The TOSThreads Thread Library

TEP:	134
Group:	Core Working Group
Type:	Documentary
Status:	Draft
TinyOS-Version:	2.x
Author:	Kevin Klues, Chieh-Jan Liang, Jeongyeup Paek, Razvan Musaloiu-E, Ramesh Govindan, Andreas Terzis, Philip Levis
Draft-Created:	13-May-2008
Draft-Version:	1.1
Draft-Modified:	2008-06-12
Draft-Discuss:	TinyOS Developer List <tinyos-devel at mail.millennium.berkeley.edu>

Abstract

This memo documents the TOSThreads thread library for TinyOS

1. Introduction

TOSThreads is an attempt to combine the ease of a threaded programming model with the efficiency of a fully event-based OS. Unlike earlier threads packages designed for TinyOS, TOSThreads supports fully-preemptive application level threads, does not require explicit continuation management, and neither violates TinyOS's concurrency model nor limits its concurrency. Additionally, adding support for TOSThreads requires minimal changes to the existing TinyOS code base, and bringing up a new platform to support the use of TOSThreads is a fairly easy process.

In TOSThreads, TinyOS runs inside a single high priority kernel thread, while all application logic is implemented using user-level threads, which execute whenever the TinyOS core becomes idle. This approach is a natural extension to the existing TinyOS concurrency model, adding support for long-running computations while preserving the timing-sensitive nature of TinyOS itself.

In this model, application threads access underlying TinyOS services using a kernel API of blocking system calls. The kernel API defines the set of TinyOS services provided to applications (e.g. time-sync [TEP133], collection [TEP119], and dissemination [TEP118]). Each system call in the API is comprised of a thin blocking wrapper built on top of one of these services. TOSThreads allows systems developers to re-define the kernel API by appropriately selecting (or implementing their own) custom set of blocking system call wrappers.

The following section describes the details of how TOSThreads interacts with TinyOS to provide each of the features described above.

2. Basic Architecture

The existing TinyOS concurrency model has two execution contexts: synchronous (tasks) and asynchronous (interrupts). These two contexts follow a strict priority scheme: asynchronous code can preempt synchronous code but not vice-versa. TOSThreads extends this concurrency model to provide a third execution context in the form of user-level application threads. Application threads run at the lowest priority, with the ability to only preempt one another. They are preemptable at any time by either synchronous code or asynchronous code, and synchronize amongst each other using standard synchronization primitives such as mutexes, semaphores, barriers, condition variables, and blocking reference counters (a custom mechanism we have developed ourselves). Take a look in `tos/lib/tosthreads/interfaces` to see the interfaces providing these primitives.

The basic TOSThreads architecture, consists of five key elements: the TinyOS task scheduler, a single kernel-level TinyOS thread, a thread scheduler, a set of user-level application threads, and a set of system call APIs and their corresponding implementations. Any number of application threads can concurrently exist (barring memory constraints), while a single kernel thread runs the TinyOS task scheduler. The thread scheduler manages concurrency between application threads, while a set of system calls provides them access to the TinyOS kernel.

In order to preserve the timing-sensitive operation of TinyOS, the kernel thread has higher priority than application threads. This *TinyOS thread* therefore always takes precedence over application threads as long as the TinyOS task queue is non-empty. Once the TinyOS task queue empties, control passes to the thread scheduler and application threads can run. The processer goes to sleep only when either all application threads have run to completion, or when all threads are waiting on synchronization primitives or blocked on I/O operations.

There are two ways in which posted events can cause the TinyOS thread to wake up. First, an application thread can issue a blocking system call into the TinyOS kernel. This call internally posts a task, implicitly waking up the TinyOS thread to process it. Second, an interrupt handler can post a task for deferred computation. Since interrupt handlers have higher priority than the TinyOS thread, the TinyOS thread will not wake up to process the task until after the interrupt handler has completed. Because interrupts can arrive at anytime, it is important to note that waking up the TinyOS thread may require a context switch with an interrupted application thread. Control eventually returns to the application thread after the TinyOS thread has emptied the task queue.

3. Modifications to the Standard TinyOS Code Base

Only two changes to the existing TinyOS code base are required to support TOSThreads: a modification to the boot sequence and the addition of a post-amble for every interrupt handler. Changes to the boot sequence only need to be made once and are independent of any platforms supported by TinyOS. Changes to the interrupt handlers MUST be handled on a microcontroller to microntroller basis. Additionally, a custom `chip_thread.h` file MUST be created for each microcontroller and place in its top level directory, e.g. `tos/chips/msp430`

3.1 Changes to the Boot Sequence

Instead of directly running the TinyOS boot sequence inside of main() as done previously, main() now calls out to a Boot.booted() event associated with booting up the thread scheduler.:

```
event void ThreadSchedulerBoot.booted() {
  //Thread sceduler specific init stuff
  ...
  ...

  //Encapsulate TinyOS inside a thread
  tos_thread = call ThreadInfo.get[TOSTHREAD_TOS_THREAD_ID]();
  tos_thread->id = TOSTHREAD_TOS_THREAD_ID;

  //Set the TinyOS thread as the current thread and activate it
  current_thread = tos_thread;
  current_thread->state = TOSTHREAD_STATE_ACTIVE;

  //Signal the boot sequence
  signal TinyOSBoot.booted();
}
```

This change is made in order to encapsulate TinyOS inside the single kernel-level thread and set it as the initial thread that starts running. Once this is done, the normal TinyOS boot sequence is ran by signaling the TinyOSBoot.booted() event.

At the bottom of the existing TinyOS boot sequence, we enter an infinite loop that continuously checks the TinyOS task scheduler to see if it has any tasks to run. If it does, it runs the next task in its queue. If it does not, it puts the microcontroller into its lowest possible power state and goes to sleep [TEP112].:

```
command void Scheduler.taskLoop() {
  for (;;) {
    uint8_t nextTask;

    atomic {
      while ((nextTask = popTask()) == NO_TASK)
        call McuSleep.sleep();
    }
    signal TaskBasic.runTask[nextTask]();
  }
}
```

By adding threads as a lowest priority execution context, we need to change these semantics slightly. Instead of going directly to sleep, we want to allow the thread scheduler to take control of the microcontroller and start scheduling any threads it has to run. Only once all application threads have run to completion, or when all threads are waiting on synchronization primitives or blocked on I/O operations is the microcontroller put to sleep.

We achieve such functionality by replacing the call to McuSleep.sleep() shown above, by a call that signals the thread scheduler to suspend the currently running thread (the TinyOS kernel thread).:

```
command void TaskScheduler.taskLoop() {
  for (;;) {
    uint8_t nextTask;

    atomic {
      while((nextTask = popTask()) == NO_TASK)
        call ThreadScheduler.suspendCurrentThread();
    }
    signal TaskBasic.runTask[nextTask]();
  }
}
```

Once the TinyOS thread has been suspended, the thread scheduler is free to begin scheduling application level threads however it sees fit.

3.2 Interrupt Handler Post-Ambles

With the changes described above, the only other *non-self-contained* TinyOS code necessary to support TOSThreads is the addition of a post-amble at the bottom of every interrupt handler. Since the number and type of interrupt handlers, as well as the semantics required for implementing them, differ from platform to platform, the way in which this post-amble is is added is highly dependent on the microcontroller in use. The post-amble itself, however, is completely platform-independent, and is provided via a `postAmble()` command included in the `PlatformInterrupt` interface.:

```
command void PlatformInterrupt.postAmble() {
  atomic {
  if(call ThreadScheduler.wakeupThread(TOSTHREAD_TOS_THREAD_ID) == SUCCESS)
    if(call ThreadScheduler.currentThreadId() != TOSTHREAD_TOS_THREAD_ID)
      call ThreadScheduler.interruptCurrentThread();
  }
}
```

As described in the following section, the call to `wakeupThread()` returns SUCCESS iff the TinyOS task queue has any tasks to process (i.e. the interrupt handler posted some tasks), otherwise it returns FAIL. Upon FAIL, we simply return from this function, and continue execution from the point at which the currently running thread was interrupted. Upon SUCCESS, we preempt the current thread, immediately scheduling the TinyOS thread for execution. The check to make sure that the current thread isn't already the TinyOS thread is simply an optimization used to bypass ”rescheduling” the already running thread.

This `postAmble()` command MUST be called at the bottom of EVERY interrupt handler provided by a platform. This interface is provided by the `PlatformInterruptC` component, and MUST be wired into every component which implements an interrupt handler. Calls to `postAmble()` MUST then be made just before any return points in the given interrupt handler.

> **Note**
>
> Attempts were made to try and simplify / automate the inclusion of the post amble through the use of MACROS and other means. It was determined in the end, however, that this was the simplest and most readable way to keep the implementation of the post-amble platform independent, while allowing it to be included on a platform by platform basis for differing interrrupt handler implementations.

As an example, consider the case of the interrupt handlers for the TIMERA0_VECTOR and ADC_VECTOR on the msp430 microcontroller:

```
TOSH_SIGNAL(TIMERA0_VECTOR) {
  //Body of interrupt handler
  ...
  ...

  call PlatformInterrupt.postAmble();
}

TOSH_SIGNAL(ADC_VECTOR) {
  //Body of interrupt handler
  ...
  ...

  call PlatformInterrupt.postAmble();
}
```

The component in which each of these handlers is defined MUST wire in the `PlatformInterrupt` interface provided by `PlatformInterruptC` and call `postAmble()` at the bottom of their interrupt handlers.

3.3 The `chip_thread.h` file

A `chip_thread.h` MUST be created in the chips directory for each microcontroller supporting the TOSThreads library. This file MUST contain definitions of the following:

(1) A structure containing space for saving any microcontroller specific registers needed to save state when doing a context switch.:

```
typedef struct thread_regs {
  ...
} thread_regs_t;
```

(2) A typedef of a `stack_ptr_t` type. For example, the msp430 microconroller has 16 bit memory addresses, so `stack_prt_t` is typedefed as follows.:

```
typedef uint16_t* stack_ptr_t;
```

(3) Definitions of the following MACROS for use by the TOSThreads thread scheduler.:

```
PREPARE_THREAD(thread, start_function)
SWITCH_CONTEXTS(current_thread, next_thread)
RESTORE_TCB(next_thread)
STACK_TOP(stack, size)
```

As explained in Section 4.2, state manipulated by these MACROS is carried around by a thread as part of its *Thread Control Block (TCB)*, defined as type 'thread_t'.

`PREPARE_THREAD()` takes two parameters: 'thread' and 'start_function'. The 'thread' parameter MUST be of type `thread_t`, and 'start_function' MUST be of type `void (*start_function)(void)`. The purpose of `PREPARE_THREAD()` is to get a thread ready for the first time it starts to execute. Primarily, it is used to set the top of the stack associated with 'thread' to its 'start_function'. When it comes time for the thread to begin executing, the address pointed to by 'start_function' will be popped off and it will start executing.

As an example, consider the definition of `PREPARE_THREAD()` for the msp430 microcontroller:

```
#define PREPARE_THREAD(t, thread_ptr)          \
  *((t)->stack_ptr) = (uint16_t)(&(thread_ptr));       \
  SAVE_STATUS(t)
```

In this case, the status register is also saved with its initial setup, but this may not be necessary for all microcontrollers.

`SWITCH_CONTEXTS()` takes two parameters: current_thread and next_thread. Both parameters MUST be of type 'thread_t'.The purpose of `SWITCH_CONTEXTS()` is to store the state of the thread associated with the 'current_thread', and swap it out with the state of the 'next_thread'. The amount and type of state saved, and how it is actually swapped out varies from microcontroller to microcontroller.

`RESTORE_TCB()` takes just one parameter: next_thread. This parameter MUST be of type 'thread_t'. `RESTORE_TCB()` is similar to `SWITCH_CONTEXTS()` except that no state is stored about the current thread before swapping in the state associated with 'next_thread'. This MACRO is primarily called at the time a thread is either killed or has run to completion.

`STACK_TOP()` takes two parameters: 'stack' and 'size'. The 'stack' parameter MUST be of type `stack_ptr_t` and 'size' MUST be an integer type (i.e. uint8_t, uint16_t, etc). As explained in Section 4.2, whenever a thread is created, it is allocated its own stack space with a given size. As a thread executes, local variables, register values, and the return address of procedure calls are pushed onto this stack. Depending on the microcontroller in use, the *top* of a thread's stack might exist at either the highest address (stack grows down) or lowest address (stack grows up) of the data structure allocated to the stack. The purpose of `STACK_TOP()` is to return a pointer of type `uint8_t*` to the location of the *top* of the stack. `STACK_TOP()` is only called once at the time a thread is first initialized.

There are only two choices for the definition of this MACRO, and both are shown below.:

```
//Stack grows down (i.e. need to return pointer to bottom of structure (hig
#define STACK_TOP(stack, size)    \
  (&(((uint8_t*)stack)[size - sizeof(stack_ptr_t)]))


//Stack grows up (i.e. need to return pointer to top of structure (lowest a
#define STACK_TOP(stack, size)    \
  (&(((uint8_t*)stack)[0]))
```

As an example, consider the msp430 and atmega128 microcontrollers. On both of these microcontrollers, a thread's stack grows down as it executes, so `STACK_TOP()` is defined using the first macro.

4. The TOSThreads Library Implementation

This section describes the implementation of TOSThreads, including the internals of the thread scheduler, the thread and system call data structures, and their corresponding interfaces.

4.1 The Thread Scheduler

The thread scheduler is the first component to take control of the microcontroller during the boot process. As mentioned previously, its job is to encapsulate TinyOS inside a thread and trigger the normal TinyOS boot sequence. Once TinyOS boots and processes all of its initial tasks, control returns to the thread scheduler which begins scheduling application threads. The scheduler keeps threads ready for processing on a ready queue, while threads blocked on I/O requests or waiting on a lock are kept on different queues.

The default TOSThreads scheduler implements a fully preemptive round-robin scheduling policy with a time slice of 5 msec. We chose this value to achieve low latency across multiple application-level computing tasks. While application threads currently run with the same priority, one can easily modify the scheduler to support other policies.

As explained in the following section, TOSThreads exposes a relatively standard API for creating and manipulating threads: `create(), destroy(), pause() and resume()`. These functions form part of the system call API, and can be invoked by any application program.

Internally, TOSThreads library components use the following `ThreadScheduler` interface to interact with a thread.:

```
interface ThreadScheduler {
  async command uint8_t currentThreadId();
  async command thread_t* currentThreadInfo();
  async command thread_t* threadInfo(thread_id_t id);

  command error_t initThread(thread_id_t id);
  command error_t startThread(thread_id_t id);
  command error_t stopThread(thread_id_t id);

  async command error_t interruptCurrentThread();

  async command error_t suspendCurrentThread();
```

```
    async command error_t wakeupThread(thread_id_t id);
  }
```

The thread scheduler itself does not exist in any particular execution context (i.e., it is not a thread and does not have its own stack). Instead, any TOSThreads library component that invokes one of the above commands executes in the context of the calling thread. Due to the sensitive nature of these commands, ONLY the interrupt handler post-ambles and blocking system call API wrappers invoke them directly.

The first three commands are used to get information associated with an instance of a thread. Calling `currentThreadId()` returns a unique identifier associated with the currently running thread. Calling `currentThreadInfo()` or `threadInfo()` on a particular thread returns a pointer to the complete *Thread Control Block (TCB)* associated with a thread. Details about the TCB structure returned by these comamnds are given in section 4.2.

The rest of the commands in this interface are used to manipulate the state of a thread, putting it into one of 4 distinct states and starting / stopping its execution as necessary. At any given time, a thread may exist in one of the following states (INACTIVE, ACTIVE, READY, SUSPENDED).

Threads are initialized into the INACTIVE state via a call to `initThread()`. This command MUST only be called once at the time a thread is first created. A call to `initThread()` MUST be followed by a call to `startThread()` at some point later in order to start the actual execution of the thread. Calls to `initThread()` always return SUCCESS;

Calls to `startThread()` return either SUCCESS or FAIL, depending on the state a thread is in when it is called. If a thread is in the INACTIVE state, calling this command puts a thread into the READY state and places it on a ready queue. Threads are scheduled for execution by puling threads off this ready queue in FCFS order. If a thread is in any state other than INACTIVE, FAIL is returned, and no other side effects occur.

Calls to `stopThread()` only return SUCCESS if called on a thread that is in the READY state (and thereby implicitly on the ready queue) and currently holds no mutexes. The `mutex_count` field in a thread's TCB is used to determine if any mutexes are currently held. If both of these conditions are met, a thread is removed from the READY queue and its state is set to INACTIVE. If either of these conditions are not met, calling `stopThread()` returns FAIL and no other side effects occur.

Calls to `interruptCurrentThread()` are made in order to preempt a currently running thread. Calling `interruptCurrentThread()` returns SUCCESS if the currently running thread is in the ACTIVE state (SHOULD always be true), otherwise it returns FAIL. Upon FAIL no side effects occur. Upon SUCCESS, the currently running thread is put into the READY state and placed on the thread scheduler's ready queue. Threads in the READY state are not blocked, and will be scheduled for execution again the next time their turn comes up. The `interruptCurrentThread()` function is currently only called in two places. At the bottom of the interrupt `postAmble()` (as shown before), and in the code implementing the round-robin preemption scheme.

Calls to `suspendCurrentThread()` return SUCCESS if the currently running thread is in the ACTIVE state (SHOULD always be true), otherwise they return FAIL. Upon SUCCESS, the currently running thread is put into the SUSPEND state and its execution is stopped. A thread in the SUSPEND state will not be scheduled for execution again until a call to `wakeupThread()` is made at some later point in time. Calls to `suspendCurrentThread()` SHOULD only be made from within the body of blocking system call API wrappers or synchronization primitives.

Calls to `wakeupThread()` take one parameter: 'thread_id'. `wakeupThread()` returns SUCCESS if the thread associated with 'thread_id' is successfully woken up and returns FAIL otherwise. For all threads other than the TinyOS thread, SUCCESS will only be returned if the thread being woken up is in the SUSPEND state. For the TinyOS thread, it must be both in the SUSPEND state and have tasks waiting on it task queue. Upon SUCCESS, a thread is put in the READY state and placed on the ready queue. Upon FAIL, no side effects occur.

Note

Most times calls to *suspendCurrentThread()*' are paired with placing the suspended thread on a queue. Calls to `wakeupThread()` are then paired with removing a thread from that queue. The queue used is matianed externally by the component issuing the suspend and wakeup. For example, every mutex variable has its own queue associated with it. Everytime a thread calls the `lock()` function associated with a mutex, it is placed onto the queue associated with that mutex if someone already holds the lock. When the owner of the lock eventually calls `unlock()` this queue is then checked and requesters are removed from the queue in FCFS order.

4.2 Threads

Section 4 discusses the API provided to an application that allows it to create and destroy theads, as well as invoke blocking system calls. This section details the internals of the thread implementation itself, focusing on the data structure used to actually represent threads.

Regardless of the API used to create a thread (either *statically* or *dynamically* as discussed in the following section), TOSThreads allocates a Thread Control Block (TCB) and stack memory for each thread at the time it is created. Each thread has a fixed stack size that does not grow over time. The code snippet below shows the structure of a TOSThreads TCB.:

```
struct thread {
  thread_id_t thread_id;
  init_block_t* init_block;
  struct thread* next_thread;

  //thread_state
  uint8_t state;
  uint8_t mutex_count;
  thread_regs_t regs;

  //start_function
  void (*start_ptr)(void*);
  void* start_arg_ptr;

  stack_ptr_t stack_ptr;
  syscall_t* syscall;
};
```

thread_id: This field stores a thread's unique identifier. It is used primarily by system call implementations and synchronization primitives to identify the thread that should be blocked or woken up.

init_block: Applications implemented using the TOSThreds library have the ability to be dynamically loaded onto a mote at runtime. It is beyond the scope of this TEP to go into the details of this process, but applications use this field whenever they are dynamically loaded onto a mote. Whenever the system dynamically loads a TOSThreads application, the threads it creates must all receive the state associated with its global variables. An initialization block structure stores these global variables and 'init_block' points to this structure.

next_thread: TOSThreads uses thread queues to keep track of threads waiting to run. These queues are implemented as linked lists of threads connected through their next_thread' pointers. By design, a single pointer suffices: threads are *always* added to a queue just before they are interrupted and are removed form a queue just before they wake up. This approach conserves memory.

thread_state This set of fields store information about the thread's current state. It contains a count of the number of mutexes the thread currently holds; a state variable indicating the state the thread is in (INACTIVE, READY, SUSPENDED, or ACTIVE); and a set of variables that store a processor's register state whenever a context switch occurs.

stack_pointer This field points to the top of a thread's stack. Whenever a context switch is about to occur, the thread scheduler calls a `switch_threads()` function, pushing the return address onto the current thread's stack. This function stores the current thread's register state, replaces the processor's stack pointer with that of a new thread, and finally restores the register state of the new thread. Once this function returns, the new thread resumes its execution from the point it was interrupted.

start_function This field points to a thread's start function along with a pointer to a single argument. The application developer must ensure that the structure the argument points to is not deallocated before the thread's start function executes. These semantics are similar to those that Unix `pthreads` define.

system_call_block This field contains a pointer to a structure used when making system calls into a TOSThreads kernel. This structure is readable by both a system call wrapper implementation and the TinyOS kernel thread. The section that follows explains how this structure is used.

4.3 Blocking System Calls

TOSThreads implements blocking system calls by wrapping existing TinyOS services inside blocking APIs. These wrappers are responsible for maintaining state across the non-blocking *split-phase* operations associated with the underlying TinyOS services. They also transfer control to the TinyOS thread whenever a user thread invokes a system call. All wrappers are written in nesC with an additional layer of C code layered on top of them. We refer to the TOSThreads *standard* C API as the API providing system calls to standard TinyOS services such as sending packets, sampling sensors, and writing to flash. Alternative API's (potentially also written in C) can be implemented as well (e.g. the Tenet API).

A user thread initiates a system call by calling a function in one of the blocking API wrappers. This function creates a local instance of a *system call block (SCB)* structure which contains: a unique 'syscall_id' associated with the system call; a pointer to the 'thread' invoking the call; a pointer to the function that TinyOS should call once it assumes control, and the set of parameters this function should receive. The SCB is used to exchange data with the TinyOS thread.:

```
struct syscall {
  syscall_id_t syscall_id;
  thread_t* thread;
  void (*syscall_ptr)(struct syscall*);
  void* params;
};
```

All variables associated with a system call (i.e., the pointer to the SCB and the parameters passed to the system call itself) can all be allocated on the local stack of the calling thread at the time of the system call. This is possible because once the calling thread invokes a system call, it will not return from the function which instantiates these variables until after the blocking system call completes. These variables remain on the local thread's stack throughout the duration of the system call and can therefore be accessed as necessary.

As discussed in Section 2, making a system call implicitly posts a TinyOS task, causing the TinyOS thread to immediately wake up and the calling thread to block. In this way, there can only be *one* outstanding system call at any given time. Thus, only a *single* TinyOS task is necessary to perform an applications' system calls. The body of this task simply invokes the function the 'system_call_block' points to.

The important interfaces and components that implement the functionality described in this section can be found in `tos/lib/tosthreads/system` and `tos/lib/tosthreads/interfaces`. The important ones to look at are the `SystemCall` interface, and the `SystemCallC` and `SystemCallP` components. Example system call wrappers themselves include `BlockingStdControlC`, `BlockingAMSenderC`, etc.

5. Programming Applications

Application written using TOSThreads can be programmed in either nesC or standard C. In nesC, threads can either be created *statically* or *dynamically* as a TOSThreads application executes. The primary difference between the two is that statically allocated threads have their TCB allocated for them at compile time while dynamic threads have them allocated at run time.

5.1 Static Threads

Static threads are created by wiring in an instance of a ThreadC component as shown below. As a parameter to ThreadC, we pass in the desired stack size for the thread.

```
configuration ExampleAppC {
}
implementation {
  components MainC, ExampleC;
  components new ThreadC(STACK_SIZE);

  MainC.Boot <- ExampleC;
  ExampleC.Thread -> ThreadC;
}
```

The ThreadC component provides a `Thread` interface for creating and manipulating static threads from within a nesC module.:

```
 interface Thread {
   command error_t start(void* arg);
   command error_t stop();
   command error_t pause();
   command error_t resume();
   command error_t sleep(uint32_t milli);
   event void run(void* arg);
}
```

Calling `start()` on a thread signals to the TOSThreads thread scheduler that that thread should begin executing. `start()` takes as an argument a pointer to a data structure to pass to a thread once it starts executing. Calls to `start()` return either SUCCESS or FAIL. Upon SUCCESS, the thread is scheduled for execution, and at some point later the `run()` event is signaled. The body of the run event implements the logic of the thread.

Calling `stop()` on a thread signals to the TOSThreads thread scheduler that that thread should stop executing. Once a thread is stopped it cannot be restarted. Calls to `stop()` return SUCESS if a thread was successfully stopped, and FAIL otherwise. `stop()` MUST NOT be called from within the thread being stopped; it MUST be called from either the TinyOS thread or another application thread.

Calling `pause()` on a thread signals to the TOSThreads thread scheduler that that thread should be paused. Pausing a thread is different than stopping it in that a *paused* thread can be restarted again later by calling `resume()` on it. Underneath, the `pause()` command is implemented by calling the thread scheduler's `suspendCurrentThread()` command. Calls to `pause()` return SUCCESS if a thread is paused, and FAIL otherwise. Essentially, `pause()` is used to perform an explicit suspension of the currently running thread. `pause()` MUST ONLY be called from within the thread itself that is being paused. Note that these are exactly the opposite semantics than those used for `stop()`.

Calling `resume()` wakes up a thread previously suspended via the `pause()` command. SUCCESS is returned if the thread was successfully resumed, FAIL otherwise.

Calling `sleep()` puts a thread to sleep for the interval specified in its single 'milli' parameter. `sleep()` MUST ONLY be called from within the thread itself that is being put to sleep. SUCCESS is returned if the thread was successfully put to sleep, FAIL otherwise.

5.2 Dynamic Threads

Dynamic threads are created by wiring in an instance of a DynamicThreadC component as shown below.:

```
configuration ExampleAppC {
}
implementation {
  components MainC, ExampleC;
  components DynamicThreadC;
```

```
    MainC.Boot <- ExampleC;
    BlinkC.DynamicThread -> DynamicThreadC;
  }
```

The nesC interface for creating and manipulating dynamic threads is.:

```
  interface DynamicThread {
    command error_t create(tosthread_t* t, void (*start_routine)(void*),
                           void* arg, uint16_t stack_size);
    command error_t destroy(tosthread_t* t);
    command error_t pause(tosthread_t* t);
    command error_t resume(tosthread_t* t);
  }
```

`create()` is used to create a new dynamic thread. It takes 4 parameters: 't', 'start_routine', 'arg', and 'stack_size'. 't' is pointer to a unique handler associated with the thread being created. 'start_routine' is the start function associated with the thread. 'arg' is a pointer to a structure passed to the start function once the thread has begun executing. 'stack_size' is the size of the stack to be dynamicaly allocated for the thread. Calls to `create()` return either SUCCESS, FAIL, or EALREADY. SUCCESS indiactes that the thread has been successfully created, FAIL means it could not be created, and EALREADY indicates that the handler assocaited with the 't' parameter is associated with an already running thread. Upon SUCCESS, a dynamic thread's TCB and stack memory are dynamically allocated, and the thread is scheduled for execution. Its 'start_routine' whill be called once it begins running. UPON FAIL or EALREADY, no side effects occur.

"destroy()" is similar to `stop()` from a static thread's `Thread` interface. It follows all the same semantics as this command, except that it deallocates the TCB and stack memory associated with a thread before returning.

`pause()` and `resume()` are identical to their counterparts in the `Thread` interface.

The API allowing TOSThreads applications to be written in standard C includes a set of functions that simply wrap the commands provided by the `DynamicThread` interface. The following section shows an example of how these functions are used.

5.3 The Full TOSThreads API

As mentioned previously, TinyOS services are presented to TOSThreads applications by wrapping their functionality inside blocking system calls. The number and type of standard services provided is constantly growing so it doesn't make sense to try and list them all out. Browse through `tos/lib/tosthreads/system/` and `tos/lib/tosthreads/lib` to see what services are currently available. Also, take a look at the various applications found in `apps/tosthreads` to see how these services are used. There are applications written in nesC using both static and dynamic threads, as well as applications written in standard C.

6. Author Addresses

Kevin Klues

284 Gates Hall
Stanford University
Stanford, CA 94305-9030
email - klueska@cs.stanford.edu

Chieh-Jan Liang
XXX
XXX
XXX
email - cliang4@cs.jhu.edu

Jeongyeup Paek
XXX
XXX
XXX
email - jpaek@enl.usc.edu

Razvan Musaloiu-E
XXX
XXX
XXX
email - razvanm@cs.jhu.edu

Ramesh Govindan
XXX
XXX
XXX
email - ramesh@enl.usc.edu

Andreas Terzis
XXX
XXX
XXX
email - terzis@cs.jhu.edu

Philip Levis
358 Gates Hall
Stanford University
Stanford, CA 94305-9030
email - pal@cs.stanford.edu

7. Citations

[TEP112] TEP 112: Microcontroller Power Management.

[TEP118] TEP 118: Dissemination.

[TEP119] TEP 119: Collection.

[TEP133] TEP 133: Packet Level Time Synchronization.