Schedulers and Tasks

TEP:	106
Group:	Core Working Group
Type:	Documentary
Status:	Final
TinyOS-Version:	2.x
Author:	Philip Levis and Cory Sharp

Note

This memo documents a part of TinyOS for the TinyOS Community, and requests discussion and suggestions for improvements. Distribution of this memo is unlimited. This memo is in full compliance with TEP 1.

Abstract

This memo documents the structure and implementation of tasks and task schedulers in TinyOS 2.x.

1. Introduction

TinyOS has two basic computational abstractions: asynchronous events and tasks. Early versions of TinyOS provided a single type of task -- parameter free -- and only a FIFO scheduling policy. While changing the latter was possible, the incorporation of tasks into the nesC language made it very difficult. Presenting task schedulers as a TinyOS component enables much easier customization, and allowing tasks to be presented as an interface enables extending the classes of tasks available. TinyOS 2.0 takes both approaches, and this memo documents the structure of how it does so as well as a simple mechanism that greatly increases system dependability.

2. Tasks and the Scheduler in TinyOS 1.x

Tasks in TinyOS are a form of deferred procedure call (DPC)[1], which enable a program to defer a computation or operation until a later time. TinyOS tasks run to completion and do not pre-empt one another. These two constraints mean that code called from tasks runs synchonously with respect to other tasks. Put another way, tasks are atomic with respect to other tasks[2].

In TinyOS 1.x, the nesC language supports tasks through two mechanisms, `task` declarations and `post` expressions:

```
task void computeTask() {
  // Code here
}
```

and:

```
result_t rval = post computeTask();
```

TinyOS 1.x provides a single kind of task, a parameter-free function, and a single scheduling policy, FIFO. `post` expressions can return FAIL, to denote that TinyOS was unable to post the task. Tasks can be posted multiple times. For example, if a task is posted twice in quick succession and the first succeeds while the second fails, then the task will be run once in the future; for this reason, even if a `post` fails, the task may run.

The TinyOS 1.x scheduler is implemented as a set of C functions in the file `sched.c`. Modifying the scheduler requires replacing or changing this file. Additionally, as tasks are supported solely through nesC `task` declarations and `post` expressions, which assume a parameter-free function, modifying the syntax or capabilities of tasks is not possible.

The task queue in TinyOS 1.x is implemented as a fixed size circular buffer of function pointers. Posting a task puts the task's function pointer in the next free element of the buffer; if there are no free elements, the `post` returns fail. This model has several issues:

1) Some components do not have a reasonable response to a failed post
2) As a given task can be posted multiple times, it can consume more than one element in the buffer
3) All tasks from all components share a single resource: one misbehaving component can cause other's posts to fail

Fundamentally, in order for a component A to repost a task after post failure, another component B must call a function on it (either a command or event). E.g., component A must schedule a timer, or expect a retry from its client. However, as many of these systems might depend on tasks as well (e.g., timers), it is possible that an overflowing task queue can cause the entire system to fail.

The combination of the above three issues mean that one misbehaving component can cause TinyOS to hang. Consider, for example, this scenario (a real and encountered problem on the Telos platform):

- A packet-based hardware radio, which issues an interrupt only when it finishes sending a packet
- A networking component that handles the interrupt to post a task to signal `SendMsg.sendDone`.
- A sensing component that posts a task when it handles an `ADC.dataReady` event
- An application component that sends a packet and then sets its ADC sampling rate too high

In this scenario, the sensing component will start handling events at a faster rate than it can process them. It will start posting tasks to handle the data it receives, until it fills the task queue. At some point later, the radio finishes sending a packet and signals its interrupt. The networking component, however, is unable to post its task that signals `SendMsg.sendDone()`, losing the event. The application component does not try to send another packet until it knows the one it is sending completes (so it can re-use the buffer). As the `sendDone()` event was lost, this does not occur, and the application stops sending network traffic.

The solution to this particular problem in TinyOS 1.x is to signal sendDone() in the radio send complete interrupt if the post fails: this violates the sync/async boundary, but the justification is that a *possible* rare race condition is better than *certain* failure. Another solution would be to use an interrupt source to periodically retry posting the task; while this does not break the sync/async boundary, until the post succeeds the system cannot send packets. The TinyOS 1.x model prevents it from doing any better.

3. Tasks in TinyOS 2.x

The semantics of tasks in TinyOS 2.x are different than those in 1.x. This change is based on experiences with the limitations and run time errors that the 1.x model introduces. **In TinyOS 2.x, a basic post**

will only fail if and only if the task has already been posted and has not started execution. A task can always run, but can only have one outstanding post at any time.

2.x achieves these semantics by allocating one byte of state per task (the assumption is that there will be fewer than 255 tasks in the system). While a very large number of tasks could make this overhead noticable, it is not significant in practice. If a component needs to post a task several times, then the end of the task logic can repost itself as need be.

For example, one can do this:

```
post processTask();
...
task void processTask() {
  // do work
  if (moreToProcess) {
    post processTask();
  }
}
```

These semantics prevent several problems, such as the inability to signal completion of split-phase events because the task queue is full, task queue overflow at initialization, and unfair task allocation by components that post a task many times.

TinyOS 2.x takes the position that the basic use case of tasks should remain simple and easy to use, but that it should be possible to introduce new kinds of tasks beyond the basic use case. TinyOS achieves this by keeping `post` and `task` for the basic case, and introducing task interfaces for additional ones.

Task interfaces allow users to extend the syntax and semantics of tasks. Generally, a task interface has an `async` command, post , and an event, `run`. The exact signature of these functions are up to the interface. For example, a task interface that allows a task to take an integer parameter could look like this:

```
interface TaskParameter {
  async error_t command postTask(uint16_t param);
  event void runTask(uint16_t param);
}
```

Using this task interface, a component could post a task with a `uint16_t` parameter. When the scheduler runs the task, it will signal the `runTask` event with the passed parameter, which contains the task's logic. Note, however, that this does not save any RAM: the scheduler must have RAM allocated for the parameter. Furthermore, as there can only be one copy of a task outstanding at any time, it is just as simple to store the variable in the component. E.g., rather than:

```
call TaskParameter.postTask(34);
...
event void TaskParameter.runTask(uint16_t param) {
  ...
}
```

one can:

```
uint16_t param;
...
  param = 34;
  post parameterTask();
...
task void parameterTask() {
  // use param
}
```

The principal difference between the simplest code for these two models is that if the component posts the task twice, it will use the older parameter in the TaskParameter example, while it will use the newer parameter in the basic task example. If a component wants to use the oldest parameter, then it can do this:

```
if (post myTask() == SUCCESS) {
  param = 34;
}
```

4. The Scheduler in TinyOS 2.x

In TinyOS 2.x, the scheduler is a TinyOS component. Every scheduler MUST support nesC tasks. It MAY also support any number of additional task interfaces. The scheduler component is resonsible for the policy of reconciling different task types (e.g., earliest deadline first tasks vs. priority tasks).

The basic task in TinyOS 2.x is parameterless and FIFO. Tasks continue to follow the nesC semantics of task and post, which are linguistic shortcuts for declaring an interface and wiring it to the scheduler component. Appendix A describes how these shortcuts can be configured. A scheduler provides a task interface as a parameterized interface. Every task that wires to the interface uses the unique() function to obtain a unique identifier, which the scheduler uses to dispatch tasks.

For example, the standard TinyOS scheduler has this signature:

```
module SchedulerBasicP {
  provides interface Scheduler;
  provides interface TaskBasic[uint8_t taskID];
  uses interface McuSleep;
}
```

A scheduler MUST provide a parameterized TaskBasic interface. If a call to TaskBasic.postTask() returns SUCCESS, the scheduler MUST run it eventually, so that starvation is not a concern. The scheduler MUST return SUCCESS to a TaskBasic.postTask() operation unless it is not the first call to TaskBasic.postTask() since that task's TaskBasic.runTask() event has been signaled. The McuSleep interface is used for microcontroller power management; its workings are explained in TEP 112[3].

A scheduler MUST provide the Scheduler interface. The Scheduler interface has commands for initialization and running tasks, and is used by TinyOS to execute tasks:

```
interface Scheduler {
  command void init();
  command bool runNextTask(bool sleep);
  command void taskLoop();
}
```

The init() command initializes the task queue and scheduler data structures. runNextTask() MUST run to completion whatever task the scheduler's policy decides is the next one: the return value indicates whether it ran a task. The bool parameter sleep indicates what the scheduler should do if there are no tasks to execute. If sleep is FALSE, then the command will return immediately with FALSE as a return value. If sleep is TRUE, then the command MUST NOT return until a task is executed, and SHOULD put the CPU to sleep until a new task arrives. Calls of runNextTask(FALSE) may return TRUE or FALSE; calls of runNextTask(TRUE) always return TRUE. The taskLoop() command tells the scheduler to enter an infinite task-running loop, putting the MCU into a low power state when the processor is idle: it never returns.

The scheduler is repsonsible for putting the processor to sleep predominantly for efficiency reasons. Including the sleep call within the scheduler improves the efficiency of the task loop, in terms of the assembly generated by the TinyOS toolchain.

This is the TaskBasic interface:

```
interface TaskBasic {
  async command error_t postTask();
  void event runTask();
}
```

When a component declares a task with the `task` keyword in nesC, it is implicitly declaring that it uses an instance of the TaskBasic interface: the task body is the runTask event. When a component uses the `post` keyword, it calls the postTask command. Each TaskBasic MUST be wired to the scheduler with a unique identifier as its parameter. The parameter MUST be obtained with the `unique` function in nesC, with a key of `"TinySchedulerC.TaskBasic"`. The nesC compiler automatically does this wiring when the `task` and `post` keywords are used.

The SchedulerBasicP implementation uses these identifiers as its queue entries. When TinyOS tells the scheduler to run a task, it pulls the next identifier off the queue and uses it to dispatch on the parameterized TaskBasic interface.

While the default TinyOS scheduler uses a FIFO policy, TinyOS components MUST NOT assume a FIFO policy. If two tasks must run in a particular temporal order, this order should be enforced by the earlier task posting the later task.

5. Replacing the Scheduler

The TinyOS scheduler is presented as a component named TinySchedulerC. The default TinyOS scheduler implementation is a module named SchedulerBasicP; the default scheduler component is a configuration that provides wire-through of SchedulerBasicP.

To replace the scheduler for a particular application, a developer SHOULD put a configuration named TinySchedulerC in the application directory: this will replace the default. The scheduler component provides a wire-through of the desired scheduler implementation. All scheduler implementations MUST provide a parameterize TaskBasic interface, as SchedulerBasicP does; this supports nesC post statements and task declarations and enables TinyOS core systems to operate properly. Generally, TinyOS core code needs to be able to run unchanged with new scheduler implementations. All scheduler implementations MUST provide the Scheduler interface.

For example, imagine a hypothetical scheduler that provides earliest deadline first tasks, which are provided through the TaskEdf interface:

```
interface TaskEdf {
  async command error_t postTask(uint16_t deadlineMs);
  event void runTask();
}
```

The scheduler implementation is named SchedulerEdfP, and provides both TaskBasic and TaskEdf interfaces:

```
module SchedulerEdfP {
  provides interface Scheduler;
  provides interface TaskBasic[uint8_t taskID];
  provides interface TaskEdf[uint8_t taskID];
}
```

An application that wants to use SchedulerEdfP instead of SchedulerBasicP includes a configuration named TinySchedulerC, which exports all of SchedulerEdfP's interfaces:

```
configuration TinySchedulerC {
  provides interface Scheduler;
  provides interface TaskBasic[uint8_t taskID];
  provides interface TaskEdf[uint8_t taskID];
```

```
}
implementation {
  components SchedulerEdfP;
  Scheduler = SchedulerEdf;
  TaskBasic = SchedulerEdfP;
  TaskEDF   = SchedulerEdfP;
}
```

For a module to have an earliest deadline first task, it uses the TaskEdf interface. Its configuration SHOULD wire it to TinySchedulerC. The key used for task unique identifiers MUST be "TinySchedulerC.TaskInterface", where *TaskInterface* is the name of the new task interface as presented by the scheduler. A common way to make sure a consistent string is used is to #define it. For example, TaskEdf.nc might include:

```
#define UQ_TASK_EDF "TinySchedulerC.TaskEdf"
```

In this example, the module SomethingP requires two EDF tasks:

```
configuration SomethingC {
  ...
}
implementation {
  components SomethingP, TinySchedulerC;
  SomethingP.SendTask -> TinySchedulerC.TaskEdf[unique(UQ_TASK_EDF)];
  SomethingP.SenseTask -> TinySchedulerC.TaskEdf[unique(UQ_TASK_EDF)];
}
```

The module SomethingP also has a basic task. The nesC compiler automatically transforms task keywords into BasicTask interfaces and wires them appropriately. Therefore, for basic tasks, a component author can either use the `task` and `post` keywords or use a TaskBasic interface. A component SHOULD use the keywords whenever possible, and it MUST NOT mix the two syntaxes for a given task. This is an example implementation of SomethingP that uses keywords for basic tasks:

```
module SomethingP {
  uses interface TaskEdf as SendTask
  uses interface TaskEdf as SenseTask
}
implementation {
  // The TaskBasic, written with keywords
  task void cleanupTask() { ... some logic ... }
  event void SendTask.runTask() { ... some logic ... }
  event void SenseTask.runTask() { ... some logic ... }

  void internal_function() {
    call SenseTask.postTask(20);
    call SendTask.postTask(100);
    post cleanupTask();
  }
}
```

The requirement that basic tasks not be subject to starvation requires that a scheduler supporting EDF tasks must ensure that basic tasks run eventually even if there is an unending stream of short deadline tasks to run. Quantifying "eventually" is difficult, but a 1% share of the MCU cycles (or invocations) is a reasonable approximation.

If the scheduler provides two instances of the same task interface, their unique keys are based on the name of the interface as the scheduler presents it (the "as" keyword). For example, imagine a scheduler

which provides two instances of TaskBasic: standard tasks and high-priority tasks. The scheduler usually selects a task for the high priority queue before the standard queue:

```
configuration TinySchedulerC {
  provides interface Scheduler;
  provides interface TaskBasic[uint8_t taskID];
  provides interface TaskBasic[uint8_t taskID] as TaskHighPriority;
}
```

It cannot always select a high priority task because that could starve basic tasks. A component that uses a high priority task would wire to TaskHighPriority with the key "TinySchedulerC.TaskHighPriority":

```
configuration SomethingElseC {}
implementation {
  components TinySchedulerC as Sched, SomethingElseP;
  SomethingElseP.RetransmitTask -
> Sched.TaskHighPriority[unique("TinySchedulerC.TaskHighPriority")];
}
```

6. Implementation

The following files in `tinyos-2.x/tos/system` contain the reference implementations of the scheduler:

- `SchedulerBasicP.nc` is the basic TinyOS scheduler, providing a parameterized TaskBasic interface.
- `TinySchedulerC.nc` is the default scheduler configuration that wires SchedulerBasicP to McuSleepC[3].

A prototype of a scheduler that supports EDF tasks can be obtained at the URL `http://csl.stanford.edu/~pal/tinysched.tgz`.

7. Author's Address

Philip Levis
358 Gates Hall
Stanford University
Stanford, CA 94305

phone - +1 650 725 9046
email - pal@cs.stanford.edu

Cory Sharp
410 Soda Hall
UC Berkeley
Berkeley, CA 94720

email - cssharp@eecs.berkeley.edu

8. Citations

Appendix A: Changing the Scheduler

The nesC compiler transforms the post and task keywords into nesC interfaces, wirings, and calls. By default, the statement:

```
module a {
  ...
}
implementation {
  task x() {
    ...
    post x();
  }

}
```

is effectively:

```
module a {
  ...
  provides interface TaskBasic as x;
}
implementation {
  event void x.runTask() {
    ...
    call x.postTask();
  }
}
```

Specifically, TinyOS maps a task with name T to a TaskBasic interface with name T. Posting T is a call to T.postTask(), and the task body is T.runTask(). Finally, T is automatically wired to TinySchedulerC with a unique() call.

While the fact that tasks are transformed into interfaces is built in to the nesC compiler, the exact names can be configured. Each platform's .platform file passes the -fnesc-scheduler option to the compiler. The standard option is:

```
-fnesc-scheduler=TinySchedulerC,TinySchedulerC.TaskBasic,TaskBasic,TaskBasic,runTask,postTask
```

There are 6 strings passed. They are:

1) The name of the scheduler component to wire the interface to (TinySchedulerC).
2) The unique string used when wiring to the scheduler component's parameterized interface (TinySchedulerC.TaskBasic).

[1] Erik Cota-Robles and James P. Held. "A Comparison of Windows Driver Model Latency Performance on Windows NT and Windows 98." In *Proceedings of the Third Symposium on Operating System Design and Implementation (OSDI).*

[2] David Gay, Philip Levis, Rob von Behren, Matt Welsh, Eric Brewer and David Culler. "The *nesC* Language: A Holistic Approach to Networked Embedded Systems." In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI).*

[3] TEP 112: Microcontroller Power Management.

3) The name of the interface on the scheduler component (TaskBasic).
4) The name of the interface type (TaskBasic).
5) The name of the event for running the task (runTask).
6) The name of the command for posting the task (postTask).

The nescc man page has further details.