_WITH_EXCEPTION

static void handler(unsigned int cause,
                    unsigned int program_counter,
                    unsigned int mtval)
{
   printf("      Hello, this is the exception handler!" TERMNL);

   if ((cause < 8) || (cause > 11))
   {
      printf("         Cause was not correctly passed to the handler." TERMNL);
      return;
   }

   printf("         Changing variable now." TERMNL);
   s_exc_var = 1;
   printf("         Setting exception program counter to next instruction."
                                                                   TERMNL);
   pn_progress_mepc();
   printf("         Returning ..." TERMNL);

   return;
}

#endif /* PN_WITH_EXCEPTION */

/*---------------------------------------------------------------------------*/

#ifdef PN_WITH_CACHE

static volatile TEST_RET invalidate(int (*invalidate_function)())
{
  /*
   * locals
   */
   static int testvar = 3;

   /* disable cache */
   printf("      Disable Cache." TERMNL);
   pn_cache_disable();

   /* value 1 stands in memory now */
   printf("      Give test variable a value of 1." TERMNL);
   testvar = 1;
   printf("      Value of variable is now %i." TERMNL, testvar);

   /* enable cache again */
   printf("      Enable Cache." TERMNL);
   pn_cache_enable();

   /* value 2 stands in cache now */
   printf("      Give test variable a value of 2." TERMNL);
   testvar = 2;
   printf("      Value of variable is now %i." TERMNL, testvar);

   /* cache invalidate */
   printf("      Invalidate cache." TERMNL);
   if (*invalidate_function == pn_cache_invalidate)
      invalidate_function(&testvar, 0);
   else if (*invalidate_function == pn_cache_invalidate_all)
      invalidate_function();
   else
   {
      printf("      The sub test invalidate() was given a wrong function."
                                                                  TERMNL);
   }

   /* read variable -> should be old value */
   printf("      Variable now has a value of %d, should have value 1." TERMNL,
                                                                 testvar);
   if (testvar != 1)
      return TEST_FAIL;

   return TEST_SUCCESS;
}

/*----------------------------------------------------------------------------*/

static TEST_RET writeback(int (*writeback_function)())
{
  /*
   * locals
   */
   static int testvar = 0;

   /* disable cache */
   printf("      Disable Cache." TERMNL);
   pn_cache_disable();

   /* value 1 stands in memory now */
   printf("      Give test variable a value of 1." TERMNL);
   testvar = 1;

   /* enable cache again */
   printf("      Enable Cache." TERMNL);
   pn_cache_enable();

   /* value 2 stands in cache now */
   printf("      Give test variable a value of 2." TERMNL);
   testvar = 2;

   /* cache writeback */
   printf("      Write back cache." TERMNL);
   if (*writeback_function == pn_cache_writeback)
      writeback_function(&testvar, 0);
   else if (*writeback_function == pn_cache_writeback_all)
      writeback_function();
   else
   {
      printf("      The sub test writeback() was given a wrong function."
                                                                  TERMNL);
   }

   /* disable cache */
   printf("      Disable Cache." TERMNL);
   pn_cache_disable();

   /* read variable -> should be new value */
   printf("      Variable now has a value of %d, should have value 2." TERMNL,
                                                                 testvar);
   if (testvar != 2)
      return TEST_FAIL;

   return TEST_SUCCESS;
}

/*----------------------------------------------------------------------------*/

static TEST_RET flush(int (*flush_function)())
{
  /*
   * locals
   */
   static int testvar = 0;

   /* disable cache */
   printf("      Disable Cache." TERMNL);
   pn_cache_disable();

   /* value 1 stands in memory now */
   printf("      Give test variable a value of 1." TERMNL);
   testvar = 1;

   /* enable cache again */
   printf("      Enable Cache." TERMNL);
   pn_cache_enable();

   /* value 2 stands in cache now */
   printf("      Give test variable a value of 2." TERMNL);
   testvar = 2;

   /* cache flush */
   printf("      Flush cache." TERMNL);
   if (*flush_function == pn_cache_flush)
      flush_function(&testvar, 0);
   else if (*flush_function == pn_cache_flush_all)
      flush_function();
   else
   {
      printf("      The sub test flush() was given a wrong function."
                                                                  TERMNL);
   }

   /* disable cache */
   printf("      Disable Cache." TERMNL);
   pn_cache_disable();

   /* read variable -> should be new value */
   printf("      Variable now has a value of %d, should have value 2." TERMNL,
                                                                 testvar);
   if (testvar != 2)
      return TEST_FAIL;

   return TEST_SUCCESS;
}

#endif /* PN_WITH_CACHE */

/*Test Cases*****************************************************************/

#ifdef PN_WITH_BASE

TEST_RET test_time(void)
{
  /*
   * locals
   */
   long long int start, end;           /* start and end time                  */

  /*
   * Read one time before actual measurement because first time takes the
   * longest and is therefore not represantative.
   */

   start = pn_time_ns();

  /*
   * Actual measurement starts here.
   */

   start = pn_time_ns();
   end   = pn_time_ns();

   printf("   Start time: %lli" TERMNL, start);
   printf("   End time:   %lli" TERMNL, end);

   if (end == start)
      goto _implausible;

   if ((start > end) && ((LLONG_MAX - (start - end)) > PLAUSIBLE_TIME))
      goto _implausible;

   if ((start < end) && ((end - start) > PLAUSIBLE_TIME))
      goto _implausible;

   return TEST_SUCCESS;

_implausible:
   printf("   Implausible." TERMNL);
   return TEST_FAIL;
}
```

```

/*---------------------------------------------------------------------------*/

TEST_RET test_numcores(void)
{
   /*
    * locals
    */
   PN_NUMC numc;                       /* number of cores                     */

   printf("   Test pn_numcores()." TERMNL);
   numc = pn_numcores();
   if (pn_numcores() != NUMCORES)
   {
      printf("   NUMCORES was %i, but pn_numcores() returned %i." TERMNL,
                                                      NUMCORES, (int) numc);
      return TEST_FAIL;
   }
   return TEST_SUCCESS;
}

/*---------------------------------------------------------------------------*/

TEST_RET test_cap(void)
{
   /*
    * locals
    */
   PN_CMSK cmsk;                       /* core mask                           */

   printf("   Test pn_m2cap()." TERMNL);
   cmsk = pn_m2cap();
   if (cmsk != M2CAP_MSK)
   {
      printf("   M2CAP_MSK was %u, but pn_m2cap() returned %u." TERMNL,
                                            M2CAP_MSK, (unsigned int) cmsk);
      return TEST_FAIL;
   }

   printf("   Test pn_m3cap()." TERMNL);
   cmsk = pn_m3cap();
   if (cmsk != M3CAP_MSK)
   {
      printf("   M3CAP_MSK was %u, but pn_m3cap() returned %u." TERMNL,
                                            M3CAP_MSK, (unsigned int) cmsk);
      return TEST_FAIL;
   }

   return TEST_SUCCESS;
}

#endif /* PN_WITH_BASE */

/*---------------------------------------------------------------------------*/

#if defined PN_WITH_LINK && defined PN_WITH_BASE

TEST_RET test_link(void)
{
   /*
    * locals
    */
   int result;                         /* result of subtest                   */

   /* check if the test case is actually doable on current architecture */
   NUMCORES_CHECK;

   /*
    * linked mode, method 1
    */

   printf(TERMNL);
   result = linked_threaded_test("pn_begin_linked()", &pn_begin_linked);

   if (result == TEST_FAIL)
      return TEST_FAIL;

   /*
    * linked mode, method 2
    */

   result = linked_threaded_test("pn_begin_linked_m()", &pn_begin_linked_m);

   if (result == TEST_FAIL)
      return TEST_FAIL;

   /*
   * linked mode, method 3
   */

   return TEST_SUCCESS;
}

#endif /* defined PN_WITH_LINK && defined PN_WITH_BASE */

/*----------------------------------------------------------------------------*/

#if defined PN_WITH_THREAD && defined PN_WITH_BASE

TEST_RET test_thread(void)
{
  /*
   * locals
   */
   int result;                            /* result of subtest                   */

   /* check if the test case is actually doable on current architecture */
   CPU_MSK_CHECK;

  /*
   * threaded mode, method 1
   */

   printf(TERMNL);
   result = linked_threaded_test("pn_begin_threaded()", &pn_begin_threaded);

   if (result == TEST_FAIL)
      return TEST_FAIL;

  /*
   * threaded mode, method 2
   */

   result = linked_threaded_test("pn_begin_threaded_m()", &pn_begin_threaded_m);

   if (result == TEST_FAIL)
      return TEST_FAIL;

  /*
   * threaded mode, method 3
   */

   return TEST_SUCCESS;
}

#endif /* defined PN_WITH_THREAD && defined PN_WITH_BASE */

/*----------------------------------------------------------------------------*/

#if defined PN_WITH_BASE && defined PN_WITH_THREAD

TEST_RET test_halt_CoPU(void)
{
  /*
   * locals
   */
   PN_CID      cid;                                /* core ID                     */
   static int  s_counter = 0;                      /* counter touched by CoPUs    */
   int         counter_copy_1, counter_copy_2;     /* counter copies              */
   int         i;                                  /* loop counter                */
   int         err;                                /* error value                 */

   /* check if the test case is actually doable on current architecture */
   CPU_MSK_CHECK;

  /*
   * pn_halt_CoPU()
   */

   printf("   Test pn_halt_CoPU()." TERMNL);

   cid = pn_begin_threaded(NUMCORE_MIN);

   if (cid == 0)
      printf("      Threaded Mode started successfully." TERMNL);

   if (cid != 0)
      while (1)
         s_counter++;
   printf("      CoPUs are counting a static counter now." TERMNL);

   for (i = 1; i < NUMCORE_MIN; i++)
   {
      if ((err = pn_halt_CoPU((PN_CID)i)) != PN_SUCCESS)
```

```
    {
       printf("   pn_halt_CoPU() returned error %d." TERMNL, err);
       return TEST_FAIL;
    }
  }
  printf("      Tried to halt them. Check if they are still counting." TERMNL);

  /* since all CoPUs should be disabled, the counter should not change */
  counter_copy_1 = s_counter;
  pn_time_ns();
  counter_copy_2 = s_counter;

  if (counter_copy_1 != counter_copy_2)
  {
     printf("   The CoPUs have not been disabled!." TERMNL);
     return TEST_FAIL;
  }
  printf("      They aren't. Good." TERMNL);

 /*
  * pn_halt_CoPU_m()
  */

  printf("   Test pn_halt_CoPU_m()." TERMNL);

  cid = pn_begin_threaded_m(CPU_MSK);
  if (cid == 0)
     printf("      Threaded Mode started successfully." TERMNL);

  if (cid != 0)
     while (1)
        s_counter++;
  printf("      CoPUs are counting a static counter now." TERMNL);

  if ((err = pn_halt_CoPU_m((CPU_MSK & 0xFFFFFFFE))) != PN_SUCCESS)
  {
     printf("      pn_halt_CoPU_m() returned error %d." TERMNL, err);
     return TEST_FAIL;
  }
  printf("      Tried to halt them. Check if they are still counting." TERMNL);

  /* since all CoPUs should be disabled, the counter should not change */
  counter_copy_1 = s_counter;
  pn_time_ns();
  counter_copy_2 = s_counter;

  if (counter_copy_1 != counter_copy_2)
  {
     printf("   The CoPUs have not been disabled!." TERMNL);
     return TEST_FAIL;
  }
  printf("      They aren't. Good." TERMNL);

  return TEST_SUCCESS;
}

#endif /* defined PN_WITH_BASE && defined PN_WITH_THREAD */

/*---------------------------------------------------------------------------*/

#ifdef PN_WITH_CACHE

TEST_RET test_cache(void)
{
 /*
  * locals
  */
  TEST_RET ret;

  /* note about skipping some parts on simulation */
  if (pn_simulation())
  {
     printf(TERMNL);
     printf("   Testing the pn_cache_...() functions is skipped in ParaNut"
            " simulation since it is excruciatingly slow." TERMNL);
     printf("   Not in simulation? Then pn_simulation() failed." TERMNL);
     printf(TERMNL);
     return TEST_SKIPPED;
  }

 /*
  * Test pn_cache_invalidate() and pn_cache_invalidate_all().
  * pn_cache_enable() and pn_cache_disable() are implicitely tested.
  */

  printf("   Test pn_cache_invalidate()." TERMNL);

```

```
   if ((ret = invalidate(&pn_cache_invalidate)) != TEST_SUCCESS)
   {
      printf("      Failure of function pn_cache_invalidate()." TERMNL);
      printf(TERMNL);
      return ret;
   }

   printf("   pn_cache_invalidate_all() is not testible due to invalidation of "
                                                      "stack." TERMNL
  TERMNL);

  /*
   * Test pn_cache_writeback() and pn_cache_writeback_all().
   * pn_cache_enable() and pn_cache_disable() are implicitely tested.
   */

   printf("   Test pn_cache_writeback()." TERMNL);

   if ((ret = writeback(&pn_cache_writeback)) != TEST_SUCCESS)
   {
      printf("      Failure of function pn_cache_writeback()." TERMNL);
      printf(TERMNL);
      return ret;
   }

   printf("   Test pn_cache_writeback_all()." TERMNL);

   if ((ret = writeback(&pn_cache_writeback_all)) != TEST_SUCCESS)
   {
      printf("      Failure of function pn_cache_writeback_all()." TERMNL);
      printf(TERMNL);
      return ret;
   }

   printf(TERMNL);

  /*
   * Test pn_cache_flush() and pn_cache_flush_all().
   * pn_cache_enable() and pn_cache_disable() are implicitely tested.
   */

   printf("   Test pn_cache_flush()." TERMNL);

   if ((ret = flush(&pn_cache_flush)) != TEST_SUCCESS)
   {
      printf("      Failure of function pn_cache_flush()." TERMNL);
      printf(TERMNL);
      return ret;
   }

   printf("   Test pn_cache_flush_all()." TERMNL);

   if ((ret = flush(&pn_cache_flush_all)) != TEST_SUCCESS)
   {
      printf("      Failure of function pn_cache_flush_all()." TERMNL);
      printf(TERMNL);
      return ret;
   }

   printf(TERMNL);

   /* enable cache since it was disabled in last test */
   pn_cache_enable();

   return TEST_SUCCESS;
}

#endif /* PN_WITH_CACHE */

/*-----------------------------------------------------------------------------*/

#ifdef PN_WITH_EXCEPTION

TEST_RET test_exception(void)
{
  /*
   * locals
   */
   int ret;                            /* return value                          */
   int i;                              /* loop variable                         */

   printf("   Test pn_exception_set_handler() and pn_ecall()." TERMNL);
   printf("      Hang in an exception handler for all environment calls."
                                                                        TERMNL);

   /* hang in the handler TODO */
   for (i = 8; i <= 11; i++)
   {
      if ((ret = pn_exception_set_handler(&handler, i)) != PN_SUCCESS)
      {
         printf("      Error in pn_exception_set_handler()." TERMNL);
         return TEST_FAIL;
      }
   }

   /* set the variable to unchanged */
   s_exc_var = 0;

  /*
   * Cause an environment call exception.
   * In the handler, the static variable s_exc_var should be changed.
   */

   pn_ecall();

   /* check if the variable was changed */
   if (s_exc_var == 0)
   {
      printf("      The test variable was not changed." TERMNL);
      return TEST_FAIL;
   }

   printf("      The test variable was changed. Good." TERMNL);

   return TEST_SUCCESS;
}

#endif /* PN_WITH_EXCEPTION */

/*-----------------------------------------------------------------------------*/

#if defined PN_WITH_SPINLOCK && defined PN_WITH_THREAD

TEST_RET test_spinlock(void)
{
  /*
   * locals
   */
   PN_CID                 coreid;
   static _pn_spinlock  lock;
   int                    i;
   int                    count_CPU[NUMCORE_MIN];
   static int             *testarrayp = s_testarray;
                                        /* pointer to position in test array   */

   /* check if the test case is actually doable on current architecture */
   CPU_MSK_CHECK;

   printf(TERMNL);

   /* initialize the lock */
   if (pn_spinlock_init(&lock) != PN_SUCCESS)
   {
      printf("   Failure of function pn_spinlock_init()." TERMNL);
      return TEST_FAIL;
   }

   printf("   Lock was initialized." TERMNL);

   /* since no one else is in the game yet, we should be able to lock it */
   if (pn_spinlock_trylock(&lock) != PN_SUCCESS)
   {
      printf("   Failure of function pn_spinlock_trylock()." TERMNL);
      return TEST_FAIL;
   }

   printf("   Locked successfully." TERMNL);

   /* locking twice should fail */
   if (pn_spinlock_trylock(&lock) == PN_SUCCESS)
   {
      printf("   Failure of function pn_spinlock_trylock()." TERMNL);
      return TEST_FAIL;
   }

   printf("   Locking twice failed as expected." TERMNL);

   /* unlocking should work */
   if (pn_spinlock_unlock(&lock) != PN_SUCCESS)
   {
      printf("   Failure of function pn_spinlock_unlock()." TERMNL);
      return TEST_FAIL;
   }

   printf("   Unlocked successfully." TERMNL);

   /* unlocking twice should fail */
   if (pn_spinlock_unlock(&lock) == PN_SUCCESS)
   {
      printf("   Failure of function pn_spinlock_unlock()." TERMNL);
      return TEST_FAIL;
   }

   printf("   Unlocking twice failed as expected." TERMNL);

   /* since the lock is unlocked now, locking it should work */
   if (pn_spinlock_lock(&lock) != PN_SUCCESS)
   {
      printf("   Failure of function pn_spinlock_lock()." TERMNL);
      return TEST_FAIL;
   }

   printf("   Locked successfully." TERMNL);

   /* destroying the lock should work */
   if (pn_spinlock_destroy(&lock) != PN_SUCCESS)
   {
      printf("   Failure of function pn_spinlock_destroy()." TERMNL);
      return TEST_FAIL;
   }

   printf("   Destroyed lock successfully." TERMNL);

   /* re-initializing the lock should work */
   if (pn_spinlock_init(&lock) != PN_SUCCESS)
   {
      printf("   Failure of function pn_spinlock_init()." TERMNL);
      return TEST_FAIL;
   }

   printf("   Lock was initialized." TERMNL TERMNL);
   printf("   Opening up 2 threads now. Immediately fill an array with the IDs"
                                                " of the cores plus 1." TERMNL);

   /* open up two threads */
   if ((coreid = pn_begin_threaded(2)) < PN_SUCCESS)
   {
      printf("   Failure of function pn_begin_threaded()." TERMNL);
      return TEST_FAIL;
   }

   /* fill the test array */
   for (i = 0; i < (ARRAYLENGTH / 2); i++)
   {
      /* get the lock */
      pn_spinlock_lock(&lock);

      /* put something into array */
      *testarrayp = pn_coreid() + 1;

      /* set testarrayp */
      testarrayp++;

      /* unlock the lock */
      pn_spinlock_unlock(&lock);
   }

   if (coreid == 0)
      printf("   Test array was filled in, end threaded mode and destroy lock."
                                                              TERMNL
 TERMNL);

   /* end threaded mode */
   if (pn_end_threaded() != PN_SUCCESS)
   {
      printf("   Failure of function pn_end_threaded()." TERMNL);
      return TEST_FAIL;
   }

   printf("   Ended threaded mode successfully." TERMNL);

   /* destroying the lock should work */
   if (pn_spinlock_destroy(&lock) != PN_SUCCESS)
   {
      if (coreid == 0)
         printf("   Failure of function pn_spinlock_destroy()." TERMNL);
      return TEST_FAIL;
   }

   printf("   Destroyed lock successfully." TERMNL);

```

```
    /* check the testarray */
    memset(count_CPU, 0, (sizeof (int)) * NUMCORE_MIN);
    for (i = 0; i < ARRAYLENGTH; i++)
    {
       if ((s_testarray[i] > NUMCORE_MIN) || (s_testarray[i] < 1))
       {
          printf("   The test array contained wrong values:" TERMNL);
          print_testarray();
          return TEST_FAIL;
       }
       else
       {
          count_CPU[s_testarray[i] - 1]++;
       }
    }

    /* check the counters */
    for (i = 0; i < NUMCORE_MIN; i++)
    {
       if (count_CPU[i] != (ARRAYLENGTH/NUMCORE_MIN))
       {
          printf("   The test array core distribution is wrong." TERMNL);
          printf("   Counted %d entries by core with ID %d. Array:" TERMNL,
                                                        count_CPU[i], i);
          print_testarray();
          return TEST_FAIL;
       }
    }

    /* test was successful, print array */
    printf("   Test array was filled in correctly! Array:" TERMNL);
    print_testarray();

    printf(TERMNL);

    return TEST_SUCCESS;
 }

 #endif /* defined PN_WITH_SPINLOCK && defined PN_WITH_THREAD */

 /*EOF*****************************************************************************/

```

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.

Implicitely 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