


RODOS

RODOS activity tutorial

Version: 2.0
Date: Sept. 2019
Author: S. Montenegro



If your application would require many threads and you are working with small micro controller, which have very limited memory, then it could be difficult to allocate a stack for each thread. In this case and if your threads could work sequentially, then you can use instead of thread something like the activities defined and implemented in this tutorial.

Activities are time triggered like normal threads, but they will be called sequentially using a single common stack for all of them, which is located in the activity-scheduler which is a real RODOS thread. Activities are just like function calls. Each activity shall implement a `step()` method which will be called by the activity scheduler according to the (from user) requested timings. The `step()` method has to return before another activity may take control. All activities are managed in a static linked list (like threads). The constructor of the activities will register each of them in the activity linked list. So the user does not need to register them anywhere. The activity scheduler will go through this list, and compare the next execution time of each activity with the current time. If the time to be executed has arrived then the scheduler will call the `step` method of the corresponding activity.

This is very efficient and consumes very little memory, but it has following main drawbacks:

1. No Pre-emption. The `step` method of any activity has to return before the next activity may be executed. Therefore do not use things like `suspendCallerUntil()`. There are alternative constructs like `YIELD_UNTIL(time)` which may be used instead.
2. You may not use semaphores. Because all activities are executed by a single thread (the activity scheduler) a semaphore would block all activities and not just the calling one. The lack of semaphores is not a big problem, because if all activities are executed sequentially and we have no pre-emption, then we do not need to protect common objects (between activities). If really required we could implement an activity semaphore which is based on the activity `YIELD`.
3. You may not keep values/variables on the stack. Because the stack will be shared among all activities, after you return from `step`, other activity will overwrite your stack (that is normal by function calls). Use a Context object instead to store your data.

1. Activity superclass and schedule

activity.h is the superclass for your activities.

To be able to return from anywhere from your step() method and continue from the same place (like if it were a thread) we define C-macros YIELD and YIELD_UNTIL, which were inspired by the protothreads from:

<http://dunkels.com/adam/pt/>

"Protothreads are extremely lightweight stackless threads designed for severely memory constrained systems, such as small embedded systems or wireless sensor network nodes. Protothreads provide linear code execution for event-driven systems implemented in C/C++. Protothreads can be used with or without an underlying operating system to provide blocking event-handlers. Protothreads provide sequential flow of control without complex state machines or full multi-threading."

Both C-macros YIELD and YIELD_UNTIL store the line number where they are called and then return. The macro GOTO_LAST_YIELD will jump to the place where we were before the last return (YIELD)

activity.cpp contains the activity scheduler and executer. It goes through the activities list and calls the step() method of all due activities. If the time for an activity has arrived and it has defined a period (a cycle time) then the scheduler will compute and update the next time to execute the activity. This is done before the step method is called, because the step could set by its own another execution time. This is the case for example when using YIELD_UNTIL(time).

2. Examples

example-simple.cpp is the most simple implementation of two independent activities. In the initiator they define their timing properties. The step() method will be called respectively. step() has to run without interruption until its end. Only after step() concludes (or returns) the next activity may be called.

example-with-yield.cpp is a little more complex using constructs like YIELD and YIELD_UNTIL. The step() method will be interrupted before it reaches its end by calling YIELD or YIELD_UNTIL(time). Then the step() method will be called again at the corresponding time and continue from the place where it called YIELD. It works because YIELD hides a return. YIELD protocols the line where it was called and returns. By the next call we reach the statement GOTO_LAST_YIELD which jumps to the line stored by YIELD. It uses a C switch construct. Inside of step() we can even create an infinity loop (while(1)), if it contains YIELD or YIELD_UNTIL.