Protothreads

The Protothreads Library 1.4 Reference Manual

June 2006

Adam Dunkels
adam@sics.se

Swedish Institute of Computer Science

Contents

1 The Protothreads Library

Author

Adam Dunkels adam@sics.se

Protothreads are a type of lightweight stackless threads designed for severly memory constrained systems such as deeply embedded systems or sensor network nodes. Protothreads provides linear code execution for event-driven systems implemented in C. Protothreads can be used with or without an RTOS.

Protothreads are a extremely lightweight, stackless type of threads that provides a blocking context on top of an event-driven system, without the overhead of per-thread stacks. The purpose of protothreads is to implement sequential flow of control without complex state machines or full multi-threading. Protothreads provides conditional blocking inside C functions.

Main features:

- No machine specific code - the protothreads library is pure C
- Does not use error-prone functions such as longjmp()
- Very small RAM overhead - only two bytes per protothread
- Can be used with or without an OS
- Provides blocking wait without full multi-threading or stack-switching

Examples applications:

- Memory constrained systems
- Event-driven protocol stacks
- Deeply embedded systems
- Sensor network nodes

See also

Example programs

Protothreads API documentation

More information and new version of the code can be found at the Protothreads homepage:

http://www.sics.se/~adam/pt/

1.1 Authors

The protothreads library was written by Adam Dunkels adam@sics.se with support from Oliver Schmidt ol.sc@web.de.

1.2 Using protothreads

Using protothreads in a project is easy: simply copy the files pt.h, lc.h and lc-switch.h into the include files directory of the project, and #include "pt.h" in all files that should use protothreads.

1.3 Protothreads

Protothreads are a extremely lightweight, stackless threads that provides a blocking context on top of an event-driven system, without the overhead of per-thread stacks. The purpose of protothreads is to implement sequential flow of control without using complex state machines or full multi-threading. Protothreads provides conditional blocking inside a C function.

In memory constrained systems, such as deeply embedded systems, traditional multi-threading may have a too large memory overhead. In traditional multi-threading, each thread requires its own stack, that typically is over-provisioned. The stacks may use large parts of the available memory.

The main advantage of protothreads over ordinary threads is that protothreads are very lightweight: a protothread does not require its own stack. Rather, all protothreads run on the same stack and context switching is done by stack rewinding. This is advantageous in memory constrained systems, where a stack for a thread might use a large part of the available memory. A protothread only requires only two bytes of memory per protothread. Moreover, protothreads are implemented in pure C and do not require any machine-specific assembler code.

A protothread runs within a single C function and cannot span over other functions. A protothread may call normal C functions, but cannot block inside a called function. Blocking inside nested function calls is instead made by spawning a separate protothread for each potentially blocking function. The advantage of this approach is that blocking is explicit: the programmer knows exactly which functions that block that which functions the never blocks.

Protothreads are similar to asymmetric co-routines. The main difference is that co-routines uses a separate stack for each co-routine, whereas protothreads are stackless. The most similar mechanism to protothreads are Python generators. These are also stackless constructs, but have a different purpose. Protothreads provides blocking contexts inside a C function, whereas Python generators provide multiple exit points from a generator function.

1.4 Local variables

Note

> Because protothreads do not save the stack context across a blocking call, local variables are not preserved when the protothread blocks. This means that local variables should be used with utmost care - if in doubt, do not use local variables inside a protothread!

1.5 Scheduling

A protothread is driven by repeated calls to the function in which the protothread is running. Each time the function is called, the protothread will run until it blocks or exits. Thus the scheduling of protothreads is done by the application that uses protothreads.

1.6 Implementation

Protothreads are implemented using local continuations. A local continuation represents the current state of execution at a particular place in the program, but does not provide any call history or local variables. A local continuation can be set in a specific function to capture the state of the function. After a local continuation has been set can be resumed in order to restore the state of the function at the point where the local continuation was set.

Local continuations can be implemented in a variety of ways:

1. by using machine specific assembler code,
2. by using standard C constructs, or
3. by using compiler extensions.

The first way works by saving and restoring the processor state, except for stack pointers, and requires between 16 and 32 bytes of memory per protothread. The exact amount of memory required depends on the architecture.

The standard C implementation requires only two bytes of state per protothread and utilizes the C switch() statement in a non-obvious way that is similar to Duff's device. This implementation does, however, impose a slight restriction to the code that uses protothreads: a protothread cannot perform a blocking wait (PT_WAIT_UNTIL() or PT_YIELD()) inside a switch() statement.

Certain compilers has C extensions that can be used to implement protothreads. GCC supports label pointers that can be used for this purpose. With this implementation, protothreads require 4 bytes of RAM per protothread.

2 Topic Index

2.1 Topics

Here is a list of all topics with brief descriptions:

3 Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

4 File Index

4.1 File List

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

5 Topic Documentation

5.1 Protothreads

Protothreads are implemented in a single header file, pt.h, which includes the local continuations header file, lc.h.

Topics

- **Protothread semaphores** 10

  This module implements counting semaphores on top of protothreads.

- **Local continuations** 13

  Local continuations form the basis for implementing protothreads.

Files

- file pt.h

  Protothreads implementation.

Data Structures

- struct pt

  Protothread control structure.

Protothread status codes

These values are returned by a protothread to indicate its current status.

- #define **PT_WAITING** 0

  The protothread is waiting for a condition.
- #define **PT_YIELDED** 1

  The protothread has yielded.
- #define **PT_EXITED** 2

  The protothread has exited via PT_EXIT().
- #define **PT_ENDED** 3

  The protothread has ended normally.

Initialization

- #define PT_INIT(pt)

  Initialize a protothread.

Declaration and definition

- #define PT_THREAD(name_args)

  Declaration of a protothread.
- #define PT_BEGIN(pt)

  Declare the start of a protothread inside the C function implementing the protothread.
- #define PT_END(pt)

  Declare the end of a protothread.

Blocked wait

- #define PT_WAIT_UNTIL(pt, condition)

  Block and wait until condition is true.
- #define PT_WAIT_WHILE(pt, cond)

  Block and wait while condition is true.

Hierarchical protothreads

- #define PT_WAIT_THREAD(pt, thread)

  Block and wait until a child protothread completes.
- #define PT_SPAWN(pt, child, thread)

  Spawn a child protothread and wait until it exits.

Exiting and restarting

- #define PT_RESTART(pt)

  Restart the protothread.
- #define PT_EXIT(pt)

  Exit the protothread.

Calling a protothread

- #define PT_SCHEDULE(f)

  Schedule a protothread.

Yielding from a protothread

- #define PT_YIELD(pt)

  Yield from the current protothread.
- #define PT_YIELD_UNTIL(pt, cond)

  Yield from the protothread until a condition occurs.

5.1.1 Detailed Description

Protothreads are implemented in a single header file, pt.h, which includes the local continuations header file, lc.h.

This file in turn includes the actual implementation of local continuations, which typically also is contained in a single header file.

5.1.2 Macro Definition Documentation

PT_BEGIN

```
#define PT_BEGIN(pt)
```

Declare the start of a protothread inside the C function implementing the protothread.

This macro is used to declare the starting point of a protothread. It should be placed at the start of the function in which the protothread runs. All C statements above the PT_BEGIN() invokation will be executed each time the protothread is scheduled.

Parameters

pt	A pointer to the protothread control structure.

Definition at line 141 of file pt.h.

PT_END

```
#define PT_END(pt)
```

Declare the end of a protothread.

This macro is used for declaring that a protothread ends. It must always be used together with a matching PT_BEGIN() macro.

Parameters

pt	A pointer to the protothread control structure.

Definition at line 153 of file pt.h.

```
00153 #define PT_END(pt) LC_END((pt)->lc); PT_YIELD_FLAG = 0; \
00154                    PT_INIT(pt); return PT_ENDED; }
```

PT_EXIT

```
#define PT_EXIT(pt)
```

Exit the protothread.

This macro causes the protothread to exit. If the protothread was spawned by another protothread, the parent protothread will become unblocked and can continue to run.

Parameters

pt	A pointer to the protothread control structure.

Definition at line 272 of file pt.h.

```
00272 #define PT_EXIT(pt)                             \
00273   do {                                          \
00274     PT_INIT(pt);                                \
00275     return PT_EXITED;                   \
00276   } while(0)
```

PT_INIT

```
#define PT_INIT(pt)
```

Initialize a protothread.

Initializes a protothread. Initialization must be done prior to starting to execute the protothread.

Parameters

pt	A pointer to the protothread control structure.

See also

PT_SPAWN()

Definition at line 106 of file pt.h.

PT_RESTART

```
#define PT_RESTART(pt)
```

Restart the protothread.

This macro will block and cause the running protothread to restart its execution at the place of the PT_BEGIN() call.

Parameters

pt	A pointer to the protothread control structure.

Definition at line 255 of file pt.h.

```
00255 #define PT_RESTART(pt)                          \
00256   do {                                          \
00257     PT_INIT(pt);                                \
00258     return PT_WAITING;                  \
00259   } while(0)
```

PT_SCHEDULE

```
#define PT_SCHEDULE(f)
```

Schedule a protothread.

This function shedules a protothread. The return value of the function is non-zero if the protothread is running or zero if the protothread has exited.

Parameters

f	The call to the C function implementing the protothread to be scheduled

Definition at line 297 of file pt.h.

PT_SPAWN

```
#define PT_SPAWN(pt, child, thread)
```

Spawn a child protothread and wait until it exits.

This macro spawns a child protothread and waits until it exits. The macro can only be used within a protothread.

Parameters

pt	A pointer to the protothread control structure.
child	A pointer to the child protothread's control structure.
thread	The child protothread with arguments

Definition at line 232 of file pt.h.

```
00232 #define PT_SPAWN(pt, child, thread)             \
00233   do {                                          \
00234     PT_INIT((child));                           \
00235     PT_WAIT_THREAD((pt), (thread));             \
00236   } while(0)
```

PT_THREAD

```
#define PT_THREAD(name_args)
```

Declaration of a protothread.

This macro is used to declare a protothread. All protothreads must be declared with this macro.

Parameters

name_args	The name and arguments of the C function implementing the protothread.

Definition at line 126 of file pt.h.

PT_WAIT_THREAD

```
#define PT_WAIT_THREAD(pt, thread)
```

Block and wait until a child protothread completes.

This macro schedules a child protothread. The current protothread will block until the child protothread completes.

Note

The child protothread must be manually initialized with the PT_INIT() function before this function is used.

Parameters

pt	A pointer to the protothread control structure.
thread	The child protothread with arguments

See also

PT_SPAWN()

Definition at line 218 of file pt.h.

PT_WAIT_UNTIL

```
#define PT_WAIT_UNTIL(pt, condition)
```

Block and wait until condition is true.

This macro blocks the protothread until the specified condition is true.

Parameters

pt	A pointer to the protothread control structure.
condition	The condition.

Definition at line 174 of file pt.h.

```
00174 #define PT_WAIT_UNTIL(pt, condition)           \
00175   do {                                          \
00176     LC_SET((pt)->lc);                           \
00177     if(!(condition)) {                          \
00178       return PT_WAITING;                        \
00179     }                                           \
00180   } while(0)
```

PT_WAIT_WHILE

```
#define PT_WAIT_WHILE(pt, cond)
```

Block and wait while condition is true.

This function blocks and waits while condition is true. See PT_WAIT_UNTIL().

Parameters

pt	A pointer to the protothread control structure.
cond	The condition.

Definition at line 193 of file pt.h.

PT_YIELD

```
#define PT_YIELD(pt)
```

Yield from the current protothread.

This function will yield the protothread, thereby allowing other processing to take place in the system.

Parameters

pt	A pointer to the protothread control structure.

Definition at line 316 of file pt.h.

```
00316 #define PT_YIELD(pt)                            \
00317   do {                                          \
00318     PT_YIELD_FLAG = 0;                          \
00319     LC_SET((pt)->lc);                           \
00320     if(PT_YIELD_FLAG == 0) {                    \
00321       return PT_YIELDED;                        \
00322     }                                           \
00323   } while(0)
```

PT_YIELD_UNTIL

```
#define PT_YIELD_UNTIL(pt, cond)
```

Yield from the protothread until a condition occurs.

Parameters

pt	A pointer to the protothread control structure.
cond	The condition. This function will yield the protothread, until the specified condition evaluates to true.

Definition at line 336 of file pt.h.

```
00336 #define PT_YIELD_UNTIL(pt, cond)                \
00337   do {                                          \
00338     PT_YIELD_FLAG = 0;                          \
00339     LC_SET((pt)->lc);                           \
00340     if((PT_YIELD_FLAG == 0) || !(cond)) {       \
00341       return PT_YIELDED;                        \
00342     }                                           \
00343   } while(0)
```

5.1.3 Protothread semaphores

This module implements counting semaphores on top of protothreads.

Files

- file pt-sem.h

  Couting semaphores implemented on protothreads.

Data Structures

- struct pt_sem

  Semaphore control structure.

Macros

- #define PT_SEM_INIT(s, c)

  Initialize a semaphore.
- #define PT_SEM_WAIT(pt, s)

  Wait for a semaphore.
- #define PT_SEM_SIGNAL(pt, s)

  Signal a semaphore.

Detailed Description

This module implements counting semaphores on top of protothreads.

Semaphores are a synchronization primitive that provide two operations: "wait" and "signal". The "wait" operation checks the semaphore counter and blocks the thread if the counter is zero. The "signal" operation increases the semaphore counter but does not block. If another thread has blocked waiting for the semaphore that is signalled, the blocked thread will become runnable again.

Semaphores can be used to implement other, more structured, synchronization primitives such as monitors and message queues/bounded buffers (see below).

The following example shows how the producer-consumer problem, also known as the bounded buffer problem, can be solved using protothreads and semaphores. Notes on the program follow after the example.

```
#include "pt-sem.h"

#define NUM_ITEMS 32
#define BUFSIZE 8

static struct pt_sem mutex, full, empty;

PT_THREAD(producer(struct pt *pt))
{
  static int produced;

  PT_BEGIN(pt);

  for(produced = 0; produced < NUM_ITEMS; ++produced) {

    PT_SEM_WAIT(pt, &full);

    PT_SEM_WAIT(pt, &mutex);
    add_to_buffer(produce_item());
    PT_SEM_SIGNAL(pt, &mutex);

    PT_SEM_SIGNAL(pt, &empty);
  }

  PT_END(pt);
}

PT_THREAD(consumer(struct pt *pt))
{
  static int consumed;

  PT_BEGIN(pt);

  for(consumed = 0; consumed < NUM_ITEMS; ++consumed) {

    PT_SEM_WAIT(pt, &empty);

    PT_SEM_WAIT(pt, &mutex);
    consume_item(get_from_buffer());
    PT_SEM_SIGNAL(pt, &mutex);

    PT_SEM_SIGNAL(pt, &full);
  }

  PT_END(pt);
}

PT_THREAD(driver_thread(struct pt *pt))
{
  static struct pt pt_producer, pt_consumer;

  PT_BEGIN(pt);
```

```
  PT_SEM_INIT(&empty, 0);
  PT_SEM_INIT(&full, BUFSIZE);
  PT_SEM_INIT(&mutex, 1);

  PT_INIT(&pt_producer);
  PT_INIT(&pt_consumer);

  PT_WAIT_THREAD(pt, producer(&pt_producer) &
                     consumer(&pt_consumer));

  PT_END(pt);
}
```

The program uses three protothreads: one protothread that implements the consumer, one thread that implements the producer, and one protothread that drives the two other protothreads. The program uses three semaphores: "full", "empty" and "mutex". The "mutex" semaphore is used to provide mutual exclusion for the buffer, the "empty" semaphore is used to block the consumer is the buffer is empty, and the "full" semaphore is used to block the producer is the buffer is full.

The "driver_thread" holds two protothread state variables, "pt_producer" and "pt_consumer". It is important to note that both these variables are declared as *static*. If the static keyword is not used, both variables are stored on the stack. Since protothreads do not store the stack, these variables may be overwritten during a protothread wait operation. Similarly, both the "consumer" and "producer" protothreads declare their local variables as static, to avoid them being stored on the stack.

Macro Definition Documentation

PT_SEM_INIT

```
#define PT_SEM_INIT(s, c)
```

Initialize a semaphore.

This macro initializes a semaphore with a value for the counter. Internally, the semaphores use an "unsigned int" to represent the counter, and therefore the "count" argument should be within range of an unsigned int.

Parameters

s	(struct pt_sem ∗) A pointer to the pt_sem struct representing the semaphore
c	(unsigned int) The initial count of the semaphore.

Definition at line 193 of file pt-sem.h.

PT_SEM_SIGNAL

```
#define PT_SEM_SIGNAL(pt, s)
```

Signal a semaphore.

This macro carries out the "signal" operation on the semaphore. The signal operation increments the counter inside the semaphore, which eventually will cause waiting protothreads to continue executing.

Parameters

pt	(struct pt ∗) A pointer to the protothread (struct pt) in which the operation is executed.
s	(struct pt_sem ∗) A pointer to the pt_sem struct representing the semaphore

Definition at line 232 of file pt-sem.h.

PT_SEM_WAIT

```
#define PT_SEM_WAIT(pt, s)
```

Wait for a semaphore.

This macro carries out the "wait" operation on the semaphore. The wait operation causes the protothread to block while the counter is zero. When the counter reaches a value larger than zero, the protothread will continue.

Parameters

pt	(struct pt *) A pointer to the protothread (struct pt) in which the operation is executed.
s	(struct pt_sem *) A pointer to the pt_sem struct representing the semaphore

Definition at line 211 of file pt-sem.h.

```
00211 #define PT_SEM_WAIT(pt, s)      \
00212   do {                                    \
00213     PT_WAIT_UNTIL(pt, (s)->count > 0);    \
00214     --(s)->count;                         \
00215   } while(0)
```

5.1.4 Local continuations

Local continuations form the basis for implementing protothreads.

Files

- file lc.h

  Local continuations.
- file lc-switch.h

  Implementation of local continuations based on switch() statment.
- file lc-addrlabels.h

  Implementation of local continuations based on the "Labels as values" feature of gcc.

Macros

- #define LC_INIT(lc)

  Initialize a local continuation.
- #define LC_SET(lc)

  Set a local continuation.
- #define LC_RESUME(lc)

  Resume a local continuation.
- #define LC_END(lc)

  Mark the end of local continuation usage.

Typedefs

- typedef unsigned short **lc_t**

  The local continuation type.

Detailed Description

Local continuations form the basis for implementing protothreads.

A local continuation can be *set* in a specific function to capture the state of the function. After a local continuation has been set can be *resumed* in order to restore the state of the function at the point where the local continuation was set.

Macro Definition Documentation

LC_END

```
#define LC_END(lc)
```

Mark the end of local continuation usage.

The end operation signifies that local continuations should not be used any more in the function. This operation is not needed for most implementations of local continuation, but is required by a few implementations.

Definition at line 108 of file lc.h.

LC_INIT

```
#define LC_INIT(lc)
```

Initialize a local continuation.

This operation initializes the local continuation, thereby unsetting any previously set continuation state.

Definition at line 71 of file lc.h.

LC_RESUME

```
#define LC_RESUME(lc)
```

Resume a local continuation.

The resume operation resumes a previously set local continuation, thus restoring the state in which the function was when the local continuation was set. If the local continuation has not been previously set, the resume operation does nothing.

Definition at line 96 of file lc.h.

LC_SET

```
#define LC_SET(lc)
```

Set a local continuation.

The set operation saves the state of the function at the point where the operation is executed. As far as the set operation is concerned, the state of the function does **not** include the call-stack or local (automatic) variables, but only the program counter and such CPU registers that needs to be saved.

Definition at line 84 of file lc.h.

5.2 Examples

5.2.1 A small example

This first example shows a very simple program: two protothreads waiting for each other to toggle two flags. The code illustrates how to write protothreads code, how to initialize protothreads, and how to schedule them.

```
/**
 * This is a very small example that shows how to use
 * protothreads. The program consists of two protothreads that wait
 * for each other to toggle a variable.
 */

/* We must always include pt.h in our protothreads code. */
#include "pt.h"

#include <stdio.h> /* For printf(). */

/* Two flags that the two protothread functions use. */
static int protothread1_flag, protothread2_flag;

/**
 * The first protothread function. A protothread function must always
 * return an integer, but must never explicitly return - returning is
 * performed inside the protothread statements.
 *
 * The protothread function is driven by the main loop further down in
 * the code.
 */
```

```
static int
protothread1(struct pt *pt)
{
  /* A protothread function must begin with PT_BEGIN() which takes a
     pointer to a struct pt. */
  PT_BEGIN(pt);

  /* We loop forever here. */
  while(1) {
    /* Wait until the other protothread has set its flag. */
    PT_WAIT_UNTIL(pt, protothread2_flag != 0);
    printf("Protothread 1 running\n");

    /* We then reset the other protothread's flag, and set our own
       flag so that the other protothread can run. */
    protothread2_flag = 0;
    protothread1_flag = 1;

    /* And we loop. */
  }

  /* All protothread functions must end with PT_END() which takes a
     pointer to a struct pt. */
  PT_END(pt);
}

/**
 * The second protothread function. This is almost the same as the
 * first one.
 */
static int
protothread2(struct pt *pt)
{
  PT_BEGIN(pt);

  while(1) {
    /* Let the other protothread run. */
    protothread2_flag = 1;

    /* Wait until the other protothread has set its flag. */
    PT_WAIT_UNTIL(pt, protothread1_flag != 0);
    printf("Protothread 2 running\n");

    /* We then reset the other protothread's flag. */
    protothread1_flag = 0;

    /* And we loop. */
  }
  PT_END(pt);
}

/**
 * Finally, we have the main loop. Here is where the protothreads are
 * initialized and scheduled. First, however, we define the
 * protothread state variables pt1 and pt2, which hold the state of
 * the two protothreads.
 */
static struct pt pt1, pt2;
int
main(void)
{
  /* Initialize the protothread state variables with PT_INIT(). */
  PT_INIT(&pt1);
  PT_INIT(&pt2);

  /*
   * Then we schedule the two protothreads by repeatedly calling their
   * protothread functions and passing a pointer to the protothread
   * state variables as arguments.
   */
  while(1) {
    protothread1(&pt1);
    protothread2(&pt2);
  }
}
```

5.2.2 A code-lock

This example shows how to implement a simple code lock - the kind of device that is placed next to doors and that you have to push a four digit number into in order to unlock the door.

The code lock waits for key presses from a numeric keyboard and if the correct code is entered, the lock

is unlocked. There is a maximum time of one second between each key press, and after the correct code has been entered, no more keys must be pressed for 0.5 seconds before the lock is opened.

```
/*
 * Copyright (c) 2004-2005, Swedish Institute of Computer Science.
 * All rights reserved.
 *
 * 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.
 * 3. Neither the name of the Institute nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE INSTITUTE 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 INSTITUTE 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.
 *
 * This file is part of the protothreads library.
 *
 * Author: Adam Dunkels <adam@sics.se>
 *
 * $Id: example-codelock.c,v 1.5 2005/10/06 07:57:08 adam Exp $
 */

/*
 *
 * This example shows how to implement a simple code lock. The code
 * lock waits for key presses from a numeric keyboard and if the
 * correct code is entered, the lock is unlocked. There is a maximum
 * time of one second between each key press, and after the correct
 * code has been entered, no more keys must be pressed for 0.5 seconds
 * before the lock is opened.
 *
 * This is an example that shows two things:
 * - how to implement a code lock key input mechanism, and
 * - how to implement a sequential timed routine.
 *
 * The program consists of two protothreads, one that implements the
 * code lock reader and one that implements simulated keyboard input.
 *
 *
 */

#ifdef _WIN32
#include <windows.h>
#else
#include <unistd.h>
#include <sys/time.h>
#endif
#include <stdio.h>

#include "pt.h"

/*---------------------------------------------------------------------------*/
/*
 * The following definitions are just for the simple timer library
 * used in this example. The actual implementation of the functions
 * can be found at the end of this file.
 */
struct timer { int start, interval; };
static int  timer_expired(struct timer *t);
static void timer_set(struct timer *t, int usecs);
/*---------------------------------------------------------------------------*/
/*
 * This example uses two timers: one for the code lock protothread and
 * one for the simulated key input protothread.
 */
static struct timer codelock_timer, input_timer;
/*---------------------------------------------------------------------------*/
/*
```

```
 * This is the code that has to be entered.
 */
static const char code[4] = {'1', '4', '2', '3'};
/*---------------------------------------------------------------------------*/
/*
 * This example has two protothread and therefor has two protothread
 * control structures of type struct pt. These are initialized with
 * PT_INIT() in the main() function below.
 */
static struct pt codelock_pt, input_pt;
/*---------------------------------------------------------------------------*/
/*
 * The following code implements a simple key input. Input is made
 * with the press_key() function, and the function key_pressed()
 * checks if a key has been pressed. The variable "key" holds the
 * latest key that was pressed. The variable "key_pressed_flag" is set
 * when a key is pressed and cleared when a key press is checked.
 */
static char key, key_pressed_flag;

static void
press_key(char k)
{
  printf("--- Key '%c' pressed\n", k);
  key = k;
  key_pressed_flag = 1;
}

static int
key_pressed(void)
{
  if(key_pressed_flag != 0) {
    key_pressed_flag = 0;
    return 1;
  }
  return 0;
}
/*---------------------------------------------------------------------------*/
/*
 * Declaration of the protothread function implementing the code lock
 * logic. The protothread function is declared using the PT_THREAD()
 * macro. The function is declared with the "static" keyword since it
 * is local to this file. The name of the function is codelock_thread
 * and it takes one argument, pt, of the type struct pt.
 *
 */
static
PT_THREAD(codelock_thread(struct pt *pt))
{
  /* This is a local variable that holds the number of keys that have
   * been pressed. Note that it is declared with the "static" keyword
   * to make sure that the variable is *not* allocated on the stack.
   */
  static int keys;

  /*
   * Declare the beginning of the protothread.
   */
  PT_BEGIN(pt);

  /*
   * We'll let the protothread loop until the protothread is
   * explicitly exited with PT_EXIT().
   */
  while(1) {

    /*
     * We'll be reading key presses until we get the right amount of
     * correct keys.
     */
    for(keys = 0; keys < sizeof(code); ++keys) {

      /*
       * If we haven't gotten any keypresses, we'll simply wait for one.
       */
      if(keys == 0) {

        /*
         * The PT_WAIT_UNTIL() function will block until the condition
         * key_pressed() is true.
         */
        PT_WAIT_UNTIL(pt, key_pressed());
      } else {
```

```
      /*
       * If the "key" variable was larger than zero, we have already
       * gotten at least one correct key press. If so, we'll not
       * only wait for the next key, but we'll also set a timer that
       * expires in one second. This gives the person pressing the
       * keys one second to press the next key in the code.
       */
      timer_set(&codelock_timer, 1000);

      /*
       * The following statement shows how complex blocking
       * conditions can be easily expressed with protothreads and
       * the PT_WAIT_UNTIL() function.
       */
      PT_WAIT_UNTIL(pt, key_pressed() || timer_expired(&codelock_timer));

      /*
       * If the timer expired, we should break out of the for() loop
       * and start reading keys from the beginning of the while(1)
       * loop instead.
       */
      if(timer_expired(&codelock_timer)) {
        printf("Code lock timer expired.\n");

        /*
         * Break out from the for() loop and start from the
         * beginning of the while(1) loop.
         */
        break;
      }
    }

    /*
     * Check if the pressed key was correct.
     */
    if(key != code[keys]) {
      printf("Incorrect key '%c' found\n", key);
      /*
       * Break out of the for() loop since the key was incorrect.
       */
      break;
    } else {
      printf("Correct key '%c' found\n", key);
    }
  }

  /*
   * Check if we have gotten all keys.
   */
  if(keys == sizeof(code)) {
    printf("Correct code entered, waiting for 500 ms before unlocking.\n");

    /*
     * Ok, we got the correct code. But to make sure that the code
     * was not just a fluke of luck by an intruder, but the correct
     * code entered by a person that knows the correct code, we'll
     * wait for half a second before opening the lock. If another
     * key is pressed during this time, we'll assume that it was a
     * fluke of luck that the correct code was entered the first
     * time.
     */
    timer_set(&codelock_timer, 500);
    PT_WAIT_UNTIL(pt, key_pressed() || timer_expired(&codelock_timer));

    /*
     * If we continued from the PT_WAIT_UNTIL() statement without
     * the timer expired, we don't open the lock.
     */
    if(!timer_expired(&codelock_timer)) {
      printf("Key pressed during final wait, code lock locked again.\n");
    } else {

      /*
       * If the timer expired, we'll open the lock and exit from the
       * protothread.
       */
      printf("Code lock unlocked.\n");
      PT_EXIT(pt);
    }
  }
}

/*
 * Finally, we'll mark the end of the protothread.
```

```
   */
  PT_END(pt);
}
/*---------------------------------------------------------------------------*/
/*
 * This is the second protothread in this example. It implements a
 * simulated user pressing the keys. This illustrates how a linear
 * sequence of timed instructions can be implemented with
 * protothreads.
 */
static
PT_THREAD(input_thread(struct pt *pt))
{
  PT_BEGIN(pt);

  printf("Waiting 1 second before entering first key.\n");

  timer_set(&input_timer, 1000);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('1');

  timer_set(&input_timer, 100);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('2');

  timer_set(&input_timer, 100);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('3');

  timer_set(&input_timer, 2000);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('1');

  timer_set(&input_timer, 200);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('4');

  timer_set(&input_timer, 200);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('2');

  timer_set(&input_timer, 2000);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('3');

  timer_set(&input_timer, 200);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('1');

  timer_set(&input_timer, 200);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('4');

  timer_set(&input_timer, 200);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('2');

  timer_set(&input_timer, 100);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('3');

  timer_set(&input_timer, 100);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('4');

  timer_set(&input_timer, 1500);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('1');

  timer_set(&input_timer, 300);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));
```

```
  press_key('4');

  timer_set(&input_timer, 400);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('2');

  timer_set(&input_timer, 500);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  press_key('3');

  timer_set(&input_timer, 2000);
  PT_WAIT_UNTIL(pt, timer_expired(&input_timer));

  PT_END(pt);
}
/*---------------------------------------------------------------------------*/
/*
 * This is the main function. It initializes the two protothread
 * control structures and schedules the two protothreads. The main
 * function returns when the protothread the runs the code lock exits.
 */
int
main(void)
{
  /*
   * Initialize the two protothread control structures.
   */
  PT_INIT(&input_pt);
  PT_INIT(&codelock_pt);

  /*
   * Schedule the two protothreads until the codelock_thread() exits.
   */
  while(PT_SCHEDULE(codelock_thread(&codelock_pt))) {
    PT_SCHEDULE(input_thread(&input_pt));

    /*
     * When running this example on a multitasking system, we must
     * give other processes a chance to run too and therefore we call
     * usleep() resp. Sleep() here. On a dedicated embedded system,
     * we usually do not need to do this.
     */
#ifdef _WIN32
    Sleep(0);
#else
    usleep(10);
#endif
  }

  return 0;
}
/*---------------------------------------------------------------------------*/
/*
 * Finally, the implementation of the simple timer library follows.
 */
#ifdef _WIN32

static int clock_time(void)
{ return (int)GetTickCount(); }

#else /* _WIN32 */

static int clock_time(void)
{
  struct timeval tv;
  struct timezone tz;
  gettimeofday(&tv, &tz);
  return tv.tv_sec * 1000 + tv.tv_usec / 1000;
}

#endif /* _WIN32 */

static int timer_expired(struct timer *t)
{ return (int)(clock_time() - t->start) >= (int)t->interval; }

static void timer_set(struct timer *t, int interval)
{ t->interval = interval; t->start = clock_time(); }
/*---------------------------------------------------------------------------*/
```

5.2.3 The bounded buffer with protothread semaphores

The following example shows how to implement the bounded buffer problem using the protothreads semaphore library. The example uses three protothreads: one producer() protothread that produces items, one consumer() protothread that consumes items, and one driver_thread() that schedules the producer and consumer protothreads.

Note that there is no need for a mutex to guard the add_to_buffer() and get_from_buffer() functions because of the implicit locking semantics of protothreads - a protothread will never be preempted and will never block except in an explicit PT_WAIT statement.

```
/*
 * Copyright (c) 2004-2005, Swedish Institute of Computer Science.
 * All rights reserved.
 *
 * 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.
 * 3. Neither the name of the Institute nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE INSTITUTE 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 INSTITUTE 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.
 *
 * This file is part of the protothreads library.
 *
 * Author: Adam Dunkels <adam@sics.se>
 *
 * $Id: example-buffer.c,v 1.5 2005/10/07 05:21:33 adam Exp $
 */

#ifdef _WIN32
#include <windows.h>
#else
#include <unistd.h>
#endif
#include <stdio.h>

#include "pt-sem.h"
#define NUM_ITEMS 32
#define BUFSIZE 8

static int buffer[BUFSIZE];
static int bufptr;

static void
add_to_buffer(int item)
{
  printf("Item %d added to buffer at place %d\n", item, bufptr);
  buffer[bufptr] = item;
  bufptr = (bufptr + 1) % BUFSIZE;
}
static int
get_from_buffer(void)
{
  int item;
  item = buffer[bufptr];
  printf("Item %d retrieved from buffer at place %d\n",
         item, bufptr);
  bufptr = (bufptr + 1) % BUFSIZE;
  return item;
}

static int
produce_item(void)
{
  static int item = 0;
```

```
  printf("Item %d produced\n", item);
  return item++;
}

static void
consume_item(int item)
{
  printf("Item %d consumed\n", item);
}

static struct pt_sem full, empty;
static
PT_THREAD(producer(struct pt *pt))
{
  static int produced;

  PT_BEGIN(pt);

  for(produced = 0; produced < NUM_ITEMS; ++produced) {

    PT_SEM_WAIT(pt, &full);

    add_to_buffer(produce_item());

    PT_SEM_SIGNAL(pt, &empty);
  }

  PT_END(pt);
}
static
PT_THREAD(consumer(struct pt *pt))
{
  static int consumed;

  PT_BEGIN(pt);
  for(consumed = 0; consumed < NUM_ITEMS; ++consumed) {

    PT_SEM_WAIT(pt, &empty);

    consume_item(get_from_buffer());

    PT_SEM_SIGNAL(pt, &full);
  }
  PT_END(pt);
}
static
PT_THREAD(driver_thread(struct pt *pt))
{
  static struct pt pt_producer, pt_consumer;
  PT_BEGIN(pt);

  PT_SEM_INIT(&empty, 0);
  PT_SEM_INIT(&full, BUFSIZE);
  PT_INIT(&pt_producer);
  PT_INIT(&pt_consumer);
  PT_WAIT_THREAD(pt, producer(&pt_producer) &
                     consumer(&pt_consumer));
  PT_END(pt);
}


int
main(void)
{
  struct pt driver_pt;

  PT_INIT(&driver_pt);

  while(PT_SCHEDULE(driver_thread(&driver_pt))) {

    /*
     * When running this example on a multitasking system, we must
     * give other processes a chance to run too and therefore we call
     * usleep() resp. Sleep() here. On a dedicated embedded system,
     * we usually do not need to do this.
     */
#ifdef _WIN32
    Sleep(0);
#else
    usleep(10);
#endif
  }
  return 0;
}
```

6 Data Structure Documentation

6.1 pt Struct Reference

Protothread control structure.
`#include <pt.h>`

6.1.1 Detailed Description

Protothread control structure.

This structure is used to store the state of a protothread. It should be declared as a local variable or as a member of a larger structure. The contents of this structure are internal to the protothread library and should not be accessed directly by the user.

See also

PT_INIT()

Definition at line 64 of file pt.h.

6.2 pt_sem Struct Reference

Semaphore control structure.
`#include <pt-sem.h>`

6.2.1 Detailed Description

Semaphore control structure.

This structure represents a counting semaphore for use with protothreads. It should be declared as a static variable and initialized with PT_SEM_INIT() before use. The contents of this structure are internal to the semaphore implementation and should not be accessed directly by the user.

See also

PT_SEM_INIT(), PT_SEM_WAIT(), PT_SEM_SIGNAL()

Definition at line 175 of file pt-sem.h.

7 File Documentation

7.1 lc-addrlabels.h File Reference

Implementation of local continuations based on the "Labels as values" feature of gcc.

7.1.1 Detailed Description

Implementation of local continuations based on the "Labels as values" feature of gcc.

Author

Adam Dunkels `adam@sics.se`

This implementation of local continuations is based on a special feature of the GCC C compiler called "labels as values". This feature allows assigning pointers with the address of the code corresponding to a particular C label.

For more information, see the GCC documentation: `http://gcc.gnu.org/onlinedocs/gcc/Labels-as-Value`

Definition in file lc-addrlabels.h.

7.2 lc-addrlabels.h

Go to the documentation of this file.

```
00001 /*
00002  * Copyright (c) 2004-2005, Swedish Institute of Computer Science.
00003  * All rights reserved.
00004  *
00005  * Redistribution and use in source and binary forms, with or without
00006  * modification, are permitted provided that the following conditions
00007  * are met:
00008  * 1. Redistributions of source code must retain the above copyright
00009  *    notice, this list of conditions and the following disclaimer.
00010  * 2. Redistributions in binary form must reproduce the above copyright
00011  *    notice, this list of conditions and the following disclaimer in the
00012  *    documentation and/or other materials provided with the distribution.
00013  * 3. Neither the name of the Institute nor the names of its contributors
00014  *    may be used to endorse or promote products derived from this software
00015  *    without specific prior written permission.
00016  *
00017  * THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS ``AS IS'' AND
00018  * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00019  * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00020  * ARE DISCLAIMED.  IN NO EVENT SHALL THE INSTITUTE OR CONTRIBUTORS BE LIABLE
00021  * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
00022  * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
00023  * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
00024  * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
00025  * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
00026  * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
00027  * SUCH DAMAGE.
00028  *
00029  * This file is part of the Contiki operating system.
00030  *
00031  * Author: Adam Dunkels <adam@sics.se>
00032  *
00033  * $Id: lc-addrlabels.h,v 1.4 2006/06/03 11:29:43 adam Exp $
00034  */
00035
00036 /**
00037  * \addtogroup lc
00038  * @{
00039  */
00040
00041 /**
00042  * \file
00043  * Implementation of local continuations based on the "Labels as
00044  * values" feature of gcc
00045  * \author
00046  * Adam Dunkels <adam@sics.se>
00047  *
00048  * This implementation of local continuations is based on a special
00049  * feature of the GCC C compiler called "labels as values". This
00050  * feature allows assigning pointers with the address of the code
00051  * corresponding to a particular C label.
00052  *
00053  * For more information, see the GCC documentation:
00054  * http://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html
00055  *
00056  */
00057
00058 #ifndef __LC_ADDRLABELS_H__
00059 #define __LC_ADDRLABELS_H__
00060
00061 /** \hideinitializer */
00062 typedef void * lc_t;
00063
00064 #define LC_INIT(s) s = NULL
00065
00066 #define LC_RESUME(s)                            \
00067   do {                                          \
00068     if(s != NULL) {                             \
00069       goto *s;                                  \
00070     }                                           \
00071   } while(0)
00072
00073 #define LC_CONCAT2(s1, s2) s1##s2
00074 #define LC_CONCAT(s1, s2) LC_CONCAT2(s1, s2)
00075
00076 #define LC_SET(s)                               \
00077   do {                                          \
00078     LC_CONCAT(LC_LABEL, __LINE__):              \
00079     (s) = &&LC_CONCAT(LC_LABEL, __LINE__);      \
00080   } while(0)
```

```
00081
00082 #define LC_END(s)
00083
00084 #endif /* __LC_ADDRLABELS_H__ */
00085 /** @} */
```

7.3 lc-switch.h File Reference

Implementation of local continuations based on switch() statment.

Typedefs

- typedef unsigned short **lc_t**

  The local continuation type.

7.3.1 Detailed Description

Implementation of local continuations based on switch() statment.

Author

Adam Dunkels adam@sics.se

This implementation of local continuations uses the C switch() statement to resume execution of a function somewhere inside the function's body. The implementation is based on the fact that switch() statements are able to jump directly into the bodies of control structures such as if() or while() statmenets.

This implementation borrows heavily from Simon Tatham's coroutines implementation in C: http://www.chiark.gree

Definition in file lc-switch.h.

7.4 lc-switch.h

Go to the documentation of this file.

```
00001 /*
00002  * Copyright (c) 2004-2005, Swedish Institute of Computer Science.
00003  * All rights reserved.
00004  *
00005  * Redistribution and use in source and binary forms, with or without
00006  * modification, are permitted provided that the following conditions
00007  * are met:
00008  * 1. Redistributions of source code must retain the above copyright
00009  *    notice, this list of conditions and the following disclaimer.
00010  * 2. Redistributions in binary form must reproduce the above copyright
00011  *    notice, this list of conditions and the following disclaimer in the
00012  *    documentation and/or other materials provided with the distribution.
00013  * 3. Neither the name of the Institute nor the names of its contributors
00014  *    may be used to endorse or promote products derived from this software
00015  *    without specific prior written permission.
00016  *
00017  * THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS ``AS IS'' AND
00018  * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00019  * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00020  * ARE DISCLAIMED.  IN NO EVENT SHALL THE INSTITUTE OR CONTRIBUTORS BE LIABLE
00021  * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
00022  * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
00023  * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
00024  * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
00025  * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
00026  * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
00027  * SUCH DAMAGE.
00028  *
00029  * This file is part of the Contiki operating system.
00030  *
00031  * Author: Adam Dunkels <adam@sics.se>
00032  *
00033  * $Id: lc-switch.h,v 1.4 2006/06/03 11:29:43 adam Exp $
00034  */
00035
00036 /**
00037  * \addtogroup lc
00038  * @{
00039  */
00040
```

```
00041 /**
00042  * \file
00043  * Implementation of local continuations based on switch() statment
00044  * \author Adam Dunkels <adam@sics.se>
00045  *
00046  * This implementation of local continuations uses the C switch()
00047  * statement to resume execution of a function somewhere inside the
00048  * function's body. The implementation is based on the fact that
00049  * switch() statements are able to jump directly into the bodies of
00050  * control structures such as if() or while() statmenets.
00051  *
00052  * This implementation borrows heavily from Simon Tatham's coroutines
00053  * implementation in C:
00054  * http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html
00055  */
00056
00057 #ifndef __LC_SWITCH_H__
00058 #define __LC_SWITCH_H__
00059
00060 /* WARNING! lc implementation using switch() does not work if an
00061    LC_SET() is done within another switch() statement! */
00062
00063 /** \hideinitializer */
00064 typedef unsigned short lc_t;
00065
00066 #define LC_INIT(s) s = 0;
00067
00068 #define LC_RESUME(s) switch(s) { case 0:
00069
00070 #define LC_SET(s) s = __LINE__; case __LINE__:
00071
00072 #define LC_END(s) }
00073
00074 #endif /* __LC_SWITCH_H__ */
00075
00076 /** @} */
```

7.5 lc.h File Reference

Local continuations.

```
#include "lc-switch.h"
```

Macros

- #define LC_INIT(lc)

  Initialize a local continuation.
- #define LC_SET(lc)

  Set a local continuation.
- #define LC_RESUME(lc)

  Resume a local continuation.
- #define LC_END(lc)

  Mark the end of local continuation usage.

7.5.1 Detailed Description

Local continuations.

Author

Adam Dunkels adam@sics.se

Definition in file lc.h.

7.6 lc.h

Go to the documentation of this file.

```
00001 /*
00002  * Copyright (c) 2004-2005, Swedish Institute of Computer Science.
00003  * All rights reserved.
00004  *
00005  * Redistribution and use in source and binary forms, with or without
00006  * modification, are permitted provided that the following conditions
00007  * are met:
00008  * 1. Redistributions of source code must retain the above copyright
00009  *    notice, this list of conditions and the following disclaimer.
00010  * 2. Redistributions in binary form must reproduce the above copyright
00011  *    notice, this list of conditions and the following disclaimer in the
00012  *    documentation and/or other materials provided with the distribution.
00013  * 3. Neither the name of the Institute nor the names of its contributors
00014  *    may be used to endorse or promote products derived from this software
00015  *    without specific prior written permission.
00016  *
00017  * THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS ``AS IS'' AND
00018  * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00019  * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00020  * ARE DISCLAIMED.  IN NO EVENT SHALL THE INSTITUTE OR CONTRIBUTORS BE LIABLE
00021  * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
00022  * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
00023  * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
00024  * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
00025  * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
00026  * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
00027  * SUCH DAMAGE.
00028  *
00029  * This file is part of the protothreads library.
00030  *
00031  * Author: Adam Dunkels <adam@sics.se>
00032  *
00033  * $Id: lc.h,v 1.2 2005/02/24 10:36:59 adam Exp $
00034  */
00035
00036 /**
00037  * \addtogroup pt
00038  * @{
00039  */
00040
00041 /**
00042  * \defgroup lc Local continuations
00043  * @{
00044  *
00045  * Local continuations form the basis for implementing protothreads. A
00046  * local continuation can be <i>set</i> in a specific function to
00047  * capture the state of the function. After a local continuation has
00048  * been set can be <i>resumed</i> in order to restore the state of the
00049  * function at the point where the local continuation was set.
00050  *
00051  *
00052  */
00053
00054 /**
00055  * \file lc.h
00056  * Local continuations
00057  * \author
00058  * Adam Dunkels <adam@sics.se>
00059  *
00060  */
00061
00062 #ifdef DOXYGEN
00063 /**
00064  * Initialize a local continuation.
00065  *
00066  * This operation initializes the local continuation, thereby
00067  * unsetting any previously set continuation state.
00068  *
00069  * \hideinitializer
00070  */
00071 #define LC_INIT(lc)
00072
00073 /**
00074  * Set a local continuation.
00075  *
00076  * The set operation saves the state of the function at the point
00077  * where the operation is executed. As far as the set operation is
00078  * concerned, the state of the function does <b>not</b> include the
00079  * call-stack or local (automatic) variables, but only the program
00080  * counter and such CPU registers that needs to be saved.
```

```
00081  *
00082  * \hideinitializer
00083  */
00084 #define LC_SET(lc)
00085
00086 /**
00087  * Resume a local continuation.
00088  *
00089  * The resume operation resumes a previously set local continuation, thus
00090  * restoring the state in which the function was when the local
00091  * continuation was set. If the local continuation has not been
00092  * previously set, the resume operation does nothing.
00093  *
00094  * \hideinitializer
00095  */
00096 #define LC_RESUME(lc)
00097
00098 /**
00099  * Mark the end of local continuation usage.
00100  *
00101  * The end operation signifies that local continuations should not be
00102  * used any more in the function. This operation is not needed for
00103  * most implementations of local continuation, but is required by a
00104  * few implementations.
00105  *
00106  * \hideinitializer
00107  */
00108 #define LC_END(lc)
00109
00110 /**
00111  * \var typedef lc_t;
00112  *
00113  * The local continuation type.
00114  *
00115  * \hideinitializer
00116  */
00117 #endif /* DOXYGEN */
00118
00119 #ifndef __LC_H__
00120 #define __LC_H__
00121
00122
00123 #ifdef LC_INCLUDE
00124 #include LC_INCLUDE
00125 #else
00126 #include "lc-switch.h"
00127 #endif /* LC_INCLUDE */
00128
00129 #endif /* __LC_H__ */
00130
00131 /** @} */
00132 /** @} */
```

7.7 pt-sem.h File Reference

Couting semaphores implemented on protothreads.

```
#include "pt.h"
```

Data Structures

- struct pt_sem

  Semaphore control structure.

Macros

- #define PT_SEM_INIT(s, c)

  Initialize a semaphore.
- #define PT_SEM_WAIT(pt, s)

  Wait for a semaphore.
- #define PT_SEM_SIGNAL(pt, s)

  Signal a semaphore.

7.7.1 Detailed Description

Couting semaphores implemented on protothreads.

Author

Adam Dunkels adam@sics.se

Definition in file pt-sem.h.

7.8 pt-sem.h

Go to the documentation of this file.

```
00001 /*
00002  * Copyright (c) 2004, Swedish Institute of Computer Science.
00003  * All rights reserved.
00004  *
00005  * Redistribution and use in source and binary forms, with or without
00006  * modification, are permitted provided that the following conditions
00007  * are met:
00008  * 1. Redistributions of source code must retain the above copyright
00009  *    notice, this list of conditions and the following disclaimer.
00010  * 2. Redistributions in binary form must reproduce the above copyright
00011  *    notice, this list of conditions and the following disclaimer in the
00012  *    documentation and/or other materials provided with the distribution.
00013  * 3. Neither the name of the Institute nor the names of its contributors
00014  *    may be used to endorse or promote products derived from this software
00015  *    without specific prior written permission.
00016  *
00017  * THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS ``AS IS'' AND
00018  * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00019  * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00020  * ARE DISCLAIMED.  IN NO EVENT SHALL THE INSTITUTE OR CONTRIBUTORS BE LIABLE
00021  * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
00022  * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
00023  * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
00024  * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
00025  * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
00026  * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
00027  * SUCH DAMAGE.
00028  *
00029  * This file is part of the protothreads library.
00030  *
00031  * Author: Adam Dunkels <adam@sics.se>
00032  *
00033  * $Id: pt-sem.h,v 1.2 2005/02/24 10:36:59 adam Exp $
00034  */
00035
00036 /**
00037  * \addtogroup pt
00038  * @{
00039  */
00040
00041 /**
00042  * \defgroup ptsem Protothread semaphores
00043  * @{
00044  *
00045  * This module implements counting semaphores on top of
00046  * protothreads. Semaphores are a synchronization primitive that
00047  * provide two operations: "wait" and "signal". The "wait" operation
00048  * checks the semaphore counter and blocks the thread if the counter
00049  * is zero. The "signal" operation increases the semaphore counter but
00050  * does not block. If another thread has blocked waiting for the
00051  * semaphore that is signalled, the blocked thread will become
00052  * runnable again.
00053  *
00054  * Semaphores can be used to implement other, more structured,
00055  * synchronization primitives such as monitors and message
00056  * queues/bounded buffers (see below).
00057  *
00058  * The following example shows how the producer-consumer problem, also
00059  * known as the bounded buffer problem, can be solved using
00060  * protothreads and semaphores. Notes on the program follow after the
00061  * example.
00062  *
00063  \code
00064 #include "pt-sem.h"
00065
00066 #define NUM_ITEMS 32
00067 #define BUFSIZE 8
```

```
00068
00069 static struct pt_sem mutex, full, empty;
00070
00071 PT_THREAD(producer(struct pt *pt))
00072 {
00073   static int produced;
00074
00075   PT_BEGIN(pt);
00076
00077   for(produced = 0; produced < NUM_ITEMS; ++produced) {
00078
00079     PT_SEM_WAIT(pt, &full);
00080
00081     PT_SEM_WAIT(pt, &mutex);
00082     add_to_buffer(produce_item());
00083     PT_SEM_SIGNAL(pt, &mutex);
00084
00085     PT_SEM_SIGNAL(pt, &empty);
00086   }
00087
00088   PT_END(pt);
00089 }
00090
00091 PT_THREAD(consumer(struct pt *pt))
00092 {
00093   static int consumed;
00094
00095   PT_BEGIN(pt);
00096
00097   for(consumed = 0; consumed < NUM_ITEMS; ++consumed) {
00098
00099     PT_SEM_WAIT(pt, &empty);
00100
00101     PT_SEM_WAIT(pt, &mutex);
00102     consume_item(get_from_buffer());
00103     PT_SEM_SIGNAL(pt, &mutex);
00104
00105     PT_SEM_SIGNAL(pt, &full);
00106   }
00107
00108   PT_END(pt);
00109 }
00110
00111 PT_THREAD(driver_thread(struct pt *pt))
00112 {
00113   static struct pt pt_producer, pt_consumer;
00114
00115   PT_BEGIN(pt);
00116
00117   PT_SEM_INIT(&empty, 0);
00118   PT_SEM_INIT(&full, BUFSIZE);
00119   PT_SEM_INIT(&mutex, 1);
00120
00121   PT_INIT(&pt_producer);
00122   PT_INIT(&pt_consumer);
00123
00124   PT_WAIT_THREAD(pt, producer(&pt_producer) &
00125                      consumer(&pt_consumer));
00126
00127   PT_END(pt);
00128 }
00129  \endcode
00130  *
00131  * The program uses three protothreads: one protothread that
00132  * implements the consumer, one thread that implements the producer,
00133  * and one protothread that drives the two other protothreads. The
00134  * program uses three semaphores: "full", "empty" and "mutex". The
00135  * "mutex" semaphore is used to provide mutual exclusion for the
00136  * buffer, the "empty" semaphore is used to block the consumer is the
00137  * buffer is empty, and the "full" semaphore is used to block the
00138  * producer is the buffer is full.
00139  *
00140  * The "driver_thread" holds two protothread state variables,
00141  * "pt_producer" and "pt_consumer". It is important to note that both
00142  * these variables are declared as <i>static</i>. If the static
00143  * keyword is not used, both variables are stored on the stack. Since
00144  * protothreads do not store the stack, these variables may be
00145  * overwritten during a protothread wait operation. Similarly, both
00146  * the "consumer" and "producer" protothreads declare their local
00147  * variables as static, to avoid them being stored on the stack.
00148  *
00149  *
00150  */
00151
```

```
00152 /**
00153  * \file
00154  * Couting semaphores implemented on protothreads
00155  * \author
00156  * Adam Dunkels <adam@sics.se>
00157  *
00158  */
00159
00160 #ifndef __PT_SEM_H__
00161 #define __PT_SEM_H__
00162
00163 #include "pt.h"
00164
00165 /**
00166  * Semaphore control structure.
00167  *
00168  * This structure represents a counting semaphore for use with protothreads.
00169  * It should be declared as a static variable and initialized with PT_SEM_INIT()
00170  * before use. The contents of this structure are internal to the semaphore
00171  * implementation and should not be accessed directly by the user.
00172  *
00173  * \sa PT_SEM_INIT(), PT_SEM_WAIT(), PT_SEM_SIGNAL()
00174  */
00175 struct pt_sem {
00176   unsigned int count;
00177 };
00178
00179 /**
00180  * Initialize a semaphore
00181  *
00182  * This macro initializes a semaphore with a value for the
00183  * counter. Internally, the semaphores use an "unsigned int" to
00184  * represent the counter, and therefore the "count" argument should be
00185  * within range of an unsigned int.
00186  *
00187  * \param s (struct pt_sem *) A pointer to the pt_sem struct
00188  * representing the semaphore
00189  *
00190  * \param c (unsigned int) The initial count of the semaphore.
00191  * \hideinitializer
00192  */
00193 #define PT_SEM_INIT(s, c) (s)->count = c
00194
00195 /**
00196  * Wait for a semaphore
00197  *
00198  * This macro carries out the "wait" operation on the semaphore. The
00199  * wait operation causes the protothread to block while the counter is
00200  * zero. When the counter reaches a value larger than zero, the
00201  * protothread will continue.
00202  *
00203  * \param pt (struct pt *) A pointer to the protothread (struct pt) in
00204  * which the operation is executed.
00205  *
00206  * \param s (struct pt_sem *) A pointer to the pt_sem struct
00207  * representing the semaphore
00208  *
00209  * \hideinitializer
00210  */
00211 #define PT_SEM_WAIT(pt, s)      \
00212   do {                                  \
00213     PT_WAIT_UNTIL(pt, (s)->count > 0);          \
00214     --(s)->count;                       \
00215   } while(0)
00216
00217 /**
00218  * Signal a semaphore
00219  *
00220  * This macro carries out the "signal" operation on the semaphore. The
00221  * signal operation increments the counter inside the semaphore, which
00222  * eventually will cause waiting protothreads to continue executing.
00223  *
00224  * \param pt (struct pt *) A pointer to the protothread (struct pt) in
00225  * which the operation is executed.
00226  *
00227  * \param s (struct pt_sem *) A pointer to the pt_sem struct
00228  * representing the semaphore
00229  *
00230  * \hideinitializer
00231  */
00232 #define PT_SEM_SIGNAL(pt, s) ++(s)->count
00233
00234 #endif /* __PT_SEM_H__ */
00235
```

```
00236 /** @} */
00237 /** @} */
00238
```

7.9 pt.h File Reference

Protothreads implementation.

```
#include "lc.h"
```

Data Structures

- struct pt

  Protothread control structure.

Macros

Protothread status codes

These values are returned by a protothread to indicate its current status.

- #define **PT_WAITING** 0

  The protothread is waiting for a condition.
- #define **PT_YIELDED** 1

  The protothread has yielded.
- #define **PT_EXITED** 2

  The protothread has exited via PT_EXIT().
- #define **PT_ENDED** 3

  The protothread has ended normally.

Initialization

- #define PT_INIT(pt)

  Initialize a protothread.

Declaration and definition

- #define PT_THREAD(name_args)

  Declaration of a protothread.
- #define PT_BEGIN(pt)

  Declare the start of a protothread inside the C function implementing the protothread.
- #define PT_END(pt)

  Declare the end of a protothread.

Blocked wait

- #define PT_WAIT_UNTIL(pt, condition)

  Block and wait until condition is true.
- #define PT_WAIT_WHILE(pt, cond)

  Block and wait while condition is true.

Hierarchical protothreads

- #define PT_WAIT_THREAD(pt, thread)

  Block and wait until a child protothread completes.
- #define PT_SPAWN(pt, child, thread)

  Spawn a child protothread and wait until it exits.

Exiting and restarting

- #define PT_RESTART(pt)
  Restart the protothread.
- #define PT_EXIT(pt)
  Exit the protothread.

Calling a protothread

- #define PT_SCHEDULE(f)
  Schedule a protothread.

Yielding from a protothread

- #define PT_YIELD(pt)
  Yield from the current protothread.
- #define PT_YIELD_UNTIL(pt, cond)
  Yield from the protothread until a condition occurs.

7.9.1 Detailed Description

Protothreads implementation.

Author

Adam Dunkels adam@sics.se

Definition in file pt.h.

7.10 pt.h

Go to the documentation of this file.

```
00001 /*
00002  * Copyright (c) 2004-2005, Swedish Institute of Computer Science.
00003  * All rights reserved.
00004  *
00005  * Redistribution and use in source and binary forms, with or without
00006  * modification, are permitted provided that the following conditions
00007  * are met:
00008  * 1. Redistributions of source code must retain the above copyright
00009  *    notice, this list of conditions and the following disclaimer.
00010  * 2. Redistributions in binary form must reproduce the above copyright
00011  *    notice, this list of conditions and the following disclaimer in the
00012  *    documentation and/or other materials provided with the distribution.
00013  * 3. Neither the name of the Institute nor the names of its contributors
00014  *    may be used to endorse or promote products derived from this software
00015  *    without specific prior written permission.
00016  *
00017  * THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS ``AS IS'' AND
00018  * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00019  * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00020  * ARE DISCLAIMED.  IN NO EVENT SHALL THE INSTITUTE OR CONTRIBUTORS BE LIABLE
00021  * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
00022  * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
00023  * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
00024  * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
00025  * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
00026  * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
00027  * SUCH DAMAGE.
00028  *
00029  * This file is part of the Contiki operating system.
00030  *
00031  * Author: Adam Dunkels <adam@sics.se>
00032  *
00033  * $Id: pt.h,v 1.7 2006/10/02 07:52:56 adam Exp $
00034  */
00035
00036 /**
00037  * \addtogroup pt
00038  * @{
00039  */
00040
00041 /**
00042  * \file
```

```
00043  * Protothreads implementation.
00044  * \author
00045  * Adam Dunkels <adam@sics.se>
00046  *
00047  */
00048
00049 #ifndef __PT_H__
00050 #define __PT_H__
00051
00052 #include "lc.h"
00053
00054 /**
00055  * Protothread control structure.
00056  *
00057  * This structure is used to store the state of a protothread. It should be
00058  * declared as a local variable or as a member of a larger structure. The
00059  * contents of this structure are internal to the protothread library and
00060  * should not be accessed directly by the user.
00061  *
00062  * \sa PT_INIT()
00063  */
00064 struct pt {
00065   lc_t lc;
00066 };
00067
00068 /**
00069  * \name Protothread status codes
00070  *
00071  * These values are returned by a protothread to indicate its current status.
00072  * @{
00073  */
00074
00075 /** The protothread is waiting for a condition. */
00076 #define PT_WAITING 0
00077
00078 /** The protothread has yielded. */
00079 #define PT_YIELDED 1
00080
00081 /** The protothread has exited via PT_EXIT(). */
00082 #define PT_EXITED  2
00083
00084 /** The protothread has ended normally. */
00085 #define PT_ENDED   3
00086
00087 /** @} */
00088
00089 /**
00090  * \name Initialization
00091  * @{
00092  */
00093
00094 /**
00095  * Initialize a protothread.
00096  *
00097  * Initializes a protothread. Initialization must be done prior to
00098  * starting to execute the protothread.
00099  *
00100  * \param pt A pointer to the protothread control structure.
00101  *
00102  * \sa PT_SPAWN()
00103  *
00104  * \hideinitializer
00105  */
00106 #define PT_INIT(pt)   LC_INIT((pt)->lc)
00107
00108 /** @} */
00109
00110 /**
00111  * \name Declaration and definition
00112  * @{
00113  */
00114
00115 /**
00116  * Declaration of a protothread.
00117  *
00118  * This macro is used to declare a protothread. All protothreads must
00119  * be declared with this macro.
00120  *
00121  * \param name_args The name and arguments of the C function
00122  * implementing the protothread.
00123  *
00124  * \hideinitializer
00125  */
00126 #define PT_THREAD(name_args) char name_args
```

```
00127
00128 /**
00129  * Declare the start of a protothread inside the C function
00130  * implementing the protothread.
00131  *
00132  * This macro is used to declare the starting point of a
00133  * protothread. It should be placed at the start of the function in
00134  * which the protothread runs. All C statements above the PT_BEGIN()
00135  * invokation will be executed each time the protothread is scheduled.
00136  *
00137  * \param pt A pointer to the protothread control structure.
00138  *
00139  * \hideinitializer
00140  */
00141 #define PT_BEGIN(pt) { char PT_YIELD_FLAG = 1; LC_RESUME((pt)->lc)
00142
00143 /**
00144  * Declare the end of a protothread.
00145  *
00146  * This macro is used for declaring that a protothread ends. It must
00147  * always be used together with a matching PT_BEGIN() macro.
00148  *
00149  * \param pt A pointer to the protothread control structure.
00150  *
00151  * \hideinitializer
00152  */
00153 #define PT_END(pt) LC_END((pt)->lc); PT_YIELD_FLAG = 0; \
00154                    PT_INIT(pt); return PT_ENDED; }
00155
00156 /** @} */
00157
00158 /**
00159  * \name Blocked wait
00160  * @{
00161  */
00162
00163 /**
00164  * Block and wait until condition is true.
00165  *
00166  * This macro blocks the protothread until the specified condition is
00167  * true.
00168  *
00169  * \param pt A pointer to the protothread control structure.
00170  * \param condition The condition.
00171  *
00172  * \hideinitializer
00173  */
00174 #define PT_WAIT_UNTIL(pt, condition)            \
00175   do {                                          \
00176     LC_SET((pt)->lc);                           \
00177     if(!(condition)) {                          \
00178       return PT_WAITING;                        \
00179     }                                           \
00180   } while(0)
00181
00182 /**
00183  * Block and wait while condition is true.
00184  *
00185  * This function blocks and waits while condition is true. See
00186  * PT_WAIT_UNTIL().
00187  *
00188  * \param pt A pointer to the protothread control structure.
00189  * \param cond The condition.
00190  *
00191  * \hideinitializer
00192  */
00193 #define PT_WAIT_WHILE(pt, cond)  PT_WAIT_UNTIL((pt), !(cond))
00194
00195 /** @} */
00196
00197 /**
00198  * \name Hierarchical protothreads
00199  * @{
00200  */
00201
00202 /**
00203  * Block and wait until a child protothread completes.
00204  *
00205  * This macro schedules a child protothread. The current protothread
00206  * will block until the child protothread completes.
00207  *
00208  * \note The child protothread must be manually initialized with the
00209  * PT_INIT() function before this function is used.
00210  *
```

```
00211  * \param pt A pointer to the protothread control structure.
00212  * \param thread The child protothread with arguments
00213  *
00214  * \sa PT_SPAWN()
00215  *
00216  * \hideinitializer
00217  */
00218 #define PT_WAIT_THREAD(pt, thread) PT_WAIT_WHILE((pt), PT_SCHEDULE(thread))
00219
00220 /**
00221  * Spawn a child protothread and wait until it exits.
00222  *
00223  * This macro spawns a child protothread and waits until it exits. The
00224  * macro can only be used within a protothread.
00225  *
00226  * \param pt A pointer to the protothread control structure.
00227  * \param child A pointer to the child protothread's control structure.
00228  * \param thread The child protothread with arguments
00229  *
00230  * \hideinitializer
00231  */
00232 #define PT_SPAWN(pt, child, thread)             \
00233   do {                                          \
00234     PT_INIT((child));                           \
00235     PT_WAIT_THREAD((pt), (thread));             \
00236   } while(0)
00237
00238 /** @} */
00239
00240 /**
00241  * \name Exiting and restarting
00242  * @{
00243  */
00244
00245 /**
00246  * Restart the protothread.
00247  *
00248  * This macro will block and cause the running protothread to restart
00249  * its execution at the place of the PT_BEGIN() call.
00250  *
00251  * \param pt A pointer to the protothread control structure.
00252  *
00253  * \hideinitializer
00254  */
00255 #define PT_RESTART(pt)                          \
00256   do {                                          \
00257     PT_INIT(pt);                                \
00258     return PT_WAITING;                  \
00259   } while(0)
00260
00261 /**
00262  * Exit the protothread.
00263  *
00264  * This macro causes the protothread to exit. If the protothread was
00265  * spawned by another protothread, the parent protothread will become
00266  * unblocked and can continue to run.
00267  *
00268  * \param pt A pointer to the protothread control structure.
00269  *
00270  * \hideinitializer
00271  */
00272 #define PT_EXIT(pt)                             \
00273   do {                                          \
00274     PT_INIT(pt);                                \
00275     return PT_EXITED;                   \
00276   } while(0)
00277
00278 /** @} */
00279
00280 /**
00281  * \name Calling a protothread
00282  * @{
00283  */
00284
00285 /**
00286  * Schedule a protothread.
00287  *
00288  * This function shedules a protothread. The return value of the
00289  * function is non-zero if the protothread is running or zero if the
00290  * protothread has exited.
00291  *
00292  * \param f The call to the C function implementing the protothread to
00293  * be scheduled
00294  *
```

```
00295  * \hideinitializer
00296  */
00297 #define PT_SCHEDULE(f) ((f) < PT_EXITED)
00298
00299 /** @} */
00300
00301 /**
00302  * \name Yielding from a protothread
00303  * @{
00304  */
00305
00306 /**
00307  * Yield from the current protothread.
00308  *
00309  * This function will yield the protothread, thereby allowing other
00310  * processing to take place in the system.
00311  *
00312  * \param pt A pointer to the protothread control structure.
00313  *
00314  * \hideinitializer
00315  */
00316 #define PT_YIELD(pt)                            \
00317   do {                                          \
00318     PT_YIELD_FLAG = 0;                          \
00319     LC_SET((pt)->lc);                           \
00320     if(PT_YIELD_FLAG == 0) {                    \
00321       return PT_YIELDED;                        \
00322     }                                           \
00323   } while(0)
00324
00325 /**
00326  * \brief      Yield from the protothread until a condition occurs.
00327  * \param pt   A pointer to the protothread control structure.
00328  * \param cond The condition.
00329  *
00330  *             This function will yield the protothread, until the
00331  *             specified condition evaluates to true.
00332  *
00333  *
00334  * \hideinitializer
00335  */
00336 #define PT_YIELD_UNTIL(pt, cond)                \
00337   do {                                          \
00338     PT_YIELD_FLAG = 0;                          \
00339     LC_SET((pt)->lc);                           \
00340     if((PT_YIELD_FLAG == 0) || !(cond)) {       \
00341       return PT_YIELDED;                        \
00342     }                                           \
00343   } while(0)
00344
00345 /** @} */
00346
00347 #endif /* __PT_H__ */
00348
00349 /** @} */
```

Index