Protothread Library (pt)

Garrett Berg, Cloudform Design, garrett@cloudformdesign.com

Introduction

Protothreads are extremely lightweight stackless threads designed for severely memory constrained systems such as small embedded systems or sensor network nodes. Protothreads can be used with or without an underlying operating system.

Protothreads 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.

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 four bytes per protothread
- Can be used with or without an OS
- Provides blocking wait without full multi-threading or stack-switching
- Freely available under a BSD-like open source license

Example applications:

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

The original protothreads library was released under an open source BSD-style license that allows for both noncommercial and commercial usage. The only requirement was that the following credit be given:

The protothreads library was written by Adam Dunkels <adam@sics.se> with support from Oliver Schmidt <ol.sc@web.de>. Adam Dunkels, 3 June 2006

More information and new versions of their library can be found at the protothreads homepage: http://www.sics.se/~adam/pt/

This is a rework of their protothread library to offer significantly more features at only minimal changes to memory. It is intended to be used with **threading.h** or **ui.h** (user interface).

Overview

There are several things to keep in mind when using protothreads:

- Every time you "yield" to the outside system, your function is being left. When it is called again, the pthread functions will return you to your yielded spot. However, **all non-static data will be lost.** Functions like "put_input" have been created to make this problem less significant, and allow you to use the same function for multiple threads.

See the example **pt_example_led1** to see protothreads in action. Also look at the other **led examples** to see more comples uses of threading and protothreads.

Function Calls

- **PT_INIT(pt) ::**
    - Call to reinitialize a protothread. You should hardly ever have to do this.
- **PT_BEGIN(pt) ::**
    - The start of a protothreaded function. Should be called immediately after variable initializations.
- **PT_END(pt) ::**
    - The end of a protothread. Nothing should be called after this.
- **PT_WAIT_UNTIL(pt, condition) ::**
    - Blocks and waits until the condition is true
    - Block == No code after this statement will run until the condition is true.
- **PT_WAIT_WHILE(pt, condition) ::**

    - Blocks and waits while the condition is true.

- **PT_WAIT_MS(pt, ms) ::**

    - Blocks until at least **ms** milliseconds has passed (approximately, depends on other functions not taking up much time)

- **PT_WAIT_THREAD(pt, thread) ::**

    - Block and wait until a child protothread completes. For instance: PT_WAIT_THREAD(pt, call_other_thread(pt2));

- **PT_SPAWN(pt, thread_function) ::**

    - Automatically creates a temporary thread and calls thread function with it.

    - Blocks until that thread is complete.

- **PT_EXIT(pt) ::**

    - signal to the outside to set thread innactive. Ends thread activity.

- **PT_YIELD(pt) ::**

    - The most common call. Yields for one cycle.

- **PT_YIELD_VALUE(pt, value) ::**

    - same as **pt->clear_output(); pt->put_output(value); PT_YIELD(pt);**

- **PT_KILLED(pt) ::**

    - Note: Does NOT kill the protothread. Use this just before **PT_END(pt)**. If an outside function kills the protothread, the code will jump here first. Use this to do cleanup (such as free memory, set pins to input, etc.)

    - Note: once the thread has been killed, **all data is lost**. So all **pt_get_input(index)**/output will be gone.

Classes

pthread

- This is the standard protothread object. This is what keeps track of the position in calls and also of input and output data. There are only a couple of functions you need to worry about.

- **pthread.put_input(value) ::**

    - put an integer or string into the “input” buffer.

- **pthread.get_int_input(index) ::**
    - get an integer from index. Note that the first item put in is index = 0, second item is index = 1, etc.
- **pthread.get_str_input(index) ::**
    - returns a character array pointer at index
- **pthread.get_type_input(index) ::**
    - returns the type of the index. What you should be concerned about are:
        - type < vt_max == value is an integer
        - type == vt_str == value is a string (character array)
- **pthread.clear_input() ::**
    - clears all input values
- **pthread.del_input(index) ::**
    - deletes input value at index. Note that all values above this will have 1 less index
- **pthread.put_output(value), get_xxx_output(index), etc ::**
    - same as input
- **pthread.clear_data() ::**
    - clears ALL data (input, output, and temp).
- **pthread.print() ::**
    - useful when debugging your code. Gives a list of all data and their values.

Common Code:

Known Input Type:

```
uint8_t mythread(pthread *pt){
  uint16_t myint = pt->get_int_input(0);
  iferr_log_catch(); // catch errors.

  // YOUR CODE HERE

  return good_value;
error:
  return bad_value;
}
```

Unknown Input Types:

```
uint8_t mythread(pthread *pt){
  // demonstrates how to determine if the first index of the input is a string or an integer.

  uint8_t type = pt->get_type_input(0);  // get the type
  iferr_log_catch(); // make sure there were no errors

  if(type < vt_maxint) uint16_t myint = pt->get_int_input(0); // it is an int

  else if(type == vt_str) char *mystr = get_function(pt->get_str_input(0)); // it is a string

  else assert(0); // errorhandling — we don’t know what it was

  // YOUR CODE HERE

  return good_value;
error:
  // clean up
  return bad_value;
}
```
