Nano-RK: an Energy-aware Resource-centric RTOS for Sensor Networks

Anand Eswaran[1], Anthony Rowe[1] and Raj Rajkumar[1,2]
Real-Time and Multimedia Systems Lab
[1]Electrical and Computer Engineering Department, {aeswaran,agr,raj}@ece.cmu.edu
[2]School of Computer Science
Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

Many sensor networking applications such as surveillance and environmental monitoring are time-sensitive in nature. To support such applications, we design and implement Nano-RK, a reservation-based real-time operating system (RTOS) with multi-hop networking support for use in wireless sensor networks. We support fixed-priority preemptive multitasking for guaranteeing that task deadlines are met, along with support for CPU and network bandwidth reservations. *Tasks can specify their resource demands and the operating system provides timely, guaranteed and controlled access to CPU cycles and network packets in resource-constrained embedded sensor environments. We also introduce the concept of virtual* energy reservations *that allows the OS to enforce energy budgets associated with a sensing task by controlling resource accesses. A light-weight wireless networking stack supports packet forwarding, routing and TDMA-based network scheduling. Nano-RK has been implemented on the Atmel ATMEGA128 processor with the Chipcon CC2420 802.15.4 transceiver chip. Our results show that a light-weight embedded resource kernel with rich functionality and timing support is practical and constitutes a simple and alternative paradigm for supporting distributed sensing tasks.*

1 Introduction

The rapid proliferation of sensor networks has placed increasing demands upon the system infrastructure for supporting scalable distributed sensor applications. As applications for sensors in areas as diverse as security surveillance, traffic monitoring, smart spaces and smart buildings continues to grow, infrastructural support for sensor network applications in the form of system software is becoming increasingly important. The push provided by the scaling of technology and the need to support increasingly complicated and diverse applications has resulted in the need for traditional multitasking operating system (OS) abstractions and programming paradigms. The case for small-footprint real-time OS support in sensor networks is strengthened by the fact that many sensor networking applications are time-sensitive in nature i.e. the data must be delivered from the source to the destination within a timing constraint. For example, in a surveillance application, data relayed by a task which is responsible for detecting intruders and subsequently alerting the gateway nodes of the system should be able to reach the gateway on a timely basis.

In this paper, we present Nano-RK, a small-footprint embedded real-time operating system with networking support. Nano-RK supports the classical operating system multitasking abstractions allowing sensor application developers to work in a familiar paradigm resulting in short learning curves, quicker application development times and improved productivity. We show that an efficient implementation of such a paradigm is practical. We associate tasks with priorities and support priority-based preemption i.e, a task can always be preempted by a higher-priority task that becomes eligible to run. For timing sensitive applications, we use priority-based preemptive scheduling to implement the rate-monotonic paradigm [18] of real-time scheduling so that a periodic sensor task set with timing deadlines can be scheduled such that their timing guarantees are honored. Since modern sensor networks use ad-hoc multi-hop wireless networking for packet relaying, we provide port-based socket abstractions that can be used by sensing tasks for sending and receiving data.

Since sensor nodes are resource-constrained and energy-constrained, we provide functionality whereby the operating system can enforce limits on the resource usage of individual applications and on the energy budget used by individual applications and the system as a whole. In particular, we implement CPU reservations and Network Bandwidth reservations wherein dedicated access of individual application to system resources is guaranteed by the OS. The OS also implements sensor reservations to enforce usage on the number of accesses to individual sensors. Since the energy used by each task is the total sum of energy consumed by the CPU/microcontroller, the radio interface and the individual sensors, a particular setting for each of these leads to an energy reservation. Since we use a static design-time approach for admission control, we provide tools for estimating the energy budget of each application and (hence) the system lifetime. The CPU, network and sensor reservation values of tasks can be iteratively modified by the system designer until the battery lifetime requirements of the node are satisfied.

1.1 Related Work

Infrastructural software support for sensor networks was introduced by Hill et al. in [13]. They proposed TinyOS, a low-footprint component-based operating system that supports modularity and concurrency using an event-driven approach. TinyOS 1.0 supports an event-driven model wherein interrupts can register events, which can then be acted upon by other non-blocking functions. We believe that there are several drawbacks to this approach. The TinyOS design paradigm is a significant departure from the traditional programming paradigm involving threads, making it less intuitive for application developers. In contrast, we support a traditional multitasking paradigm retaining task abstractions and multitasking. Unlike TinyOS, where tasks cannot be interrupted, we support priority-based preemption. Nano-RK provides timeliness guarantees for tasks with real-time requirements. We provide task management, task synchronization and high-level networking primitives for the developers use. While our footprint size and RAM requirements are larger than that of TinyOS, our requirements are consistent with current embedded microcontrollers. A sensor network microcontroller may typically have 32-64KB of ROM and 4-8 KB of RAM. Therefore, Nano-RK is optimized primarily for RAM and secondarily for ROM. SOS [11] is architecturally similar to TinyOS with the additional capability for loading dynamic runtime modules. In contrast to SOS, we propose a static, multitasking paradigm with timeliness and resource reservation support.

The Mantis OS [12] is the most closely related work to ours in the existing literature. In comparison to Mantis, we provide explicit support for periodic task scheduling that naturally captures the duty cycles of multiple sensor tasks. We support real-time tasksets that have deadlines associated with their data delivery. We use the mechanisms of CPU and network reservations to enforce limits on the resource usage of individual tasks. With respect to networking we provide a rich API set for socket-like abstractions, and a generic system support for network scheduling and routing. Nano-RK supports power management techniques and provides several power-aware APIs for system use.[1]

While low-footprint operating systems such as μC/OS, OSEK and Emeralds [16] support real-time scheduling, they do not have support for wireless networking. Our networking stack is significantly smaller in terms of footprint as compared to existing implementation of wireless protocols like Zigbee (around 25 KB ROM and 1.5 KB RAM) and Bluetooth (around 50KB ROM). We also provide high-level socket-type abstractions, and hooks for users to develop custom MAC protocols.

Our system infrastructure can be used to complement distributed sensor applications such as an energy-efficient surveillance system ([1, 2]). Our contributions are compatible with the literature on real-time networking / resource allocation protocols [6, 10], energy-efficient routing/ scheduling schemes [4, 14] , data aggregation schemes [17], energy efficient topology control [9] and localization schemes [5, 8]. Nano-RK can be used as a software platform for building higher-layer middleware abstractions like [3]. Our energy reservation mechanism can also be used to prevent the type of energy DoS attacks described in [7].

Finally, our work complements [21] in extending the Resource Kernel (RK) paradigm to energy-limited resource-constrained environments like sensor networks (and hence the name "Nano-RK").

1.2 Organization of The Paper

The rest of this paper is organized as follows. Section 2 describes the requirements and design goals of a real-time operating system for sensor networks. Section 3 describes how a typical sensor application can leverage the programming paradigm to create a distributed sensing application. In Section 4, we describe the Nano-RK architecture and application programming interface in detail. Section 5 describes our implementation of Nano-RK on the ATMega128 processor. Section 6 presents an early evaluation of Nano-RK features. Finally, Section 7 summarizes our contributions and discusses potential avenues for future work.

Device	**Period**	**Execution Time**
Radio	Sporadic	10 ms
Microphone	200 Hz	10 us
Light Sensor	166 Hz	10 ms
Smart Camera	1 Hz	300 ms
GPS	5 Hz	10 ms

Figure 1: An example sensor taskset.

2 Design Goals for a Sensor RTOS

We present the following design goals for an RTOS targeting wireless sensor networks.

- **Multitasking:** The OS should provide a simple and intuitive programming paradigm for easy use by application developers. It is desirable to retain the traditional multitasking paradigm familiar to both desktop and embedded system programmers. Application developers should be able to concentrate on application logic rather than low-level system issues such as scheduling and networking.

- **Networking Stack Support:** The OS should support multihop networking, routing and simple user-level networking abstractions similar to sockets. In particular, low-level networking details such as reliable packet transmission, multicasting, queue management etc. should be handled by the OS.

- **Support for Priority-based Preemption:** Node battery lifetime continues to be a major challenge in sensor networks. Hence, given that energy consumed by processing per bit is significantly less than the per-bit energy consumed by the radio interface, there is a trend toward increased local processing (such as embedded vision and sound processing). This typically results in increased task execution times. In such situations where task run-times are large, there is a need for priority-based preemption to give precedence to higher priority events.

  True preemptive multitasking becomes necessary in a system where multiple inputs to the system must be serviced at different rates within a required period. For instance, imagine a sensing platform consisting of a microphone, light sensor, radio interface, a GPS for position information or time synchronization and a smart camera system. Figure 1 shows typical periods and execution times for these devices. Manually scheduling such a task set can become daunting using timer interrupts.

  A non-preemptive scheme might handle the radio with an external interrupt, the light and microphone with two priority-based timers, and leave the GPS and camera processing for the main program loop. Even in this situation, the developer may encounter difficulties because the camera servicing time is longer than the period of the GPS. Given that many low-end microcontrollers have limited timer interrupts, it can become difficult to schedule such a task set. Developers may need to resort to manual time splicing of their functions, thus making future modifications difficult. With a preemptive priority-based system, each of these sensing functions would be supported by a prioritized periodic task.

[1] Power-aware APIs can also be used by applications, albeit with prudence.

- **Timeliness and Schedulability:** Most sensor applications such as surveillance tend to be time-sensitive in nature where packets must be relayed and forwarded on a timely basis. While routing and network link scheduling are important components in ensuring that packets meet their end-to-end delay bounds, timing support on each node in the network is also essential. In order to satisfy end-to-end deadlines, local tasks on each node have deadlines associated with the completion of their local data relaying and processing. Managing the deadlines of these tasks requires support of a *real-time* operating system.

- **Battery Lifetime Requirements:** Guaranteeing sensor node battery lifetimes of 3 to 5 years is a very desirable objective in many sensor networks. If limits on the usage of energy can be enforced, lifetime guarantee requirements of the system as a whole can likely be provided (under reasonable assumptions about operating conditions such as network connectivity). The OS can also ensure that the system energy is apportioned in a manner commensurate with task importance such that critical tasks are guaranteed their energy budget.

- **Enforcement of Resource Usage Limits:** Since sensor nodes are resource-constrained, precious CPU cycles, network buffers and bandwidth should be apportioned to application needs. OS support for guaranteed, timely and limited access to system resources is conducive to supporting application deadlines and balanced apportioning of system slack (residual unused resources). This mechanism can also be used to place some limits on the impact of faulty or malicious tasks on system operation.

- **Unified Sensor Interface Abstraction:** Providing a unified and simple abstraction for accessing sensor readings and actuating responses would greatly benefit the end-user. In particular, low-level details associated with sensor/actuator configurations should be abstracted away from the user. Sensors should be supported using device drivers that can return real-world units as well as raw ADC values.

- **Small Footprint:** The current trend of low-end embedded processors is towards larger ROM sizes (64 KB to 128 KB) and smaller RAM sizes (2 KB to 8 KB). The OS architecture should be compliant with this trend by optimizing for RAM with a higher priority than ROM and optimizing for runtime efficiency. This memory constraint also implies that when the choice exists, one prefers a static configuration to a dynamic decision that requires additional data storage and run-time manipulations.

3 Nano-RK Programming Model

In this section, we show an example application that uses multiple tasks running in Nano-RK to monitor sensors and relay information to a remote node. Each task operates at a different frequency and priority. The example network consists of a sender node and a receiver node. The sender node hosts three tasks responsible for sensing sound, light and temperature. The receiver node collects packets from the sensing node and sets an LED signifying which sensor was triggered. The microphone is sampled at 1 KHz and is filtered such that a packet is sent only if an increase in volume above a particular threshold is detected. If people enter a room and begin working, an initial packet is sent. The system adjusts to the new ambient sound level, thus suppressing future packets. The light sensor is monitored once per second and a packet is transmitted only if the value changes beyond a certain threshold averaged over the past 10 readings. This allows the sensor to detect sudden changes like when an overhead light is turned on, but the task ignores slowly shifting changes in intensity levels like the sun moving during the day. The temperature sensor is read every three seconds and a packet is transmitted whenever the temperature changes by more than 2 degrees from the previously transmitted packet. If the temperature exceeds a certain threshold indicating fire or some other safety concern, a high priority packet is sent each time the sensor is read. The tasks are written using the C programming language. Shown below is a sample task that communicates on a port. The prefix `nrk_` is used on Nano-RK-specific datatypes and system calls (only for the sake of illustration).

```
void Sound_Task()
{
  int i, status, sound,prev_sound;
  // Transmit Buffer Stored in App.
  char tx_buff[2];
  nrk_port_des my_port;

  // setup socket on port 0 to broadcast
  port_des = nrk_port(tx_buff, 2, 0);
  // set to broadcast to all nodes
  nrk_connect(port_des, 0xFFFF);
  while (1) {
      sound = read_sensor(MIC);
      printf("T1 Sound = %d\r",sound);
      tx_buff[0] = sound;
      if (sound_change(sound,prev_sound)) {
          // Send Data using the socket
          nrk_port_send(my_port);
          wait_until_sent();
          printf("T1 Sent Packet\r");
      }
      prev_sound = sound;
      nrk_suspend_task();
    }
}
```

The next code segment shows how a task is created, and how its period and reservation are configured. Since we prefer a static design-time approach, task parameters like period and reservation capacities are populated during initialization and image creation rather than at run-time using system APIs. However, API support is available for programmers who need flexible reconfigurability of taskset properties. (These dynamic configuration calls can be unlinked from the final executable image if desired.)

```
nrk_task_type Task1;

Task1.task = Sound_Task;
Task1.Ptos = (void *) &Stack1[STACKSIZE - 1];
Task1.TaskID = 1;
Task1.priority = 3;
Task1.Period = 10;
Task1.set_cpu_reserve = 5;
Task1.set_network_reserve = 3;
Task1.set_sensor_reserve = 3;
nrk_activate_task (Task1);
```

The next code sample shows a receiver task. For the sake of demonstration purposes, each type of sensor packet is collected by a different task listening on a different port. The task runs at a priority based on its CPU reservation parameters, which in turn will likely depend on the sensor frequency.

```
void Get_Sound_Task()
{
```

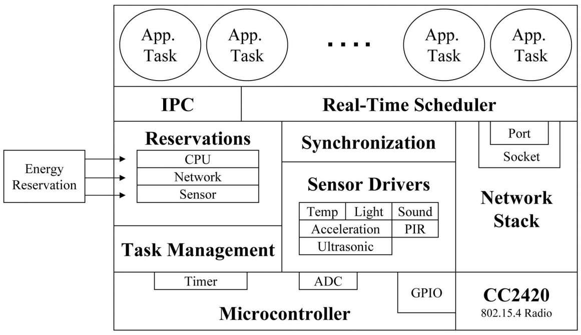

Figure 2: Nano-RK Architecture Diagram

```
  int i, status;
  char rx_buff[2];
  nrk_port_des my_port;

  // setup receive socket on port 0
  my_port = nrk_port(rx_buff, 2, 0, 0);
  // Listen on this socket for a packet
  nrk_listen(my_port);

  while (1) {
      CLR_LED();  // clear indicator
      // Non-block call to wait
      // for data on this socket
      nrk_port_read();
      nrk_wait_until_read();
      SET_LED();  // light up reception indicator
    }
}
```

The Appendix lists the complete set of Nano-RK APIs.

4 The Nano-RK Architecture

The particular requirements of systems support in sensor networking that were discussed earlier impose unique challenges with respect to designing an RTOS. In this section, we describe the architecture of Nano-RK, its constructs and capabilities. The overall system architecture of Nano-RK is shown in Figure 2.

4.1 Static Approach

Given the memory constraints of embedded sensor operating systems, Nano-RK uses a static design-time framework. This approach is consistent with sensor networking assumptions because as compared to traditional operating systems (where processes can be dynamically spawned), the OS and the applications are co-located in a single address space. In particular, admission control and real-time schedulability analysis tests are carried out offline as compared to taking a dynamic online approach. We would like to stress that a static approach does not mean that task properties and configuration parameters cannot be reconfigured during run-time. Rather, a static approach enforces the checks to ensure that the dynamic reconfiguration does not adversely affect application and system guarantees in a pre-deployment offline setting as compared to running dynamic admission control algorithms. Data- (or control-) dependent modifications to the task code such as changing task periods, resource usage limits, resource priorities and configuration of various parameters such as the network buffer sizes and stack sizes of each task can be changed to accommodate mode changes. With current energy and memory constraints, the run-time configurations will need to be verified offline at design-time[2]. This results in a light-weight operating system with a small footprint while retaining the rich set of functionality found in conventional RTOSs.

4.2 The Reservation Paradigm

The reservation paradigm, as implemented in a Resource Kernel [21], is a simple and practical paradigm for guaranteeing timeliness and enforcing temporal isolation in real-time operating systems. In resource kernels, applications specify timeliness and resource requirements, and the OS enforces guaranteed access to system resources and schedules tasks so that the application timeliness requirements are satisfied. While the resource kernel abstraction has so far been used in dynamic runtime settings, the resource reservation paradigm is desirable for static settings as well. A sensor application task can specify its requirement of CPU cycles, network bandwidth and network buffers over fixed periods which will be enforced by the Nano-RK kernel. Only tasks that have not depleted their reservation quota rates are eligible for scheduling. In deference to the stringent constraints of sensor nodes, exactly a single task is associated with a reservation. In contrast, the classical resource kernel concept allows for zero, one or more tasks to be bound to a reservation.

In summary, Nano-RK supports CPU reservations, sender/receiver network bandwidth reservations and sensor/actuator reservations. All of these reservations can be combined to enforce a virtual node-wide (and perhaps even system-wide) energy reservation.

4.3 Power Awareness Support

Since maximizing battery lifetime is often a primary objective in sensor networks, there is a need for aggressive power savings by operating at low duty cycles. Nano-RK enforces this in the form of *virtual energy reservations*. Note that the energy consumed by a task is the total sum of the CPU/microcontroller energy, radio interface energy and the energy associated with turning on sensors and actuators. The CPU and radio energy consumed by the task can be adjusted fairly accurately by changing the CPU and network reservation sizes. In order to bound the energy consumed by sensors, Nano-RK provides *sensor reservations*. Our unified sensor interface provides functionality wherein sensors are turned off (gated) by default and any access to a sensor is an atomic operation that consists of the sensor being turned on, its value being read and then being turned off again. This makes it possible for the operating system to set an upper limit on the number of accesses made to a sensor over a particular period. Thus it is is possible to map a resource tuple of $(CPU, Network, Sensor)$ reservations to a particular power level. Given periodic tasks, one can calculate the mean power used by all tasks over a hyper-period, giving a reasonably accurate estimate of the node lifetime. By modifying the values of the $(CPU, Network, Sensor)$ reservation-tuple, the mean energy consumed by each task can be varied. This can be used for either controlling the node lifetime or for varying the proportion of system energy allotted to each task (for example, certain mission-critical tasks can be allocated a high energy budget). We again note that energy reservations is implemented by controlling the $(CPU, Network, Sensor)$ reservation-tuple at a pre-deployment stage. This is consistent with the predominantly static approach that Nano-RK adopts.

[2] Future revisions of Nano-RK may relax this constraint.

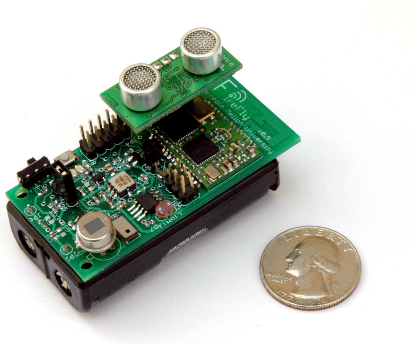

Figure 3: FireFly Sensor node w/ ultrasonic sensor.

4.4 Socket Abstractions and Routing

Since sensor nodes operate in a multihop networking environment, it may be necessary to route data periodically from sensor nodes to one or multiple gateways, or among nodes in a neighborhood for local coordination. Thus, nodes would benefit greatly from a sockets-like abstraction that masks the low-level details of MAC scheduling and routing from the application. In typical distributed sensor tasks, multiple nodes co-ordinate to achieve a common objective (e.g. sensing the presence of an intruder). Nano-RK's socket-like abstractions allow tasks to communicate with each other, with APIs for enabling a task to bind to a unique port. For memory efficiency, applications handle their own data buffers. The operating system is responsible for populating application buffers upon the reception of packets. The OS is also responsible for handling (reliable or unreliable) one-hop transmission of packet on behalf of the application. With corresponding OS control flags, it is possible to collapse multiple packets that have a common "next hop to destination" into a single packet, thus resulting in energy-efficient data forwarding.

Routing table data structures and destination look-up functions are managed by Nano-RK. We provide the basic infrastructural support over which ad-hoc routing protocols such as AODV [20] and DSR [19] can be implemented in the OS.

5 An Implementation of Nano-RK

In this section, we describe in detail our implementation of a static reservation-based preemptive operating system and the networking stack for the Atmel ATMega128 using the Chipcon CC2420 radio interface. Figure 4 shows a breakdown of Nano-RK's current resource requirements.

5.1 Hardware and Sensor Support

Nano-RK currently operates on an Atmel ATMEGA128-based sensor node shown in Figure 3 called **FireFly** built at CMU. The ATMEGA128L processor is an 8-bit microcontroller consisting of 128 KB of code space and 4 KB of data memory. It uses the Chipcon CC2420 802.15.4 wireless transceiver chip for communication. The platform has the following sensors: light, temperature, sound, passive infrared motion detection, and dual axis acceleration. In addition to sensors, the board has an ultrasonic transceiver add-on and 1 MB of external flash memory. All sensor peripherals can be gated by the main processor to conserve power. The processor has two internal 8-bit and four 16-bit timers, an 8-channel 10-bit ADC, a watchdog timer and six different sleep modes. The instruction set includes 135 instructions with multiplication instructions taking two clock cycles and the rest executing in a single cycle.

	Power	**Energy**
CPU		$(0.05mW * t_{idle}) + (24.0mW * t_{active})$
Idle	$0.05mW$	$0.05mW * t_{idle}$
Active	$24.0mW$	$24.0mW * t_{active}$
Network		$(.06mW * t_{idle}) + (1.8\mu J * N_{rx\_bytes}) + (1.6\mu J * N_{tx\_bytes})$
RX	$59.1mW$	$1.8\mu J$ per byte
TX	$52.1mW$	$1.6\mu J$ per byte
Idle	$.06mW$	$.06mW * t_{idle}$
Sensor		
Light, Temp	$.09mW$	$11.25nJ$ per reading
Microphone	$2.34mW$	$2.87\mu J$ per reading
PIR	$5.09mW$	$1\mu J$ per reading
Accel	$1.8mW$	$11.25nJ$ per reading
Ultrasonic TX	$60mW$	$15\mu J$ per ping
Ultrasonic RX	$30.8mW$	$30.8mW * t_{active}$

Table 1: Energy statistics for current hardware setup.

Most of the sensors use the Atmel's onboard analog to digital converter (ADC), with the exception of the accelerometer which requires software PWM (pulse width modulation) decoding. Nano-RK provides a set of sensor system calls that read raw sensor data and convert the data into meaningful units. It also ensures that the sensor reservations are updated when these calls are made. These functions are atomic, preventing deadlock due to hardware communication interruptions.

5.2 Task Management and Scheduling

Nano-RK task control block (TCB) structures are populated during initialization and system image creation[3]. They store the register context of all task (registers and stack), the task's priority, period of recurrence, $(CPU, network, Sensor)$ reservation sizes, port identifiers etc. Two linked lists of TCB pointers are used to order the set of active and suspended tasks respectively, based on period of recurrence. Tasks can block on certain events (such as being awakened at a certain point of time or the arrival of a network packet) and can be unsuspended and enqueued in the OS active list when the pending events occur. We suspend tasks that have pending events rather than using a polling-based implementation of Nano-RK system calls. This is done for energy-efficiency reasons because if there are no tasks eligible to run, the system can be powered down to sleep.

Our system uses priority-based preemptive scheduling and while we provide explicit support for periodic tasks, we also support aperiodic and sporadic tasks in our framework. The highest priority task that is eligible to run in the system is always scheduled by the operating system. A periodic task can suspend itself after the completion of its current instance using the `wait_until_next_period()` system call.

We implement priority ceiling protocol emulation (Highest Locker Priority protocol) to bound the blocking time encountered by a higher priority process due to the phenomenon of priority inversion (wherein a shared resource needed by the high-priority process is currently being used by a lower-priority pro-

[3] While we provide API support to modify fields in the TCB during run-time, we encourage a static configuration for a small-footprint ROM image.

Component	**Resource**
Context Swap Time	45 μs
Mutex Structure Overhead	5 bytes per resource
Stack Size Per Task	32 → 128 (typically 64) bytes
OS Struct Overhead	50 bytes per task
Network Structure Overhead	164 bytes
8 tasks, 8 mutexes, and 4 16-byte network buffers	< 2KB RAM, 10KB ROM

Figure 4: Breakdown of a Nano-RK configuration.

cess). In particular, each mutex is associated with a priority ceiling. When a mutex is acquired (using `lock_mutex()`), the priority of the task is elevated to the priority ceiling of the mutex. Once the mutex is released (using the `unlock_mutex()` system call), the priority of the task reverts to its original level. This results in bounded priority inversion which can be accounted for in the offline schedulability test. Thus, real-time synchronization is supported in Nano-RK.

Rather than provide explicit message box support, we provide system support for conventional semaphores that can be used by tasks to manipulate application buffers in a controlled manner for facilitating inter-process communication (using message boxes). This obviates the necessity for OS buffer space for storing message data and allows efficient *zero copying* [15] mechanisms to facilitate information sharing among tasks. Semaphores can also be used as a generalization of mutexes for guarding access to multiple resources.

5.3 Timing

Our implementation is based on the POSIX time structure `timeval`. We use a structure composed of two 32-bit numbers to represent `(seconds, nanoseconds)` fields in our time structures. For periodic tasks, we operate the timer in a one-shot mode wherein the next timing interrupt is triggered when either a task is scheduled to be awakened because of an event or because it becomes eligible for scheduling. In either case, the global `TOD` (time of day) counter field in the OS is updated. The `TOD` counter field is incremented periodically, and overflows will not occur for practically foreseeable intervals of time. Our system thus allows support for fine-granularity timing requirements of real-time applications while maintaining a (practically) non-overflowing notion of absolute time.

5.4 Reservation Support

CPU reservations are created statically by populating the corresponding task's TCB structure with reservation information[4]. In order to support CPU reservations, we associate each task with a `ticks_consumed` field, that accounts for the CPU cycles used by that application. When the `ticks_consumed` parameter exceeds the reservation size quota (reservation) specified by the system call, based on the policy of the reservation (*hard* or *soft*) one of the two following operations is carried out: if the reservation policy is *hard*, the application is immediately suspended and the reservation is replenished during the next period; if the reservation policy is *soft*, the application can consume the slack cycles unused by other tasks and when the system slack is depleted, it is suspended. For *soft* reservations, the CPU reservation quota is replenished during the next period of that task independent of whether slack resources were consumed or not. Since we take an offline approach, Nano-RK has full knowledge of system slack cycles. We allow an application to query the status and usage of its reservation using the `cpu_resv_query()` so that it can potentially adapt its behavior accordingly. Highly energy-sensitive deployments of Nano-RK environments will likely discourage or even disallow the use of soft reservations, due to the excess energy that can be consumed by prolonged execution or communications.

Network reservations are implemented statically by configuring the task TCB structures with parameters for setting a quota on the network bandwidth that is usable by that particular task. We provide two types of network reservations: network sender reservations that set a limit on bytes or packets that can be transmitted in a given period and network receiver reservations that enforce limits on the amount of received data or active radio receiver time in a given period. Each time a packet is received, task TCB parameters `bytes_consumed` and `packet_used` are incremented. When the network bandwidth quota equals the network reservation size, based on the network reservation policy of *hard* or *soft*, the application is either suspended or is eligible to access slack bandwidth in the system. The packet-count reservation sets a quota for the radio energy consumed while the number of bytes over a period enforces the limits on bandwidth consumption.

Sensor reservations are implemented in a similar manner by statically populating task TCB parameters to specify the number of times a sensor can be read in a given period. Each `sensor_read()` call increments a counter. If this counter exceeds the reservation quota, the process is suspended and the quota is replenished during the next period of the task.

As discussed earlier, *virtual energy reservations* can be implemented by judiciously choosing values of $(CPU, Network, Sensor)$ reservation values.

5.5 Network Protocol Stack

Nano-RK contains a lightweight network protocol stack that allows for port-based communication. Since the network stack is tightly integrated with the OS and execution/communication information is available, optimizations using global application knowledge such as automatic packet aggregation, network reservations, and buffer management are possible. An incoming data packet triggers an interrupt that handles the arrival of the packet. Packet transmissions are handled by a periodic network task responsible for servicing all outgoing packets of all tasks.

In traditional Unix-based systems, the OS is responsible for allocating and managing buffers for network communication. Statically allocating uniform buffers can be wasteful in sensor applications that transmit and receive only a few bytes per packet. In Nano-RK, the networking buffers are sized and allocated by the applications, but are managed by the network stack using a *zerocopy* buffer mechanism [15]. Upon successful reception of a packet by the OS interrupt routine, a data ready status flag is set that allows the application to directly manipulate the memory. The memory will not be touched again by the OS until the application explicitly resets the data ready status flag (using `port_receive_release()` or by calling another `port_read()` command). This allows the application developer to process the data in place to conserve memory and CPU cycles.

Packets transferred over the network contain a port and a destination address in their header. Upon reception of a packet des-

[4] We also support using `cpu_resv_create()` and `cpu_resv_modify()` APIs during run-time. However, we stress that these calls should be used with discretion to prevent malicious/faulty applications from dominating the use of system resources.

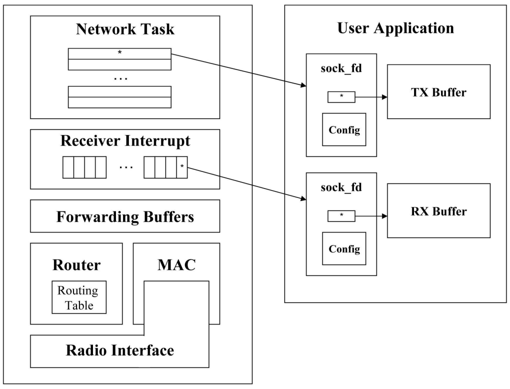

Figure 5: Nano-RK network stack showing placement of buffers in the application.

tined for the current node, the port is extracted from the header and used to identify which buffer the packet's data should be placed into. If the destination address is configured as a broadcast packet (0xFFFF), then all applications in the network that are listening on a particular port will receive the data. Unlike the traditional socket model, multiple applications on the same sensor node can listen on a single port. In this mode of operation, all listening applications must read the data before the OS will update the port with new packets. Though this would necessitate handling conflicts at the user level, it allows for multiple applications to share buffers, thus saving memory. Tasks that do not want to handle such conflicts can choose to use unique port numbers.

Information describing each socket is stored in a port_des data structure. This structure contains a pointer to the payload buffer, the size of the buffer, the port, destination address, message priority and hardware-specific parameters. The structure is instantiated through the use of the port() function call. The port_des data structure must then be passed to all transmit and receive functions. The hardware-specific parameters in this structure allow full configuration of the Chipcon CC2420 features such as encryption, radio transmit power, CRC error checking, clear channel assessment, and frequency selection. Internally, the OS contains an array of pointers to port data structures. This array is indexed by the port number. The server functionality of the network stack consists of three main commands: listen(), sock_read() and select(). The listen() function takes a port data descriptor as a parameter and is used to notify the OS that the application is expecting packets on a particular port. When a packet is received by the radio transceiver, an external interrupt routine is triggered which reads the packet from the radio's FIFO. Header information is collected before data, allowing the interrupt routine to take appropriate action without buffering the entire packet. If the destination address matches the current node's address, then the port is used to direct the memory into the application-defined buffer. If the port is unknown or found to be busy, then the packet is either dropped or is overwritten in the buffer (depending on a user-defined replacement policy). After the packet has been copied into the correct buffer, a status flag associated with that port is set, handing control of the buffer to the application. At this point, the scheduler is called to see if a task is waiting on an incoming packet. Once it resumes execution, the task can gain access to the buffer using the port_read() function. The port_read() function is non-blocking returning 1 only if a packet was correctly received. The OS will not update that buffer until the application has relinquished its control with either another port_read() call or an explicit port_read_release() call. Using the wait_until_read() function call, the application can block until a packet has been received. The select() function can be called with an 8-bit port index mask and a timeout. This is similar to the original BSD sockets implementation which supported 32 open file descriptors per process, and used one bit corresponding to each file descriptor in the mask. The select() function will hand over control of the task and not return until a matched packet is received or the timeout expires. If a selected packet is received, then select() will return its port descriptor.

The interface to the client functionality of the network stack consists of a connect() command and a port_write() function. The connect() command registers the outgoing port descriptor with the network stack. The port_write() command takes as an argument a port descriptor and updates a data ready field. Each time the network task is called, it will check all registered ports to see if there is a pending message for transmission. If more than one message resides in the transmit queue, the priority fields can determine the order in which they are to be sent. If the "next hop" of the messages are identical and have a size smaller than the maximum payload size, then the messages are aggregated and sent as a single packet. Since the application and OS share the same buffer, port_write_status() can be used to check if the packet was sent by the network task before updating the existing buffer with new data. The task can also turn over control until the packet is sent using the wait_until_sent() scheduler command.

5.5.1 Routing and MAC Support

Routing and MAC protocols in sensor network environments can be dramatically different (compared to traditional networks) and even be application-specific. Nano-RK tries to isolate this functionality allowing developers to implement their own protocols. We provide a basic template for a CSMA-CA MAC protocol with static routing tables. Though it is beyond the scope of this paper, we have also developed a TDMA-based protocol that utilizes global time synchronization to schedule packet transmissions. In the following section, we will discuss the infrastructure currently in place allowing for custom MAC and routing protocols.

When a packet is received by the OS, a routing function is called by the receive packet interrupt. This routing function can then access a routing table to determine if the current node is along the path to the destination. The routing function can also receive control packets such as route request packets used in ad-hoc routing protocols like AODV and DSR. These control packets have the ability to add, remove or update entries in the routing table. If the destination of the packet is found in the routing table, then the packet is added to a forwarding buffer inside the OS. Like any normal application transmit buffer, this forwarding buffer will be checked the next time the networking task is called.

The MAC layer control resides for the most part in the transmit portion of the network task. The only MAC layer control required in the receiver interrupt is the ability to automatically and immediately acknowledge a received packet to ensure reliable transmission. The MAC layer functionality such as col-

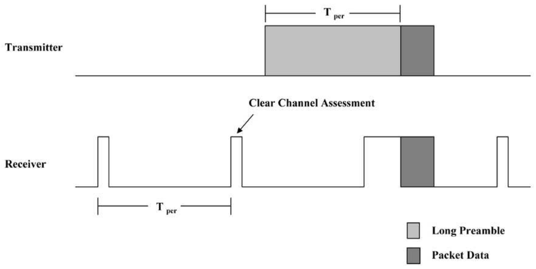

Figure 6: Low Power Listen transmit and receive waveforms.

lision sensing back-off times or TDMA-based access schemes are also incorporated into the networking task. Since this task is periodically called by the OS and is responsible for all data transmissions, it is the ideal place to implement custom MAC protocols. Along these same lines, Nano-RK includes a driver for low-level radio controls.

6 Experimental Evaluation

In this section, we provide simple examples that use the Nano-RK infrastructure. The first example is a power-efficient multi-hop networking MAC. The second example demonstrates how the energy reservation functionality provided by the OS can be used to guarantee pre-specified network (battery) lifetimes.

6.1 Low-Power Listening MAC

In this section, we show how the abstractions of periodic tasks naturally capture the duty cycle of distributed sensing applications. Using a method similar to Low-Power Listening (LPL) described in [22], we tested sending data over multiple hops. Figure 6 shows the basic operation of LPL. The receiver wakes up once every t_{per} for a short duration ($100 \mu s$) to assess the channel. If a message preamble is detected, the receiver continues to stay awake in order to receive the packet, otherwise it goes back to sleep. This implies that the sender needs to transmit a preamble for a period long enough for the receiver to hear it i.e, the transmitter synchronization preamble pulse width is t_{per}. There is an inherent trade-off between receiver power and transmitter power for operating at a particular application duty cycle and at a particular t_{per}. We would like the OS network task period (which is equal to t_{per}) to be set such that the total energy expended per packet is minimized. In the graph shown in Figure 8, we consider the effects of total energy consumed as a function of t_{per} for a 4-node chain where the application duty cycle requirements are 1 packet every 10 seconds. This packet contains the aggregated readings of temperature, light and sound sensors. For this configuration with our hardware setup, we experimentally found the optimal t_{per} to be around $100 ms$, as shown in the graph. We also plot the multihop end-to-end latency associated with the delivery of packets in the 4-hop chain network for the same period of the network task.

6.2 Effectiveness of Energy Reservations

In this subsection, we show how energy reservations and network bandwidth reservations can be utilized to maximize the network (battery) lifetime of sensor networks. Most sensor networks are topologically organized to form a forwarding tree that pushes sensor data to one or more gateway. The duty cycles of the individual nodes are chosen judiciously in order to guarantee the lifetime requirements of the sensor network. While implementing large distributed sensing systems, it is possible, even likely, that application code on one or more sensor nodes is configured in an energy-unfriendly manner by transmitting an excessive number of packets or by not aggressively turning its power components off. OS-enforced energy reservations are critical in ensuring that the network remains connected over its operational lifetime.

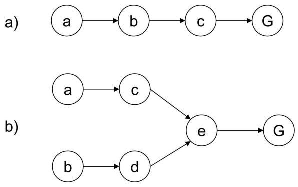

Figure 7: Sensor testbed setup. Topology (a) used for MAC experiment, (b) used for energy reservation experiment

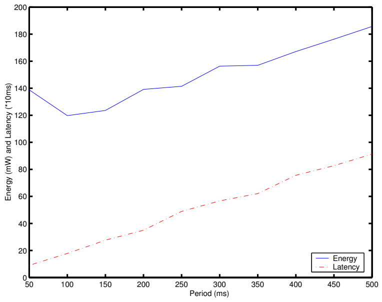

Figure 8: Graph of energy and multi-hop packet latency vs duty-cycle length. Packet latency was measured over four hops with a packet being sent once every 10 seconds.

Our experimental setup consisted of a sensor network that was arranged as shown in Figure 7 with a target lifetime of 2 years.

The duty cycles for achieving the lifetime requirements of 2 years are shown in Table 2. Since forwarding functionality is handled by the OS, if reservations were not used, the OS would simply forward packets further up the tree. We considered an

Node	**Reserve** [TX pkt/10 sec, RX pkt/10 sec]	**TX Rate** [pkt/10 sec]	**Total Packets Handled *without* Reserve**	**Total Packets Handled *with* Reserve**
a	[1,2]	1	720	720
b	*n/a*	300	216000	216000
c	[1,2]	1	1362	1350
d	[1,2]	1	195113	1358
e	[1,2]	1	196643	3211

Table 2: Enforcement from Energy Reservations

energy-efficient variant of naive packet forwarding where the network tasks of nodes higher up in the tree hierarchy collapse data belonging to their children in their forwarding buffers and tag their data forming a single packet. This aggregated packet is then forwarded up the tree. This is possible because sensor data payload sizes are typically small (2-4 bytes). In such a scenario, the duty cycles of all nodes[5] in the tree are typically equal. A sensor module operating on node d was configured incorrectly to send 300 packets every 10 seconds rather than 1 packet every 10 seconds.

Two experiments were conducted, one without energy reservations and another with energy reservation in place. The total number of packets handled[6] by each board was monitored for 2 hours for each experiment. The energy reservation for a node is enforced by setting a limit on the number of packets transmitted and on the time that the radio receiver is kept on.

After collecting the counter values for each node shown in Table 2, we used the sensor node's power characteristics shown in Table 1 to calculate the mean power for each node and from the node with maximum mean power, the network lifetime was calculated assuming a 2000 mAH battery. The network lifetime was found to be 34 days and 2.9 years when reservations were not used and used respectively. This demonstrates how reservations can find natural practical applications in designing robust sensor networks and contribute to significant improvements.

7 Conclusions and Future Work

In this paper, we described Nano-RK, a reservation-based energy-aware real-time operating system with wireless networking support for resource-constrained sensor network environments. We support a classically structured multitasking OS with API support for task management, synchronization, IPC and high-level networking abstractions, with these functions specifically tailored to the constrained sensor network environments. We enforce limits on CPU, bandwidth and sensor usage of individual tasks by using a reservation-based approach to enforce bounds on timeliness, QoS and node lifetime. We adopt a static design-time approach as compared to a dynamic run-time approach for creating an embedded sensor taskset. Our OS design uses several optimizations for memory- and energy-efficiency reasons while retaining a rich set of capabilities. Nano-RK will be made available for public use in the near future.

While we have shown that traditional RTOS abstractions are possible and natural for supporting sensor networking applications, there are several additions, enhancements, and optimizations that we are currently exploring. In particular, while the basic operating system abstractions are in place, the networking architecture of our sensor network is still work-in-progress. We do not currently support end-to-end deadline guarantees associated with packet delivery and are exploring scheduling and routing schemes based on Time Division Multiple Access (TDMA) scheduling using global time synchronization. We currently support only static routing and are exploring the use of optimized dynamic energy-efficient custom routing schemes. We also plan to make further optimizations to the OS for supporting energy-efficiency including support for low duty-cycle operation using TDMA techniques. We are currently designing OS-supported bootstrapping protocols for initializing and configuring the taskset parameters and for the distributed collection of connectivity information for large-scale sensor networks.

Acknowledgements: This work was supported in part by the Office of Naval Research, the National Science Foundation, Bosch Corporation and by Taiwan's Institute for Information Industry.

References

[1] Tian He, Sudha Krishnamurthy, John A. Stankovic, Tarek Abdelzaher, Liqian Luo, Radu Stoleru, Ting Yan, Lin Gu, Jonathan Hui, Bruce Krogh. *Efficient Surveillance System Using Wireless Sensor Networks* MobiSys'04, Boston, MA, June 2004.

[2] Juang P, Oki H, Wang Y, Martonosi M, Peh L-S, Rubenstein D *Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet* In Proceedings of ACM ASPLOS Conf, 2002

[3] Tarek Abdelzaher et al *EnviroTrack: Towards an Environmental Computing Paradigm for Distributed Sensor Networks* IEEE International Conference on Distributed Computing Systems, Tokyo, Japan, March 2004

[4] Hyung Seok Kim, Tarek Abdelzaher, Wook Hyun Kwon *Minimum-Energy Asynchronous Dissemination to Mobile Sinks in Wireless Sensor Networks* ACM SenSys, Los Angeles, CA, November, 2003.

[5] Tian He, Chengdu Huang, Brian Blum, John Stankovic, Tarek Abdelzaher Range-Free Localization Schemes for Large Scale Sensor Networks The 9th Annual International Conference on Mobile Computing and Networking (Mobicom), San Diego, CA, September 2003

[6] John A. Stankovic, Tarek Abdelzaher, Chenyang Lu, Lui Sha, Jennifer Hou *Real-Time Coomunication and Coordination in Embedded Sensor Networks* Proceedings of the IEEE, Vol. 91, No. 7, July 2003.

[7] A. D. Wood and J. Stankovic *Denial of Service in Sensor Networks* IEEE Computer, 35(10):54-62, 2002.

[8] Wei-Peng Chen, Jennifer C. Hou, and Lui Sha *Dynamic clustering for acoustic target tracking in wireless sensor networks* IEEE Trans. on Mobile Computing, Special issue in self-reconfiguring sensor networks, Vol. 3, Number 3, July-September 2004.

[9] Ning Li, Jennifer C. Hou and Lui Sha *Design and analysis of a MST-based distributed topology control algorithm for wireless ad-hoc networks* IEEE Trans. on Wireless Communications, Vol. 4, No. 3, pp. 1195–1207, May 2005.

[10] Simone Giannecchini, Marco Caccamo, Chi-Sheng Shih *Collaborative Resource Allocation in Wireless Sensor Networks* ECRTS 2004: 35-44

[11] Simon Han, Ramkumar Rengaswamy, Roy S Shea, Eddie Kohler, Mani B Srivastava *SOS : A Dynamic Operating System for Sensor Nodes* In Third International Conference on Mobile Systems, Applications and Services (Mobisys) June 2005

[12] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, R. Han *MANTIS: System Support For MultimodAl NeTworks of In-situ Sensors* 2nd ACM International Workshop on Wireless Sensor Networks and Applications (WSNA) 2003

[13] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, Kristofer Pister *System architecture directions for network sensors* ASPLOS 2000, Cambridge, November 2000.

[14] Wei Ye, John Heidemann, and Deborah Estrin. *An Energy-Efficient MAC protocol for Wireless Sensor Networks.* In Proceedings of the IEEE Infocom, pp. 1567-1576 June, 2002.

[15] Hsiao Keng, and J. Chu *Zero-copy TCP in Solaris* Proceedings of the USENIX, 1996.

[16] Zuberi, K. M., Pillai, P., and Sin, K. G. *EMERALDS: A small-memory real-time microkernel* In Proceedings of the 17th ACM Symposium on Operating System Principles ACM Press, pp. 277–291. June 1999

[17] Chalermek Intanagonwiwat, Ramesh Govindan and Deborah Estrin *Directed diffusion: A scalable and robust communication paradigm for sensor networks* In Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking (MobiCOM '00), August 2000

[18] C. L. Liu and J. W. Layland *Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment* Journal of the ACM, V20, N1, 1973, pp. 46-61.

[19] D. Johnson and D. Maltz. *Dynamic Source Routing in Ad Hoc Wireless Networks.* In T. Imielinsinki and H. Korth, Editors. Mobile Computing. Kluwer Academic Publishers, 1996

[20] C. Perkins, E. Belding-Royer, and S. Das. *Ad hoc On-Demand Distance Vector (AODV) Routing.* In IETF RFC 3561, July 2003.

[21] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. *Resource kernels: A resource-centric approach to real-time and multimedia systems.* In Proc. of the SPIE/ACM Conference on Multimedia Computing and Networking. January 1998

[22] J. Polastre, J. Hill, D. Culler *Versatile Low Power Media Access for Wireless Networks* In SenSys November 2004

[5] This is expressed as a per-task receive-time budget.

[6] This was achieved by maintaining separate counters for number of packets transmitted and number of packets received.

Appendix. The Nano-RK API

The system calls provided by the OS can be classified based on functionality. In its default configuration, Nano-RK supports a maximum of 16 tasks, 16 ports, 8 mutexes and 8 semaphores. For richer (or poorer) environments, constants in header files will need to be consistently re-defined.

Task Management APIs

activate_task()	Activates a periodic task; an aperiodic task has a period of -1
wait_for_next_period()	Suspend task until beginning of next period
wait_until()	Suspends Task until absolute time T
wait()	Suspends task for at least t
get_tid()	Returns TID of current task
terminate_task()	Permanently ends a periodic task
*get_priority()	Returns priority of specified task
*set_priority()	Changes priority of specified task

Network Communication APIs

port()	Set parameters and return port_des
connect()	Register port_des with network task
listen()	Listen for message on port_des
port_send()	Non-blocking send
port_send_status()	Check if message was sent
port_receive()	Non-blocking read
port_receive_release()	App finished with rx buffer
wait_until_received()	Block until packet received
wait_until_sent()	Block until packet sent
select()	Block until one wakeup event, or a timeout

Task Synchronization APIs

set_priority_ceiling()	Sets mutex priority ceiling
get_priority_ceiling()	Gets mutex priority ceiling
sem_init()	Creates a semaphore and sets it to v
sem_wait()	P(sem) wait for s>0 then s=s-1
sem_signal()	V(sem) s=s+1
lock_mutex()	Puts data into a message box
unlock_mutex()	Receives data from a message box

Sensor APIs

init_sensor()	Initialize a sensor
get_sensor_status()	Returns the status of a sensor
set_sensor_status()	Sets a sensor parameter
read_sensor()	Returns a sensor value
convert_sensor()	Translate sensor value to real world unit
wait_until_sensor()	Suspend task until a sensor completes
power_wake()	Power up a sensor
power_down()	Power down a sensor

Reservation APIs

cpu_rsv_query()	Queries CPU reserve properties
*cpu_rsv_modify()	Modify CPU reserve properties
netsndr_rsv_query()	Queries network sender reserve properties
*netsndr_rsv_modify()	Modifies network sender reserve properties
netrcvr_rsv_query()	Queries network receiver reserve properties
*netrcvr_rsv_modify()	Modifies network receiver reserve properties
snsr_rsv_query()	Queries sensor reserve properties
*snsr_rsv_modify()	Modify sensor reserve properties

Power Management APIs

query_energy()	Query residual battery energy
set_energy_mode()	Set energy savings mode (future)
get_energy_mode()	Get energy savings mode (future)
†tx_power_set()	Change radio transmitter power
powerdown()	Power the system down for t seconds

Radio APIs

radio_init()	Initialize the radio
radio_config()	Configure the radio
radio_rx_on()	Turn on the radio receiver
radio_rx_off()	Turn off the radio receiver
radio_tx_packet()	Transmit a packet
radio_read_byte()	Read a byte of a packet

Scheduling APIs

set_period()	Sets period of task
set_deadline()	Sets deadline of periodic task
get_time()	Returns global OS time
set_sched_policy()	Changes the scheduling policy used

UART APIs

kprint()	Safe print to serial port
set_UART()	Sets the UART parameters
putc()	Sends a single character over the UART
getc()	Read a character from the UART

Routing APIs

†add_route()	Delete route from routing table
del_route()	Add a route into the routing table

*This functionality may be optionally removed or unlinked from Nano-RK due to their run-time implications. Please use with caution.

†Transmit power will be set in conjunction with the routing scheme.