A Reference for the Nano-RK Real-Time Operating System

Chad M. Byers, Yang Yang, Yang Li, Kavitha Bhaskar

***Abstract*—In this paper, we give an overview of a real-time reservation-based operating system called Nano-RK used primarily in wireless sensor networks.**

I. Introduction

A. Real-Time Operating System (RTOS)

A real-time operating system is an operating system for real-time applications. A RTOS guarantees running applications with a consistent timing. Real-time operating systems do this by providing programmers with a high degree of control over task prioritization and allow checking to make sure that important deadlines are met. This can be especially important in measurement and automation systems where downtime is costly or a program delay could cause a safety hazard.

Real-time operating systems can be categorized as either hard real-time systems or soft real-time systems. In a hard-real time system, completion of a task beyond its deadline is considered as useless. In contrast to hard-real time system, soft-real time systems tolerate latency and use up the unused slack cycles of other processes. An application that runs on hard real-time systems is referred to as deterministic if its timing can be guaranteed within a certain margin of error. The amount of error in the timing of a task over subsequent iterations of a program or loop is referred to as jitter.

A real-time operating system is known to manage multiple tasks at the same time and allows prioritizing of process threads . Being a real-time system, an RTOS should also contain a sufficient number of interrupt levels. The main differences a real-time operating system has between that of general-purpose operating systems are (1) the need for a deterministic timing of tasks by the RTOS coupled with (2) a preemptive scheduling whereby time-based scheduling is used to ensure that the highest priority, ready task is running. General purpose operating systems, on the other hand, do not ensure deterministic timing of tasks and use general-purpose scheduling algorithms that overall try to achieve a fair system response.

B. Sensor Networks

The rapid popularity of sensor networks in various domains has placed "increasing demands upon the system infrastructure for supporting scalable distributed sensor applications" [1] The sensor networks are being used for various applications such as security, surveillance, traffic monitoring, smart spaces and smart buildings that are continuing more mainstream [1]. 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. Nano-RK uses no new method for multitasking but rather it follows the traditional methods of most operating systems to keep the learning curve low for the developers of Nano-RK. Nodes in a sensor network are known to be both constrained by both their computational resources/capabilities and the amount of energy they can consume in their lifetime as most are remotely deployed with infrequent maintenance or inspection.

II. Related Works

Real time operating systems are mainly used in embedded systems and robotics. Various real time operating systems, for embedded and non-embedded systems, have been developed such as Lynx, QNX, and VxWorks. A notable embedded real time operating system for robotics and automation systems is the Chimera OS while a more compact embedded OS named TinyOS, which does not support timing guarantees, has been developed using an event-driven programming model for cooperative multi-tasking. MantisOS is a non-real-time based operating system that does not include support for resource reservations, unlike Nano-RK. As we move out of the domain of embedded systems, several larger real-time operating systems have been developed such as uCos, Emerald, and OSEK.

III. Nano-RK

A. The Design of A Resource Kernel

Nano-RK, described in [3], has been designed to meet the requirements of multi-hop networking in wireless sensor network, and runs on various sensor nodes. It was developed by Carnegie Mellon University and was first introduced in a paper at RTSS 2005. Nano-RK overcomes the drawbacks of Tiny Os and MantisOS by providing (1) timeliness guarantees for tasks with real-time requirements and (2) high-level networking primitives as well as task management and synchronization mechanisms. It supports classical operating system multitasking abstraction to lower the learning curve for developers providing quicker development cycles. Nano-RK takes advantage of priority-based preemptive scheduling to help honor the real-time factor of being deterministic thus ensuring task timeliness and synchronization [5]. Due to the characteristic of limited battery power on the wireless node, Nano-RK provides CPU, network, and sensor efficiency through the use of virtual energy reservations, labeling this system as a *resource kernel*. These energy reservations can enforce energy and communication budgets to minimize negative impact on the node's operational lifetime from unintentional errors or malicious behavior by other nodes within the network. It supports packet forwarding, routing and other network scheduling protocols with the help of a light-weight wireless networking stack. Compared with other

current sensor operating systems, Nano-RK has optimized its microcontroller for current trends in hardware configuration of larger ROM (32-64KB) and smaller RAM (4-8KB)[3]. Thus, it provides rich functionality and timeliness scheduling with a small-footprint for its embedded resource kernel (RK).

B. Features of Nano-RK

Static Configuration - Nano-RK uses a static design-time approach for energy usage control. Dynamic task creation is disallowed by Nano-RK requiring application developers to set both task and reservation quotas/priorities in a static testbed design. This design allows the developers to create an an energy budget for each task in order to maintain application requirements as well as energy efficiency throughout the system's lifetime. Using a static configuration approach, all of the runtime configurations as well as the power requirements are predefined and verified by the designer before the system is deployed and executed in the real world. This approach also helps to guarantee the stability and small-footprint characteristics when compared with traditional RTOSs.

Watchdog Timer support - Watchdog is a software timer that triggers a system reset action if the system hangs on a crucial faults for an extended period of time. The watchdog mechanism can bring the system back from the nonresponsive state into normal operation by waiting until the timer goes off and subsequently rebooting the device. In Nano-RK, the watchdog timer is tied directly to the processor's reset signal REBOOT_ON_ERROR. By default, it is enabled when the system boots and reset each time the scheduler executes. If the system fails to respond within the predefined time period, the system will reboot and run the initialization instruction sequence to hopefully regain control.

Deep Sleep Mode - Another feature of Nano-RK is the deep sleep mode. For energy efficiency reasons, if there are no eligible tasks to run, the system can be powered down and given the option to enter deep sleep mode. When the system is in deep sleep mode, only the deep sleep timer can wake the system up with a predefined latency period. After waking up from the deep sleep mode, the next context swap time is set to guarantee the CPU wakes up in time. If a sensor node does not wish to perform deep sleep, it also is presented with the choice to go into a low energy consumption state while still managing its peripherals.

C. Basic Applications

The Nano-RK operating system for sensor networking platforms is designed for real-time sensor networking usage such as wireless packet transmission, light/temperature monitoring, sound monitoring and global positioning (GPS) data acquisition and synchronization. Practical applications include industrial control and automation, smart home monitoring, inventory and personnel tracking systems, embedded system education and hazardous environment monitoring [4]. An example application of Nano-RK in the real world is for a real-time voice streaming capability in wireless sensor networks. When coal miners are trapped in the mine tunnel, the rescuer can deploy microphones, cameras and air quality sensors from the surface to a mine tunnel, which enables the tracking of miner and its viability as an end-to-end rescue communication network for miners during disasters [4].

IV. Abstraction of Underlying Hardware

Since the Nano-RK operating system is a real-time operating system designed to be deployed on small embedded systems such as wireless sensor nodes, within the code downloaded directly from the Nano-RK website, there is support for four wireless sensor node platforms, namely, the Firefly sensor family (2.1 and 2.2), the Imec sensor family (standard and Gateway), the MicaZ sensor, and the T-mote sensor series. All platform-specific code related to timers, registers, pin declarations, flag values, LEDs, explicit communication port definitions, onboard RAM memory access, and any custom defined macros is contained in the path /src/platform/*. Platform-specific device driver information is contained in the path /src/drivers/platform/* and deals with drivers for analog-to-digital converters and additional per-sensor capabilities such as the Firefly 2.1 audio sensor capability and the Firefly 2.2 sensor's LCD add-on.

Onboard each sensor node is a microcontroller that handles the majority of interaction between the Nano-RK operating system and the physical device it resides on. By default, Nano-RK supports two families of microcontrollers,the Atmega family and the MSP430x family, however, any microcontroller or sensor node can be supported through defining your own custom hardware abstraction layers (HALs) accordingly. The hardware abstraction layers for Nano-RK are found in the path /src/kernel/hal/*. Nano-RK defines a clean, standard interface of stub functions and global variables in the /src/kernel/hal/include folder to abstract away hardware-specific functionality and information that must be handled in assembly code by the sensors microcontroller. The abstracted Nano-RK stub functions have the "nrk" prefix and are used throughout the main code of the operating sytem itself. Examples include: *nrk_target_start* which initializes the target hardware's (registers, timers, etc), *nrk_int_disable* and *nrk_int_enable* to disable/enable interrupts respectively, *nrk_stack_pointer_restore* and *nrk_stack_pointer_init* to handle the swapping and initializing of stacks in raw memory, and many more. The majority of the hardware-specific functionality provided by the hardware to these stub functions build off the pin declarations, flag definitions, and custom-defined macros mentioned previously in /src/drivers/platform/*. This type of abstraction and level of indirection allows the Nano-RK operating system to maintain device-independence for the majority of its main operation and provides an extendable and interchangeable design for various sensors seeking its support.

V. From a Developer Task to an Operating System TCB

As we mentioned before, Nano-RK can support sensor applications such as GPS, temperature, light, and audio. To control and schedule all these applications in the operating

system, Nano-RK provides the developer a data structure *nrk_task_type* to define their own specific tasks' attributes as shown in Figure 1.

```
typedef struct task_type {

    int8_t  task_ID;
    void *Ptos;                   // Top of stack pointer
    void *Pbos;                   // Bottom of stack pointer
    void (*task)();               // Function pointer to task entry point
    bool    FirstActivation;      // Whether task has been activated
    uint8_t prio;                 // Task priority, higher value signifies greater
    uint8_t Type;                 // Type of task
    uint8_t SchType;              // Type of scheduling
    nrk_time_t period;            // Period of task
    nrk_time_t cpu_reserve;       // CPU Reserve of task
    nrk_time_t offset;            // Starting offset phase

} nrk_task_type;
```

Fig. 1. Developer Task Structure

After a developer has initialized this structure, Nano-RK will copy each field of the developer task structure into the operating system task control block structure *NRK_TCB*, as well as initialize other attributes in the *NRK_TCB* (Figure 2).

```
typedef struct os_tcb {
    NRK_STK         *OSTaskStkPtr;        /* Pointer to current top of stack */
    NRK_STK         *OSTCBStkBottom;     /* Pointer to bottom of stack    */


    bool      elevated_prio_flag;
    bool      suspend_flag;
    bool      nw_flag;                 // allows user to wake up on event or nw;
    uint8_t   event_suspend;           // event 0 = no event ; 1-255 event type;
    int8_t    task_ID;                 // For quick reference later, -1 means not active
    uint8_t   task_state;              // Task status
    uint8_t   task_prio;               // Task priority (0 == highest, 63 == lowest)
    uint8_t   task_prio_ceil;          // Task priority (0 == highest, 63 == lowest)
    uint8_t   errno;                   // 0 no error 1-255 error code
    uint32_t  registered_signal_mask; // List of events that are registered
    uint32_t  active_signal_mask;     // List of events currently waiting on

    // Inside TCB, all timer values stored in tick multiples to save memory
    uint16_t  next_wakeup;
    uint16_t  next_period;
    uint16_t  cpu_remaining;
    uint16_t  period;
    uint16_t  cpu_reserve;
    uint16_t  num_periods;

} NRK_TCB;
```

Fig. 2. Task Control Block structure (as seen *within* the Nano-RK operating system)

Since Nano-RK does not permit the dynamic creation of tasks at run-time, it maintains a static number of operating system TCBs (Task Control Blocks) in an array called *nrk_task_TCB* where *NRK_MAX_TASKS* represents the number of tasks defined by users shown in Figure 3.

```
NRK_TCB  nrk_task_TCB[NRK_MAX_TASKS];   /* Table of TCBs */
```

Fig. 3. The static array of tasks currently registered with the Nano-RK operating system)

VI. Initializing Nano-RK on Target Hardware

To initialize Nano-RK on an embedded system's device, only a few function calls are required (shown in Figure 4). These function calls perform both hardware-specific and Nano-RK specific functionality before invoking the start of the operating system and are briefly discussed below.

```
#include <nrk.h>
#include <include.h>
#include <ulib.h>
#include <stdio.h>

int main ()
{
    nrk_setup_ports();
    printf( "Starting up ... \r\n" );
    nrk_init();
    nrk_time_set(0,0);
    nrk_create_taskset(); // User-defined
    nrk_start();
    return 0;
}
```

Fig. 4. The sequence of function calls necessary to prepare the target hardware device and Nano-RK)

nrk_setup_ports()
The nrk_setup_port() is the first function invoked upon system startup. This function initializes all the ports as well as the message receive queue buffer, which are defined based on the current platform in the CPU specific configuration file ulib.h. For example, in FireFly 2.1, the initialization process includes nrk_pin(). The nrk_pin() function defines high-level nrk pins mappings to hardware pins and ports. The ports setup is important because it provides the basic communicating ports to receive the messages for further operation.

nrk_init()
Before defining the user tasks, nrk_init() is called to initialize and clear the kernel stack and set up the idle task which will run when no other tasks are available in an infinite loop. It will also initialize all OS TCB structure parameters, resources, and signals as well as check to make sure the watchdog and voltage check have been successfully set up. The sequence of steps performed in this function are illustrated in Figure 5. Several important global variables used throughout the execution of the operating system which are initialized are: *nrk_cur_task_prio*,*nrk_cur_task_TCB*, *nrk_high_ready_TCB*, and *nrk_high_ready_prio*. Lastly, all structures related to the management of energy reservation and resources in _nrk_reserve[ ] are cleared and/or initialized to null as well as all system semaphores in _nrk_sem_list[ ]. Afterwards, entries in the task TCB structure nrk_task_TCB[] are cleared out and a double linked list of ready task queue structures in _nrk_readyQ[] are initialized. After all the operations above are finished, nrk_init() calls nrk_activate_task(&IdleTask) to initialize the idle task's stack. Since idle_task is the first activated task, the attributes in task TCB structure need to be reset as well.

nrk_create_taskset()
The *nrk_create_taskset()* function is where all user defined tasks will be initialized and activated. To define a specific task, a user should create a stack and an entry function for the task. Whenever the task is run by Nano-RK, the entry function will be called to do the specific work designed for

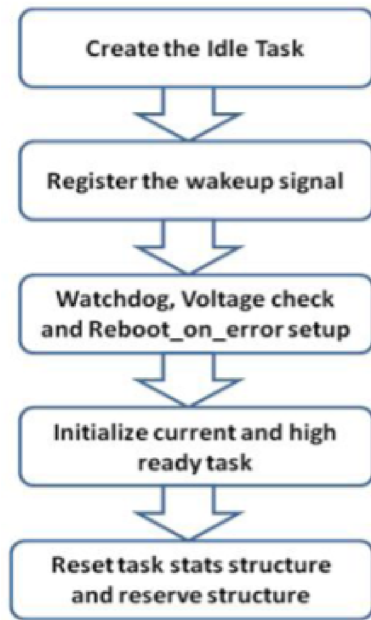


Fig. 5. Initialization Procedures

the task. Following this, the stack and the entry function will be combined with the task when initializing its attributes. Finally, the *nrk_activate_task()* and *nrk_TCB_init()* functions will be called to make the user-defined task runnable. These functions are defined in /src/kernel/source/nrk_task.c. The *nrk_activate_task()* function first initializes the task's stack. If the task is activated for the first time, the *nrk_TCB_init()* function will be called to create and initialize a TCB for the user-defined task; otherwise, just the stack pointer of the task's TCB will be updated. Before returning, this function will check whether the task needs to be run immediately (*next_wakeup* == 0). If so, it will change the state of the task's TCB to **READY** and add the task to the ready queue.

nrk_start()

The nrk_start() function is used to start the execution of ready tasks from the queue performed on a priority basis meaning that the highest priority ready task will ultimately be chosen to run at its conclusion. The nrk_task_TCB array is first scanned to check for the uniqueness of tasks using their task identifiers. Since the head node of the ready queue contains the highest priority task, the task ID of the highest priority task is easily obtained and set to the highest ready priority task. Initially this is also the highest current task running. The nrk_target_start() sets timers and interrupts on the target embedded system's device where the timer setup procedure is hardware-specific. The _nrk_os_timer_reset() sets up the asynchronous timer, clears interrupt flags, resets counters on interrupts and the prescaler. Timer 0 is an interrupt timer that calls the scheduler while Timer 1 is a high precision timer. _Nrk_os_timer_start() sets the divider for each timer and enables interrupts on timer 0. A canary value is inserted into the kernel stack to ensure there is no stack overflow. The task's stack pointer is then set to the last cell of the stack array in the nrk_stack_pointer_init() function and the highest ready task is now ready to run.

VII. Reserves

Nano-RK is known as the *"energy-aware resource-centric RTOS for sensor nodes"*. CPU cycles, sensors, actuators, network buffers, and bandwidth should be used only to the extent that is required by the application. The Nano-RK operating system emphasizes the importance of application deadlines and balanced distribution of system slack cycles. Slack cycles are a result of residual or unused resource time throughout the operating system's execution. Nano-RK supports guaranteed, timely and limited access to system resources.

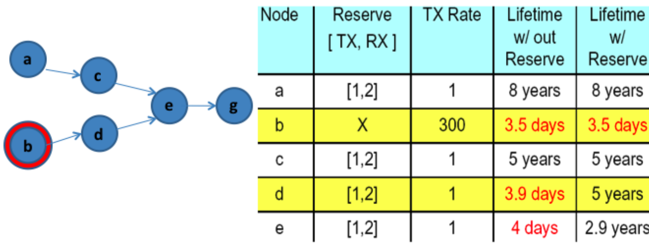
A sensor application task can specify its requirement over fixed periods which will be enforced by the Nano-RK kernel. Only tasks that have not depleted their reservation quota rates are eligible for scheduling. Using a CPU reservation, each task can be given a budget for how long it is allowed to execute in a given period of time. Network reservations, on the other hand, are based on a per task budget for a task's network usage in the transmission and receiving of packets. The number of system calls to a particular peripheral in a given period of time forms the baseline to reserve resources based on sensors and actuators. Total energy per node is calculated as the energy consumed by the CPU, network, and the sensors/actuators.

Violation of the reservation quota by each node can be handled in two ways based on whether the reservation follows a soft reservation policy or a hard reservation policy. A hard reservation policy suspends the application immediately on violation of the reservation followed by replenishing of the existing resources. On the other hand, a soft reservation policy consumes the residual or slack cycles of other tasks instead of suspending the application.

To *initialize* the nrk_reserve list, it is required to set all the reservation bits to inactive. *Creation* of the reserves involves going through the nrk_reserve list and setting the reservation to active followed by returning the reservation ID. Legitimate reservation IDs are used to set the nrk_reserve. The reservation length between replenishments of resources is calculated and an access count is maintained for the number of times a particular resource has been consumed within the calculated period of time. The sum of the current time and the time left for the next replenishment is stored in the set_time variable. There should be a constant *updating* of resource reservations by checking if the current access time has become greater than the set_time or not. If so, then the current access time is reset to 0, the resource is replenished, and a new set_time value is calculated. If the current access time is still within the set_time value, then the resource is allowed to be accessed and the current access is reduced by 1. This updating is done when a resource needs to be consumed and thus performed on a demand basis to make the resource reservation a more efficient process.

A. Importance of Resource Reservation

Using an example scenario shown in Figure 6, a deployed node may becomes faulty during its lifetime and broadcast messages into its environment without constraint. As a result, any listening nodes, such as node d and node e, will continuously listen and forward the unwanted packets from this faulty node to the specified gateway node (G). Without a reservation-based protocol, such as that employed by Nano-RK, the number of messages that can be transmitted can significantly shorten the lifetime of deployed nodes from 3-5 years down to 4-5 days. This clearly shows the need and importance of Nano-RK's design in the reservation of resources.



Node	Reserve [TX, RX]	TX Rate	Lifetime w/ out Reserve	Lifetime w/ Reserve
a	[1,2]	1	8 years	8 years
b	X	300	3.5 days	3.5 days
c	[1,2]	1	5 years	5 years
d	[1,2]	1	3.9 days	5 years
e	[1,2]	1	4 days	2.9 years

Fig. 6. Example scenario without the use of an energy reservation

VIII. Nano-RK Semaphores

Energy in embedded systems, such as wireless sensor networks, is a precious resource that must be conserved to maximize the lifetime of the node when deployed in the environment. Nano-RK implements a non-polling suspend protocol when a task is waiting on an event such as a signal or a resource's semaphore. This suspend protocol ensures that a task does not sit idling in the system consuming precious CPU cycles to poll the operating system for if the event it is waiting on has occurred. Semaphores in Nano-RK are similar to those in a traditional operating system in that they are a protected variable that implements restriction to the *number of accesses* that can occur to a shared system resource. In Nano-RK, shared resources can include storage devices (Flash and other secondary memory devices), sensor node resources (GPS, temperature sensors, light sensors, actuators, etc), and various other system resources depending on the target hardware the operating system is deployed in.

The variable structures and container used for managing semaphores are contained within the nrk_events.h (.cc) files. Each system resource initialized in Nano-RK has a semaphore dedicated to it within the global semaphore array *nrk_sem_list* found in nrk_defs.h defined to hold up to the maximum number of resources set. A count of the number of system resources is also maintained in the global varaible *_nrk_resource_cnt*. It is important to note that a semaphore is not automatically assigned to each resource but merely space within the array is established for **if** the developer chooses to associate a semaphore with it. A semaphore maintains three variables within it: (1) an 8-bit **count** integer that holds the total number of accesses allowed to this shared resource (2) an 8-bit **value** integer that contains the current number of accesses remaining and (3) an 8-bit **resource_ceiling** integer that represents the priority a task will acquire when it gains access to the semaphore. The resource ceiling variable is used to implement the Highest Locker Priority protocol to help mitigate the problem of priority inversion whereby a high priority task is waiting on a low priority task that has gained access to a shared resource that it needs to continue. The resource ceiling of a semaphore is statically-configured in the testbed environment for each resource and does not change after deployment into the field. Testing and simulation are crucial to both the setting of task priorities and resource priorities to maintain overall good system performance. The six functions associated with a semaphore are listed below along with a brief description of each:

nrk_get_resource_index – A scan of the *nrk_sem_list* array is made looking for the index of the semaphore that corresponds to the address passed in as a parameter. When an address match is made, the index is returned by the function.

nrk_sem_create – This function takes the count and ceiling priority as parameters that are to be associated with a newly created semaphore. It scans the *nrk_sem_list* array looking for an open, unused semaphore. When it finds an unused semaphore it sets its count, value, and resource ceiling variables accordingly to that of their associated parameters and increments the global resource counter. The address of the created semaphore is returned by the function.

nrk_sem_delete – After determining the semaphore's id in the *nrk_sem_list* array, this function resets all of the semaphores attributes and decrements the global resource counter.

nrk_sem_pend – After determining the semaphore's id in the nrk_sem_list array the function disables interrupts and determines whether the semaphore has any shared accesses left to the resource. If no remaining accesses exist, the task trying to access the task (*nrk_curr_task_TCB*) has its event suspend variable set to signify it is suspending on a resource. The task's active signal mask that usually stores individual bits set for signals is used to instead hold the index of the resource it is suspending on. Interrupts are then re-enabled and the task will suspend until the resource becomes available through a *nrk_sem_post*. If there are remaining accesses, the semaphore current access count will be decremented and the task that gained access to the resource will have its elevated priority flag set and its elevated priority ceiling will be set to the ceiling of the semaphore.

nrk_sem_post – Using the determined semaphore's id, this function will increment the number of accesses allowed to the semaphore (count), clear the current task's elevated priority flag, and go through the list of TCB structures to find those tasks that are suspended on this resource's id by checking their event suspend flag to see if it is set as being a resource suspension and whether their active signal mask contains this semaphore's id. For all task's that meet these criteria, the function will clear their event suspend flag, clear their active signal mask, and change their state to only be

suspended so that the next highest priority task scheduled will gain access to this resource / semaphore.

nrk_sem_query – This function returns how many current accesses to the shared resource's semaphore (passed in as a parameter) exist and can be used to allow tasks to decide whether they should wait on a resource or continue executing.

IX. Signals

Similar to traditional operating systems such as Linux, a signal is a bit used as an indicator to wake tasks up waiting on an event. In Nano-RK, a global variable called _nrk_signal_list is used to register particular bits with events while each task control block maintained by the operating system contains its own active bit mask corresponding to signals it is currently waiting on. Similar to semaphore, when a task is suspended on an event in Nano-RK it will not consume CPU resources supporting its role as an energy-aware and efficient resource kernel.

A task is suspended when it is waiting on some event and this state does not consume CPU time slide. In Nano-RK, the maximum size of signal stack is 32, which means there can be at most 32 unique signals. Each signal represents a bit in a 32 bit number. To wake up a task using a specific type of signal, first we need to do is create a signal and bind it to a task. A sequence of task states exist within Nano-RK outlined in Figure 7. The **ready** state corresponds to a task which is capable of being run when it is the current highest priority task in the system which is signified by the global variable nrk_high_ready_TCB pointing to its TCB structure. Upon being selected to run, the task's state will then be changed to **running** signifying that it is the sole currently operating task with its TCB structure pointed to by nrk_cur_task_TCB global variable. While it is currently executing, a task may encounter a resource that is not available or a event that is must wait for causing its state to become **SUSPENDED** and its event suspend status to become either **RSRC_EVENT_SUSPENDED** or **SIG_EVENT_SUSPENDED**. It waits in the suspended state until the proper event occurs when it will then be placed back on the ready queue until it is scheduled as the next high ready task. All functionality associated with events in Nano-RK is contained in *nrk_events.h/.cc* and a brief description of each is given below:

nrk_signal_delete() - If the bit of a signal has not been created by nrk_create_signal, it return NRK_ERROR. Otherwise, it first disables interrupts, then check if the task has been registered to that signal and will clear its active mask, put it in the SUSPENDED state. Finally, it removes the bit from the global registered mask of all signals, and enables interrupts again.

nrk_event_signal() - It will go through the TCB task list to find the task with sig_id, and further looks for those tasks which have their event suspend variable set to SIG_EVENT_SUSPENDED. If so, it will further check if their active signal mask bit is the same as the task sig_id. Then the process is placed in the SUSPENDED state, the next wakeup time is set to 0, and the task's event suspend flag is cleared. Finally, it enable interrupts again, and returns.

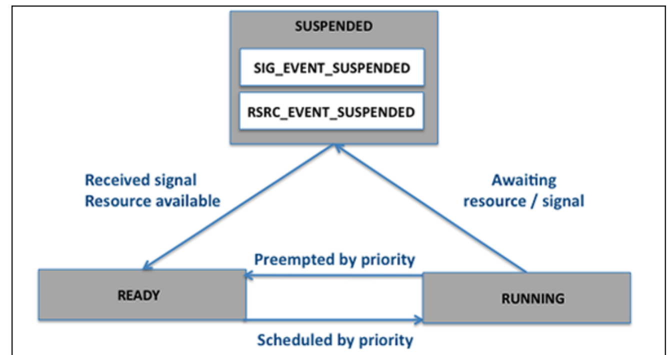


Fig. 7. Status a task can take on during its execution

nrk_event_wait() - This function will make sure the mask bits are registered by that process, and set the current task's active signal mask to be equal to the event mask passed in. It sets all the active bits to the inputting event_mask, and changes the event_suspend state to be SIG_EVENT_SUSPENDED. Next, it checks to see if this event mask has the signal to wait until the next wakeup. If the bit is set then the process will call nrk_wait_until_nw(), otherwise, it continues on executing but does get its bits set (the event_mask).

signal_register() - Upon calling this function, the current task will go into the global list of all signals that have been initialized so far, make sure the signal has actually even been created and if so it registers the current running task by changing its TCB-¿registered_mask bit by OR-ing it with the sig_id passed in as a parameter.

signal_unregister() - This function goes into the current task's TCB and makes sure the bit has been registered. If so, it clears the bit in both the registered mask and the active mask. Otherwise, it return NRK_ERROR.

get_registered_mask() - It returns the 32-bit signal mask from the TCB of the current task to show which bits have been set for signals or events.

X. The Nano-RK Ready Queue

So far we have only discussed Nano-RK's data structure for maintaining the operating system attributes of each task, namely, the *nrk_task_TCB* table. Although the *nrk_task_TCB* table holds the various flags and priority values necessary to schedule the tasks in this real-time operating system, iterating across the entire structure every time a task is to be scheduled would incur a performance cost on the system. For this, Nano-RK has implemented a double-linked list of ready queue nodes within a fixed-size array, termed the *ready queue*, that orders all **ready** tasks in **decreasing** order by whichever of the task's priorities (elevated priority vs. original task priority) is higher.

As the number of tasks running within the Nano-RK implementation is statically-configured in a testbed before deployment, the ready queue size is also fixed to this number

of tasks (including the idle task) that can be ready to run. A fixed-length array named *_nrk_readyQ* is found within the nrk_defs.h file along with two pointers to reference the two most important cells within this array. The free node pointer (*_free_node*) and the head node pointer (*_head_node*) point to the next cell in the array to be allocated and the current highest priority task ready to run, respectively.

Since the target hardware of Nano-RK is small embedded systems such as sensor nodes, using the least amount of memory within each object of the ready queue to store information necessary for scheduling is a primary concern. Nano-RK takes a light-weight approach to this problem by designing each ready queue node to only store (1) the task ID of a ready task and (2) two ready queue node pointers (previous and next) to implement the double-linked list structure. Any node reachable through following the *_head_node* pointer is inherently ordered in the ready queue by decreasing order. This means that the previous pointer of a node always points in the direction of nodes whose task IDs are of higher priority tasks while the next pointer always points in the direction of lower priority tasks. The list of free nodes are also double-linked together with the *_free_node* pointer that points to the next free cell to be allocated to a ready task node. To avoid maintaining a tail node pointer and having to iterate across the entire linked list to reach the tail, the cell pointed to by the *_free_node* always points to the "tail" node on the list of ready tasks.

The three main functions associated with the ready queue are contained within the nrk_task.c (.h) files and a brief explanation of their function are listed below:

nrk_get_high_ready_task_ID – This function retrieves the task identifier of the *_head_node*, the highest priority task ready to run, and returns it to the user. (Note: No error detection is used to detect whether the head node has a task associated with it or not)

nrk_add_to_readyQ – This function takes the task identifier of a task to be added to the ready queue and iterates across the current list of ready tasks searching for the proper position to insert a new node for the task such that ready queue remains sorted in decreasing order afterwards. As it iterates across ready queue it compares the highest priority, either the elevated priority (if set, due to a resource it is holding) or normal task priority, to the highest priority of the task to be added until the proper position is found. Using the *_free_node* pointer it allocates a new ready queue node to the task and redirects the pointers to maintain the order and linked list structure.

nrk_rem_from_readyQ – Using the task identifier passed in as a parameter, the function iterates across the list of ready queue nodes looking for the node whose task identifier matches that passed in. When a match is found it redirects the pointers appropriately to maintain the order and structure of the ready queue and returns.

XI. Scheduler

The core of Nano-RK is a static preemptive real-time scheduler which is priority-based and energy efficient. For priority-based preemptive scheduling, the scheduler always selects the highest priority task from the ready queue. To save energy, tasks do not poll for a resource but rather tasks will be blocked on certain events and can be unlocked when the events occur. When there is no task in the ready queue, the system can be powered down to save energy.

When the system is working, one and only one task (current task), signified by the nrk_cur_task_tcb, is running for a predefined period. So the most important job of the scheduler is to (1) decide which task should be run next and (2) for how long the next task should be run until the scheduler is triggered to run again.

For deciding the next task to run, the scheduler simply selects the highest priority task from the ready queue using the head_node pointer as shown in Figure 8. The code snippet is listed as following (in /src/kernel/source/nrk_scheduler.c):

```
...
task_ID = nrk_get_high_ready_task_ID();
nrk_high_ready_prio = nrk_task_TCB[task_ID].task_prio;
nrk_high_ready_TCB = &nrk_task_TCB[task_ID];
...
nrk_cur_task_prio = nrk_high_ready_prio;
nrk_cur_task_TCB = nrk_high_ready_TCB;
...
```

Fig. 8. Code snippet for assigning the next task to run

The second task the scheduler must perform is to calculate the period of time that should elapse before the scheduler is triggered to try and reschedule another task which is illustrated in Figure 9.

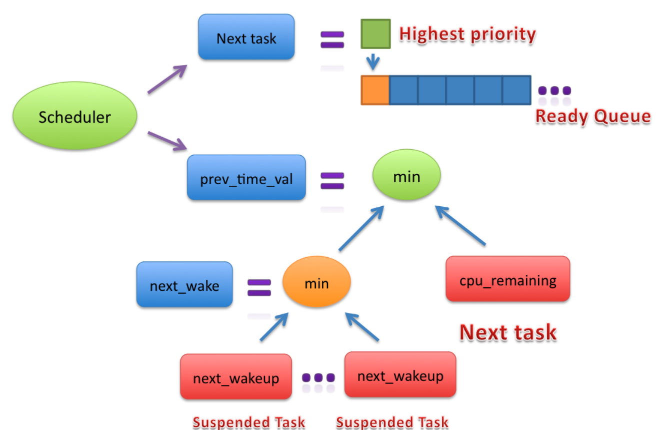


Fig. 9. The steps taken by the Nano-RK scheduler to determine the next task and how long the next task should run

Within the hardware of a sensor node, two timers exist, Timer 0 is designated as the interrupt timer and used to trigger the scheduler when it goes off. Using this hardware interrupt, the scheduler needs to calculate the number of elapsed ticks before the interrupt timer (Timer 0) is triggered again. To do this, the scheduler checks all suspended tasks

to get the minimum *next_wakeup* value which corresponds to the number of ticks before the task will be woken up again. Then, because the next task (the highest priority task in ready queue) is already determined, the scheduler compares the minimum *next_wakeup* it calculated with the *cpu_remaining* of the next highest priority task that should be scheduled. The smaller of the two values will be assigned to *_nrk_prev_timer_val*, and subsequently the interrupt timer (Timer 0). This value corresponds to the period of time in which the next task will run.

The code snippet that performs the task of calculating the previous timer value is located in /src/kernel/-source/nrk_scheduler.c and shown in Figure 10:

```
...
for (task_ID=0; task_ID < NRK_MAX_TASKS; task_ID++){
  if (nrk_task_TCB[task_ID].task_state == SUSPENDED ) {
    ...
    if(nrk_task_TCB[task_ID].next_wakeup!=0 &&
      nrk_task_TCB[task_ID].next_wakeup<next_wake ) {
      next_wake=nrk_task_TCB[task_ID].next_wakeup;
    }
  }
}
...
if(task_ID!=NRK_IDLE_TASK_ID){
  if(nrk_task_TCB[task_ID].cpu_reserve!=0 &&
      nrk_task_TCB[task_ID].cpu_remaining<MAX_SCHED_WAKEUP_TIME){
    if(next_wake>nrk_task_TCB[task_ID].cpu_remaining)
      next_wake=nrk_task_TCB[task_ID].cpu_remaining;
  } else {
    if(next_wake>MAX_SCHED_WAKEUP_TIME)
      next_wake=MAX_SCHED_WAKEUP_TIME;
  }
}
...
_nrk_prev_timer_val=next_wake;
...
nrk_start_high_ready_task();
```

Fig. 10. Code snippet of how the operating system determines the previous timer value

Besides these two main functions, the scheduler also updates the current task's *cpu_remaining* and all tasks' *next_wakeup* based on *_nrk_prev_timer_val* of current task.

XII. Conclusions

References

[1] A. Krogh, J. Hertz and R.G. Palmer, *Introduction to the Theory of Neural Computation.* Addison-Wesley, Redwook City, CA, 1991.

[2] D. Marr and T. Poggio, "Cooperative computation of stereo disparity," *Science*, vol. 195, pp. 283-328, 1977.

[3] Anand Eswaran and Anthony Rowe and Raj Rajkumar, "Nano-RK: an Energy-aware Resource-centric RTOS for Sensor Networks,",2005.

[4] Anand Eswaran and Anthony Rowe and Raj Rajkumar, "FireFly: A Time Synchronized Real-Time Sensor Networking Platform".

[5] Rahul Mangharam, Anthony Rowe, Raj Rajkumar ,Ryohei Suzuki, "Voice over Sensor Networks",2006.