NanoQplus 2 (Nano OS) Specification

2007. 11. 23

Sensor Network OS Research Team

Embedded S/W Division

ETRI 한국전자통신연구원
Electronics and Telecommunications Research Institute

Contents

List of Figures

1. Introduction

In recent years, the availability of cheap and small micro sensor node, and low power wireless communication enabled the large-scaled deployment of sensor nodes in Wireless Sensor Networks (WSN). WSN allows us to address, monitor, and eventually control a wide aspect of real-world problems. For instances, there are such applications as follows: monitoring health condition of our elder living independently at their home, tagging small animals unobtrusively, and tracking endangered species across large remote habitats, etc.

For practical use of these applications in real world, making the small-sized sensor nodes is important and, furthermore, through software algorithms supporting low-power mechanism, increasing the life-time of sensor nodes is more crucial in WSN. When sensor network programmers make a program for sensor network applications, the development of application is very difficult without any middleware or operating system. Therefore, we developed a nano operating system, referred to as Nano OS, to support the flexible and convenient programming mechanism of these low-power software algorithms.

There are a lot of research activities world-wide for sensor operating systems. Levis et al. proposed Berkeley's Tiny OS architecture designs and implementations. TinyOS is a well-known operating system and the Mote platform has been widely used in various applications. It features a component based architecture which enables rapid innovation and implementation while minimizing code size as required by the severe menmory constraints inherent in sensor networks. Han et. al. proposed a SOS operating system that consists of dynamically loadable modules and a kernel, which implements messaging, dynamic memory, and module loading and unloading, among other services. SOS is an operating system for Mote-class wireless sensor networks. Bhatti et al. proposed a MANTIS operating system. MANTIS provides a thread-based embedded operating system for wireless sensor networks. MANTIS supports preemptive multithreading. It also enables sensor nodes to natively interleave complex tasks with time-sensitive tasks, therby mitigating the bounded buffer producer-consumer problem.

Nano OS is a new multi-threaded, lightweight, and low-power sensor network operating system integrated with a general-purpose single-board hardware platform to enable flexible and rapid prototyping of WSN. The key design goals of Nano OS are ease of use, i.e. a small learning curve that encourages novice programmers to rapidly prototype novel sensor network applications, as well as flexibility, so that expert researchers can continue to adapt and extend the hardware/software system to apply the needs of their own advanced research.

In this documentation, we provide all about Nano OS; a philosophy, design issues, architecture, implementation details, installation and how-to-use it for sensor network applications.

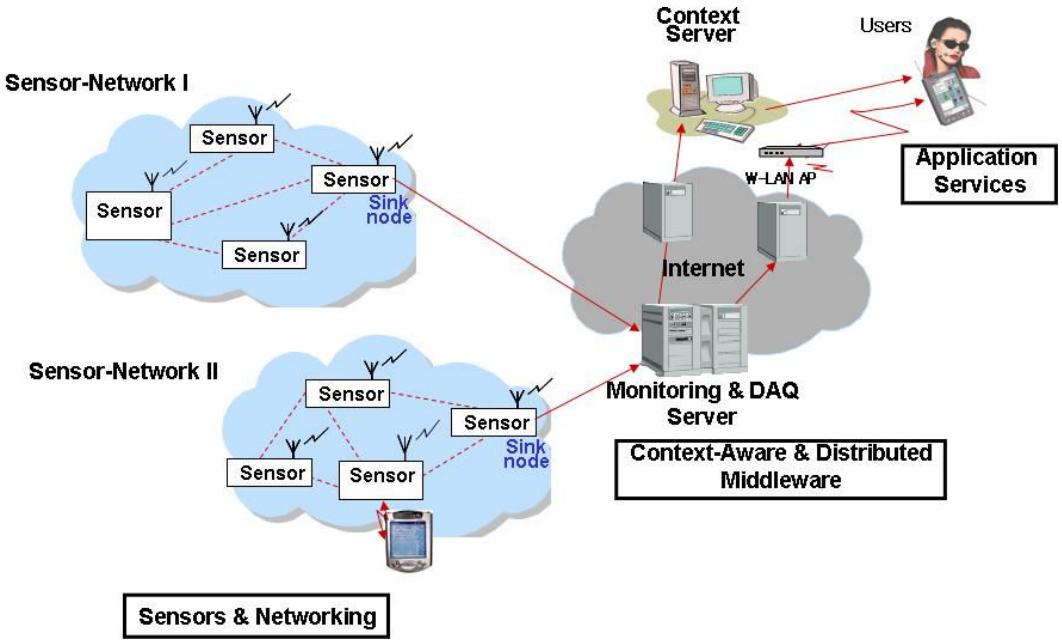

Figure 1. A sensor network model

Figure 1 shows a sensor network model that sensor data in environments are gathered and transferred to the end user. Usually, sensor nodes are formed into a group to communicate sensor data more efficiently. The sensed data is sent to the DAQ (Data Acquisition) server or context server for being monitored. Since the servers are connected with the end users directly or indirectly via wireless network or Internet, applications can use them for their own purpose.

Wireless sensor networks consist of hundreds or even thousands of very small sensor nodes. Since this sensor network model is fundamental in ubiquitous environments, Nano OS is based on this model.

Sensor operating systems installed on various sensor nodes must provide optimized environments for a specific sensor application to build wireless sensor network platforms efficiently, and support many kinds of devices and middlewares. The followings are design considerations in a sensor operating system.

1) Performance : Sensor nodes have several hardware constraints such as small memory, battery-powered and wireless communication. It will be a difficult task to have superior performance due to the hardware constraints. Nevertheless, good performance is a primary issue in sensor network research area. The kernel image should be quite small (with less than 10KB) and the sensor node with two AA-battery must be operated up to a few months by efficient power management. Also, sensing data needs to be transferred as fast as possible even in wireless communication.

2) Optimization : A sensor network consists of hundreds or even thousands of small sensor nodes. Thus, the hardware of a sensor node must be small in size if possible. The hardware

constraints of a sensor node affect the size of software loaded on it. The kernel size of a sensor operating system should be small by optimization. The cost of the sensor node is another optimization factor to be considered in design.

3) Energy efficiency : Sensor operating systems must support for providing information of how much energy is left in the sensor node. Kernel schedulers and wireless communication modules need to manage the energy consumption to ensure node´s long life-time even with less durable energy sources like batteries. Since there may be duplicate sensor nodes in wireless sensor network environments, we can consider of increasing the energy efficiency by using such a strategy.

4) Reliable communication : Wireless communication is assumed to be unreliable because there may be a lot of obstacles for signals to be transferred in the air. If sensor nodes are moving, the situation even becomes worse. A sensor operating system must have a reliable protocol for wireless communication.

5) Scalability : Various kinds of applications must be able to be built for a sensor operating system. In other words, the operating system should be designed considering the scalabilty and a standard interface among sensor node platforms.

6) Easy Programming : It is desirable that programmers can easily write sensor programs for sensor applications without the constraint of learning the whole operating system. For this, the sensor hardware should be abstracted from the user as much as possible in the sensor operating system, and a monitoring system for debugging must be supported.

2. Nano OS

Nano OS is a small operating system developed in ETRI for sensor network applications. Currently, it is developed based on ETRI-SSN sensor node platform and features multi-threading, dynamic module reconfiguration, and low-power management. Comparing with the berkerly TinyOS, which is based on the event-driven programming model, Nano OS provides the thread programming model for sensor nodes.

There are pros and cons of the event-driven and multi-threaded system. The event-driven system design gives us smaller context switching latency, more efficient memory usage based on single stack management, but response time and preemptivity are much poorer than that of the multi-threaded systems. On the other hand, the multi-threaded design enables preemption, but necessary memory space for the thread's stack is somewhat larger than that of the event-driven. However, they have, in theory, "duality" with respect to each other.

Nano OS provides a package of software that handle a variety of issues that may occur when forming sensor network applications such as hardware abstraction, task management, power management, RF message handling, routing, sensing, and actuating.

2.1. Architecture

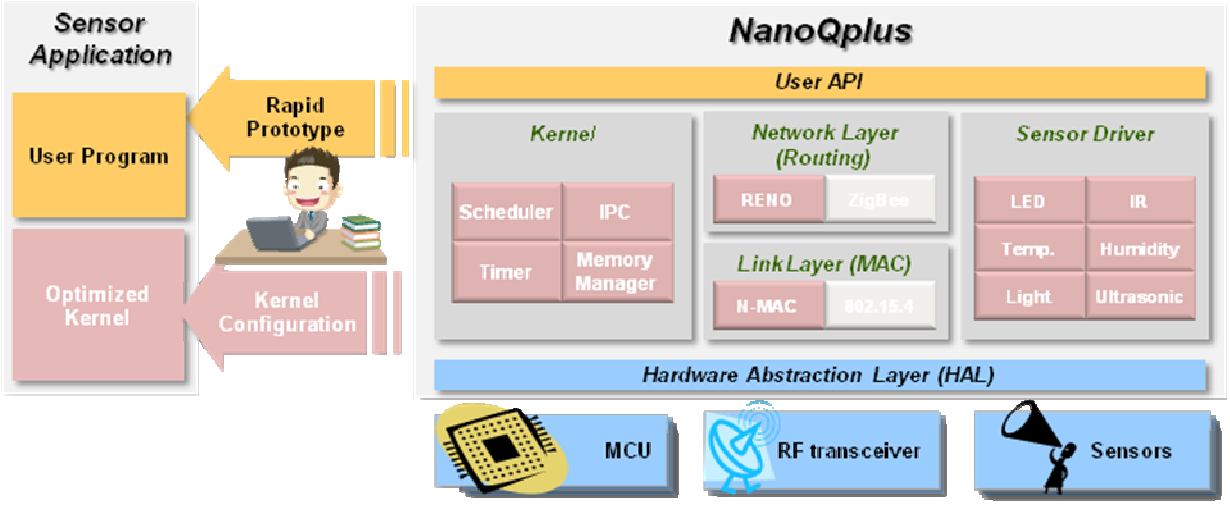

Figure 2. Nano OS architecture

Figure 2 shows Nano OS architecture. The Nano OS architecture resembles a classical modular and layered design, and consists of dynamically-loaded modules included in hardware part, operating system part, and application part, respectively. The hardware part is composed of MCU (e.g. ATmega128, CC2430, MSP430), RF module that can be CC2420 or other products for wireless communication, and Sensors/Actuators. The operating system part has a role as kernel scheduler and network protocol stack for handling RF messages, and it

has device driver modules, called as HAL, for abstracting the hardware part. The abstraction of hardware is one of basic characteristics in all operating systems. Further, the operating system part also offers the system APIs for convenient development of WSN applications to sensor networking programmers. In the end, the application part provides a way of interacting with the operating system part via system APIs.

2.2. Key Modules

1) HAL (Hardware Abstraction Layer) : HAL has a role as device driver modules for abstracting the hardware part and made up of several components such as LED, CLOCK, POWER, RFM (RF Module), UART, and ADC (Analog to Digital Converter).

2) Kernel : The kernel consists of thread management, memory management and power management. The thread management part initiates a task scheduler to perform context switching to handle thread operations. A thread executes a piece of code called “task”. Nano OS has a preemption-RR scheduler as a thread scheduler. The memory management part handles memory-related operations such as memory allocation or deallocation. The power management part controls the power modes provided by the CPU and CC2420. It attempts to minimize the power consumption of the sensor node.

3) Link Layer : The link layer manages one-hop distance wireless communication. This layer is strongly connected to the network layer. Currently, Nano OS supports the Nano Mac, which provides all basic MAC functions such as auto-ACK, data retransmission and CCA for wireless communication. Sooner or later, 802.15.4 MAC will be added in the link layer.

4) Network Layer : The network layer is responsible for sending the data from one sensor node to another sensor node by multiple hops. If the data was not sent to the destination, the source node should deal with the situation properly. The basic routing algorithm supported by Nano OS is referred to as RENO (A **R**eactive Routing Algorithm for **E**TRI **N**ano **O**S). It adopts an on-demand reactive routing protocol, managing the routing table. The Zigbee routing algorithm (the standard routing algorithm for wireless sensor networks) will be also included in Nano OS in the future.

There are several modules in Nano OS. They are LED, UART, Sensor(Gas, Light, Temperature, PIR, Ultra Sonic, Humidity, Battery), Actuator, Power Manager, Heap, Multi-Threaded Kernel, Semaphore, Message Queue, MAC (Nano MAC) and Routing(RENO Routing). Separating modules enables to reconfigure Nano OS applications. The dependencies among modules are shown in the figure.

- nos/kernel/mm : memory management files

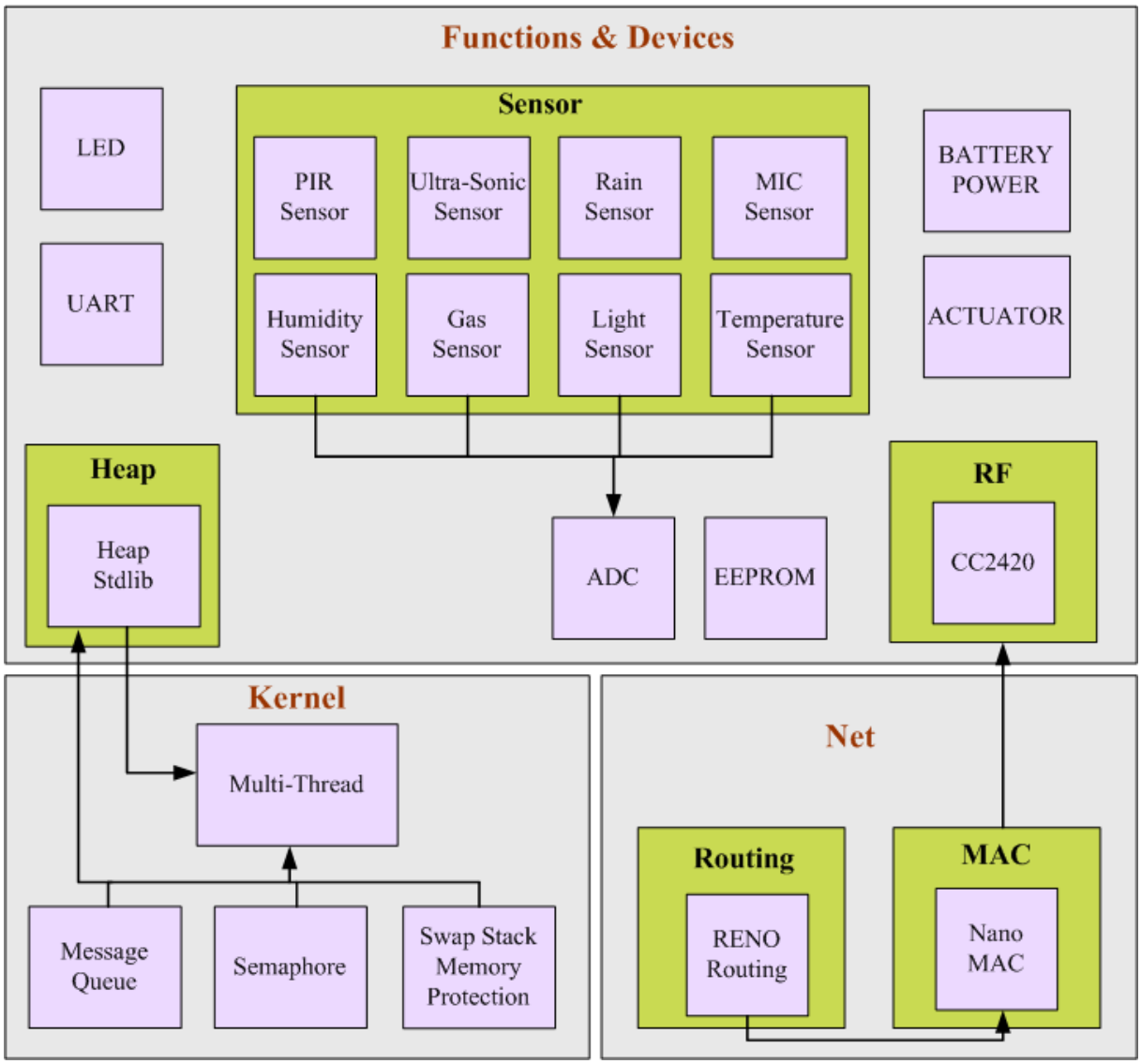

Figure 3. Module Dependency

Source codes such as kernel and device drivers are located in nos directory. To enhance portability, hardware dependent codes are separately located in nos/arch and nos/driver directory. The platform dependent codes are in nos/platform directory. The kernel directory, /nos/kernel, has only hardware independent codes. We assure that this approach will make Nano OS more portable. The directory tree of Nano OS is organized as follows.

.

```
nos
 |- include
 |- kernel
 |    +-thread
 |- net
 |    |- mac
 |    +- routing
 |- arch
 |    |- atmega128
 |          +- mm
 |    |- msp430
 |          +-mm
 |- drivers
 |    +- rf
 +- platform
       |- etri-ssn
       |- nano-24
       |- micaz
       |- zigbex
       |- isn-400n
       |- ubimote
       |- sky-z200
       |- tmote-sky
       |- ubicoin
       +- hmote2420
```

- nos/include : common header files and user-API related header files
- nos/kernel : kernel codes
- nos/kernel/thread : kernel thread implementations
- nos/net : network related codes
- nos/net/mac : MAC layer protocol implementation
- nos/ net/routing : network layer protocol implementation
- nos/arch : MCU dependent codes. It has subdirectories of MCUs.
- nos/arch/$(MCU) : MCU specific codes
- nos/arch/$(MCU)/mm : memory management files
- nos/drivers : platform independent codes in device drivers
- nos/drivers/rf : device drivers for RF chip products

- nos/platform : platform-specific implementation
- nos/platform/$(PLATFORM) : platform specific codes

2.3. Functionalities

Nano OS supports a variety of development environments for easy programming and debugging. Please refer to the web page, http://www.qplus.or.kr, for more information. The following characteristics are provided in Nano OS.

1) Supports a development tool for Nano OS, called “Nano Esto”
2) Supports code optimization (minimization) by Nano OS configuration function
3) Supports writing a sensor application rapidly by “Rapid Prototype” function
4) Supports testing programs to demonstrate each Nano OS module and provide examples.
5) Supports easy-to-code style in programming with C, thread-based C coding. (You don’t have to learn another program language)

Do you have any plans to build a nice sensor network application?
Then, Nano OS is the answer for the first step!

3. Nano OS Details

3.1. Kernel

3.1.1. OS initialization

All applications must call nos_init() to initialize Nano OS. The nos_init() function includes all functions necessary for initialization in a module.

void nos_init(void); **Parameters** None **Return Values** None **Description** Initialize all necessary modules if they are selected.

3.1.2. Memory Management

The memory management algorithm is an algorithm derived from avr-libc 1.4. In Nano OS, this algorithm is referred to as “standard heap library”, shortly “stdlib”. The heap structure in stdlib consists of a list of free blocks. At frist, when a memory allocation is requested, it attempts to find a block of the same size. If this attempt fails, it continues to search for a block of the nearest size, instead. If it still fails, it extends the heap size and allocates a memory segment to it. When a memory segment is deallocated, the stdlib algorithm merges the deallocated block into the adjacent block, forming a large block.

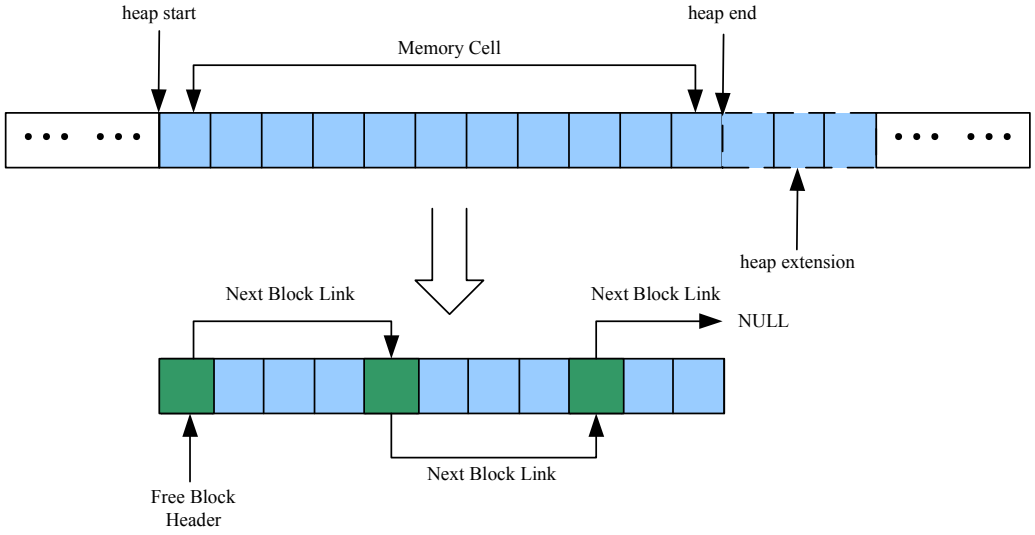

Figure 4. Heap memory architecture

The fragmentation problem is addressed by employing the linear heap architecture. The heap stdlib has a ‘list’ made of free blocks.

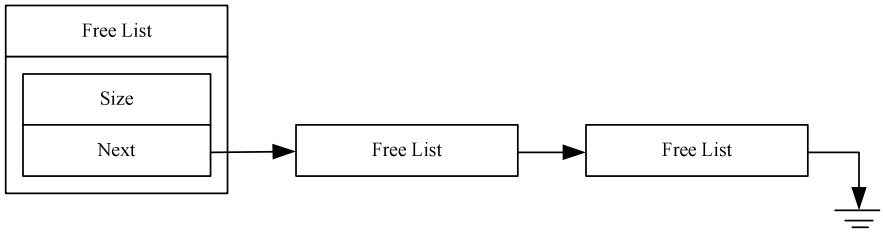

Figure 5. Free block list

The actual code of the free block structure is

```
struct __freelist {
        UINT16 sz;
        struct __freelist *nx;
};
```

(1) heap initialization

void NOS_HEAP_INIT(void);

Parameters

No Parameter

Return Values

The size of heap created

Description

Initialize the heap memory

(2) heap memory allocation

The stdlib_malloc() function compares the requested memory size with the minimum size that can be allocated, and then changes the requested memory size. At the first search, it scans through the free blocks of the same size and saves the nearest of them. If multiple free blocks of the same size are found, it returns the corresponding block after removing the blocks from free blocks.

If the first search fails, the second search proceeds with the most adjacent blocks. After sorting the requested blocks, it attempts to find a matched block in the search. If a block of the sorted size matches with a free block, it returns the corresponding block, not a free block. If the searched block is larger in size, the block is divided into an allocated part and the other part of the the block. Keeping the other part, it returns the pointer of the matched block.

If no matching block has been found at the end of the second search, it extends the heap size and allocates the memory of the requested size, and returns the pointer of the allocated one.

void *nos_malloc(UINT16 len);

Parameters

len : memory size to be allocated

Return Values

If successful, returns a memory pointer for the allocated memory.

If failed, returns NULL.

Description

Allocate a memory segment with the length **len**

(3) heap memory deallocation

To deallocate the heap memory segment, it is necessary to connect the returned block to free block list of the heap. When it is connected, if they are adjacent, then it merges into a large free block.

void nos_free(void *p);

Parameters

p : return address of the allocated memory

Return Values

No value

Description

Deallocate the memory pointed by **p**.

3.1.3. Thread Management

In Nano OS, the kernel scheduler uses a round-robin algorithm with a priority queue. The priorities are categorized into 6 levels (0-5), given as follows.

Level	Macro	Description
5	PRIORITY_ULTRA	The highest priority level, reserved by the system
4	PRIORITY_HIGHEST	The user application level thread. Schedulable with one of 4 levels according to application characteristics.
3	PRIORITY_HIGH	
2	PRIORITY_NORMAL	
1	PRIORITY_LOW	
0	PRIORITY_LOWEST	The lowest priority level, reserved by the system as an idle thread.

Nano OS manages thread queues with a 16-bit variable. Since a thread can be represented

with 3 bits, maximally, 5 threads (3x15=15bits) can be supported by a 16-bit variable. The following figure shows a thread queue with 5 threads (3, 4, 2, 1, 5) that are inserted. The **queue** is 16-bit variable and each thread is indexed with the 8-bit **idx** variable. Unlike the thread queue, attributes of each thread are recorded in 8-bit variable.

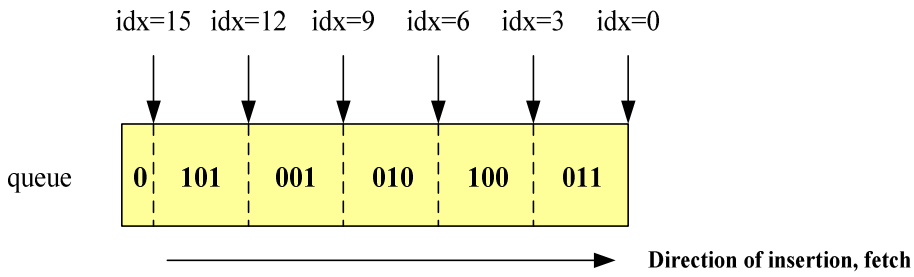

Figure 6. Thread Queue

The idle thread can have a task list. If the program registers a task in the task list, the task is executed when the idle thread is resumed. Since the work is registered as a function, there is no overhead of creating and managing stacks. It can be used for checking system or batch processing.

(1) Scheduler initialization

The nos_sched_init() function initializes the scheduling related variables and TCB (thread control blocks) data structure, and allocates memory blocks to them. It initializes the work queue and calls the nos_sched_hal_init() function for scheduling.

void nos_sched_init();
Parameters
No Parameter
Return Values
No value
Description
Initialize the scheduler

(2) Timer initialization

The nos_sched_hal_init() initializes the hardware timer and its ISR (interrupt service routine). This function sets timer-related registers so that hardware timer interrupt can occur at every 5, 10 or 32 ms.

The timer service routine decreases the tick count of a waiting thread at every timer interrupt.

If the tick count of a thread is equal to 0, the thread is awaken and inserted into the ready queue to be READY_STATE.

void nos_sched_hal_init(); **Parameters** No Parameter **Return Values** No value **Description** Initialize the timer for the scheduler

Hardware timer for scheduler ISR **Parameters** No Parameter **Return Values** No value **Description** Service routine for the scheduler timer

(3) Thread context switching

The context switching function saves all registers of a currently running thread and inserts the thread into the ready queue with the corresponding priority. Then, it searches a highest priority thread and deletes it from the ready queue. If the resultant ready queue is empty, the ready priority queue flag is cleared indicating that there is no thread in the ready queue. After selected thread gets to be in RUNNING_STATE, the stack pointer points to the selected thread and all register values for the selected thread are restored to resume the thread.

void nos_ctx_sw(); **Parameters** No Parameter **Return Values** No value **Description** Perform a context switch to a highest priority thread in the ready queue

(4) Thread scheduling

The scheduler is initiated by the nos_sched_start() function. It is called by the main()

function, and then it becomes an idle thread with the lowest priority. It sets the timer 0 and calls the thread switching function to perform a first context switch. Thereafter, it performs a context switch if there is a thread in the ready queue. If there is no thread in the ready queue, it will make MCU sleep by invoking the Sleep() function as an idle thread.

void nos_sched_start();
Parameters
No Parameter
Return Values
No value
Description
Begin to schedule user threads

(5) Thread APIs

Nano OS thread APIs will obey rules recommeded by POSIX. We support various thread related APIs such as the thread creation and destruction, synchronization, suspending and resuming, etc.

TCB (thread control block) contains the control information for a thread.

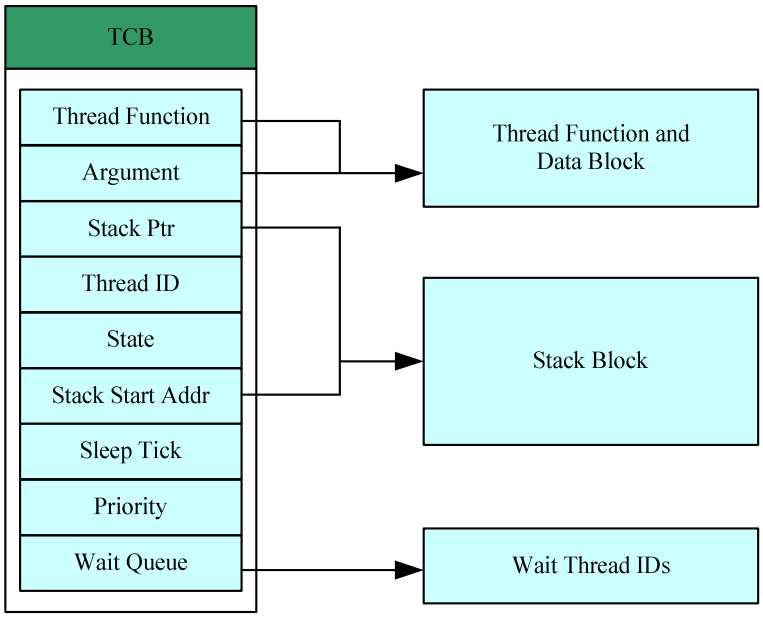

Figure 7. TCB structure

The thread function saves the locations at which thread and arguments will resume. The ‘Thread ID’ is a unique id of the thread and the ‘State’ is the state of the thread. The ‘Stack Ptr’ saves the current stack pointer, and ‘Stack Start Addr’ is used for checking the stack

overflow. The ‘Sleep Tick’ is set when the thread_sleep( ) function is called, and the ‘Priority’ stores the priority level of the thread. The ‘Wait Queue’ has a list of waitng threads. The number of TCB blocks are controlled by the macro MAX_NUM_TOTAL_THREAD. The actual TCB structure is shown here:

```
typedef struct tcb
{
        void            (*func)(void *);
        void            *args_data;
        STACK_PTR       sptr;
        UINT8                   id;
        UINT8                   state;
        STACK_PTR       stack_start_addr;
        UINT16          sleep_tick;
        UINT8           priority;
        THREAD_QUEUE        wait_q;
} *TCB;
```

The thread state is set as one of the five states.

```
READY_STATE : the ready state for execution
RUNNING_STATE : the currently running program
WAITING_STATE : the waiting state for an event
SLEEPING_STATE : the sleeping state
EXIT_STATE : the exit state to be terminated
```

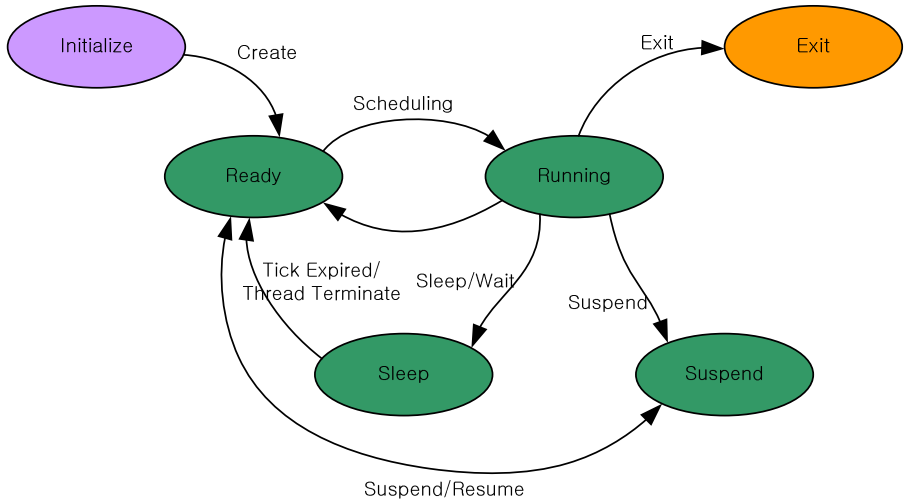

Figure 8. Thread state transition diagram

When a thread related function is called, one of the following values are returned.

- THREAD_NO_ERROR : success
- THREAD_ID_DUPLICATE_ERROR : the thread id already exists.
- THREAD_PRIORITY_OUT_OF_RANGE_ERROR : the priority set is out of the allowed range.
- THREAD_PRIORITY_CHANGE_ERROR : the system cannot change its priority.
- THREAD_RESUME_ERROR : the system cannot resume the thread.

1) Thread creation

The thread_create() checks if the tid value is valid. If valid, it creates and sets up a TCB block for that thread. Then it inserts the thread into a ready queue, which is related to the priority of the thread.

UINT8 nos_thread_create(void (*func)(void *args), void *args_data, UINT16 stack_size, UINT8 priority);

Parameters

tid : unique thread ID

func : entry point of thread

args_data : a pointer of arguments

stack_size : thread stack size

priority : thread priority

Return Values

created thread id

Description

Create a thread with **tid** whose priority is **priority**, whose stack size is **stack_size**, that will execute the function **func(args)**.

2) Thread destruction

The thread_exit() function wakes up all waiting threads for this thread and inserts them into the ready queue. All memory segments for the thread (e.g. stack, TCB) are deallocated and the state is changed to be EXIT_STATE. In the end, it performs a context switch to resume one of the other threads in the ready queue.

void nos_thread_exit();

Parameters

No parameter

Return Values

No Value

Description

Destroy the currently running thread

3) Thread joining (for synchronization)

The thread_join() function checks if a target thread is created. If it is, thread_join inserts the thread id into the waiting queue of the target thread, and then changes its state to WAIT_STATE. At the last stage, it performs a context switching for other threads to be executed.

void nos_thread_join(UINT8 tid);

Parameters

tid : thread ID to be joined

Return Values

No Value

Description

Wait until the thread **tid** terminates

4) Changing thread priority

The thread_priority_change () function takes the thread id and the priority as parameters to perform the change. After checking validation of the change, it attempts to change the target thread's priority. If the thread state is in READY_STATE, the target thread is deleted from the ready queue with the previous priority and inserted into the ready queue with the new priority.

UINT8 nos_thread_priority_change(UINT8 tid, UINT8 new_priority);

Parameters

tid : target thread ID for priority change

new_priority : priority level to be changed

Return Values

THREAD_NO_ERROR : success

THREAD_PRIORITY_CHANGE_ERROR : fail

Description

Change the priority of the thread **tid** to the **new_priority**

5) Suspending and Resuming the thread

The thread_suspend() function can suspend the thread itself or the other thread. In case of suspending itself, it sets the suspend flag and becomes in READY_STATE. Then, it performs a context switching function to schedule the other threads. In case of suspending the other thread, it also sets the suspend flag. If the state of the other thread is in READY_STATE, it deletes the thread from the ready queue. If there are not any threads in ready queue, it unsets the ready queue flag.

The thread_resume() function resumes the given thread (it means that the thread is inserted into the ready queue for execution) that was suspended. Since the thread cannot resume itself, it must be checked before doing any further tasks. The function unsets the suspend flag, and if the thread state is READY_STATE, then the thread is inserted into the ready queue with the corresponding priority.

void nos_thread_suspend(UINT8 tid); **Parameters** tid : thread ID to be suspended **Return Values** No value **Description** Suspend the thread **tid**

UINT8 nos_thread_resume(UINT8 tid); **Parameters** tid : target thread ID **Return Values** THREAD_RESUME_ERROR : fail THREAD_NO_ERROR : success **Description** Resume the thread **tid**

6) Dealying thread

To delay a thread for given time, use the thread_sleep(UINT16 ticks) function. This function delays the execution of the thread for **ticks** time.

void nos_thread_sleep(UINT16 ticks); **Parameters** ms : the delay time in tick, where a tick is scheduling time unit in milli-second **Return Values**

No value

Description

Delay the thread for **ticks** time. Note that the resolution of the delay time is the context switching period (e.g. 50ms, or 250 ms).

The ticks depends on the context switching time of the the kernel. So we provide the following functions for the input of time. They are redefined with the nos_thread_sleep(…) function.

#define nos_thread_sleep_ms(ms)

Parameters

ms : time in milli-secon

#define nos_thread_sleep_sec(sec)

Parameters

sec : time in second

7) Waking up thread

To wake up a thread in asleep, use the thread_wakeup(UINT8 tid) function.

UINT8 nos_thread_wakeup(UINT8 tid);

Parameters

tid : target thread ID

Return Values

THREAD_WAKEUP_ERROR : fail

THREAD_NO_ERROR : success

Description

Wake up the target thread with tid if it is in asleep.

3.1.4. Inter-Thread Communication (ITC)

In multi-threaded operating system, threads sometimes need to communicate among each other. In this section, two kernel objects, the message queue and the semaphore, that are used for inter-thread communications in Nano OS, were introduced. These objects are important in thread programming because they are used for handling the thread synchronization.

3.1.4.1. Message Queue

The message queue is a kernel object that enables to send or receive data among threads. The message queue should be created before use. If it is no longer used, it must be deleted. Message queues work in a FIFO principle (First In, First Out).

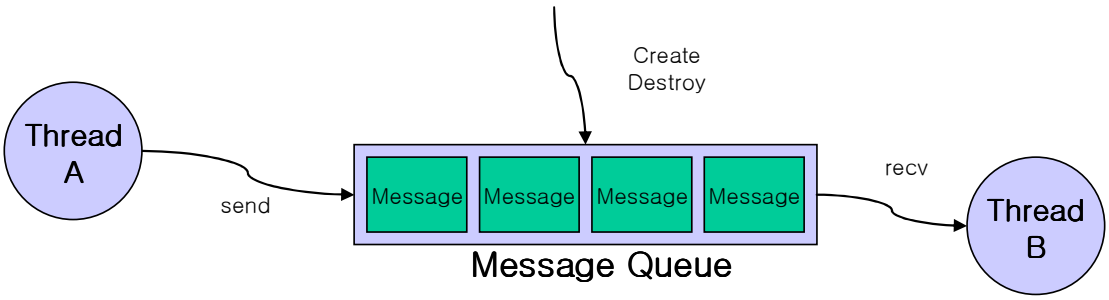

Figure 9. Message queue

In Nano OS, threads can continue to perform their execution even after a thread sends a message to the message queue, regardless of the message transmitted to the receiver thread (*non-blocking send*). If there is no message to read from the message queue, the receiver thread is blocked until there is a new message in the queue (*blocking receive*). Different types of message queues can be created according to the message type. Nano OS supports both the non-blocking and blocking communication through the message queue.

The message queue is handled as the Fig. 10.

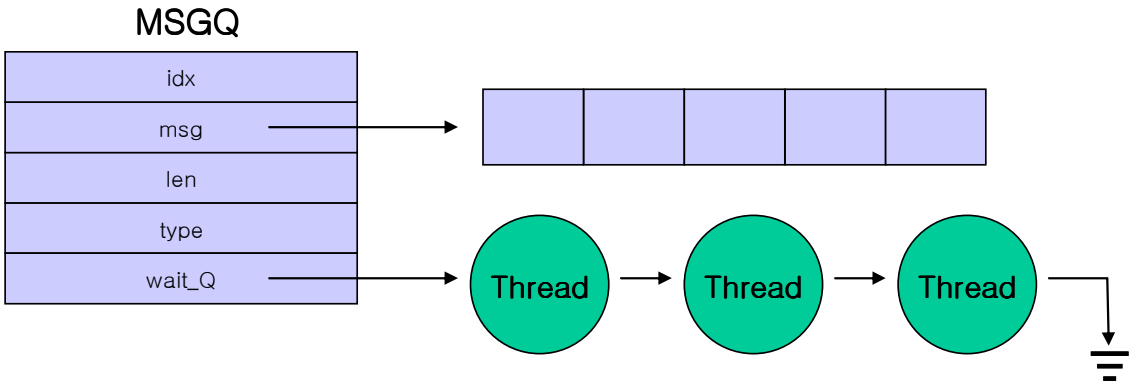

Figure 10. Message queue structure

The message queue has the following 4 types.

MSGQ_UCHAR : unsigned character MSGQ_CHAR : character MSGQ_UINT : unsigned integer MSGQ_INT : integer

The actual code of the message queue looks like this.

```
typedef struct msgq
```

```
{
    UINT8 idx; // queue index
    void *msg; // message array
    UINT8 len; // queue size
    UINT8 type; // message type in the queue
    THREAD_QUEUE wait_Q; // waiting thread queue
} *MSGQ;
```

(1) Message queue creation

The msgq_create() function allocates the message queue structure and configures the queue structure for the message types and the length of the queue properly. It returns the queue structure if succeeded.

```
MSGQ msgq_create(UINT8 type, UINT8 len);
```

Parameters

type : data type of a message to be stored

len : size of the message queue

Return Values

The structure of the message queue created

Description

Create a message queue with the message type **type** and the length **len**

(2) Message queue deletion

The msgq_destroy() function deallocates the message queue structure and exits the function.

```
void msgq_destroy(MSGQ mq);
```

Parameters

mq : message queue to be destroyed

Return Values

None

Description

Destroy the message queue **mq**

(3) Sending message (blocking send)

To send a message to another thread, the message has to be saved into the message queue using the msgq_send() function. The function checks if there is room left for the message. If there is no room, it returns MSGQ_FULL_ERROR. Then, if it gets a wrong type, it returns

MSGQ_TYPE_MAT_ERROR. Otherwise, it performs typecasting for the message and inserts it into the message queue.

UINT8 msgq_send(MSGQ mq, void *data); **Parameters** mq : message queue to which a message will be sent data : pointer of data to be sent **Return Values** Returns the result of sending MSGQ_NO_ERROR : success MSGQ_FULL_ERROR : the message queue is full MSGQ_TYPE_MATCH_ERROR : the types do not match **Description** Send a message pointed by **data** to the message queue **mq**

(4) Receiving message (blocking receive)

To receive a message from the message queue, the msgq_recv() function is used. It checks if there is a message in the message queue, and if any, then it typecasts the message for the data and returns MSGQ_NO_ERROR. Otherwise, the thread is on the waiting list and immediately performs a context switching.

UINT8 msgq_recv(MSGQ mq, void *data); **Parameters** mq : The message queue at which a message will be arrived data : The pointer of data to be received **Return Values** Return the result of receiving MSGQ_NO_ERROR : Success MSGQ_TYPE_MATCH_ERROR : Message types do not match **Description** Receive a message, that will be stored at the pointer **data**, from the message queue **mq**

(5) Sending message (non-blocking send)

In non-blocking send, program control returns immediately after sending data to the message queue. So, it does not guarantee the delivery of the message to the message queue because it returns program control even if it didn't send the message now. The rest is the same as the operation of the blocking send in the message queue. To wait for the message sent to deliver

to the message queue completely, use the nos_msgq_wait_until_isend_complete(mq) function.

UINT8 msgq_isend(MSGQ mq, void *data);

Parameters

mq : message queue to which a message will be sent

data : pointer of data to be sent

Return Values

Returns the result of sending

MSGQ_NO_ERROR : success

MSGQ_FULL_ERROR : the message queue is full

MSGQ_TYPE_MATCH_ERROR : the types do not match

Description

Send a message pointed by **data** to the message queue **mq** in non-blocking mode

(6) Receiving message (non-blocking receive)

In non-blocking receive, program control returns immediately after receiving data to the message queue. So, it does not guarantee the delivery of the message from the message queue because it returns program control even if it didn't receive the message now. The rest is the same as the operation of the blocking receive in the message queue. To wait for the message to be delivered from the message queue completely, use the nos_msgq_wait_until_irecv_complete(mq) function.

UINT8 msgq_irecv(MSGQ mq, void *data);

Parameters

mq : The message queue at which a message will be arrived

data : The pointer of data to be received

Return Values

Return the result of receiving

MSGQ_NO_ERROR : Success

MSGQ_TYPE_MATCH_ERROR : Message types do not match

Description

Receive a message in non-blocking mode, that will be stored at the pointer **data**, from the message queue **mq**

3.1.4.2. Semaphore

The semaphore is used for synchronization among threads in multi-threaded environments. Semaphore should be created and configured before use. Whenever the thread calls the wait() function, the semaphore value decreases. The wait() function obtains the semaphore as long as the semaphore value is greater than 0. If the semaphore value is equal to 0, the thread calling the wait() function changes into waiting state. Threads that do not have to use the semaphore call signal() function to return their semaphores. In this case, the semaphore values will increase, and the other threads that require the semaphores will get an opportunity of obtaining them. The semaphore that is no longer used shall be deallocated.

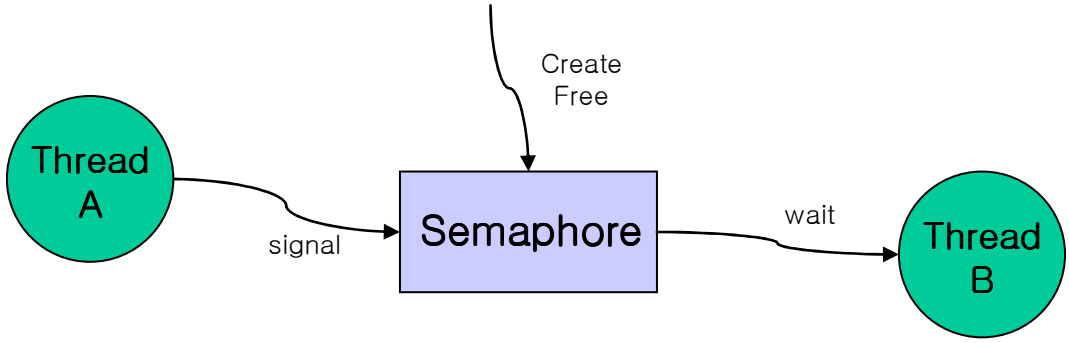

Figure 11. Semaphore

Information of a semaphore is addressed in the semaphore structure shown below. All semaphores have an internal semaphore value. According to the value, the threads that require the semaphore can either acquire the semaphore or become in waiting state. A waiting queue in a semaphore structure has a list of threads that want to acquire the semaphore.

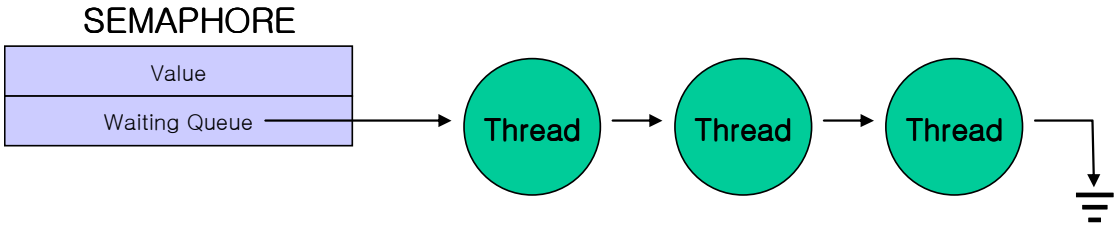

Figure 12. Semaphore structure

The actual code of semaphore structure looks like this.

```
typedef struct semaphore
{
 UINT8 val; // Semaphore value
 THREAD_QUEUE wait_q; // Wait queue
} *SEMAPHORE
```

(1) Semaphore creation

The semaphore_create() gets the semaphore value as a parameter, which is the maximum value that the semaphore can have. That is, this value directly indicates the number of threads that can acquire the semaphore at the same time. If succeeded, it returns the pointer of semaphore structure.

SEMAPHORE semaphore_create(UINT8 value);

Parameters

value : the maximum value of the semaphore

Return Values

pointer of the semaphore structure created

Description

Creates a semaphore with the semaphore value **value**

(2) Semaphore destruction

A semaphore that is no longer used must be deallocated.

void semaphore_free(SEMAPHORE sem);

Parameters

sem : semaphore structure to be deallocated

Return Values

None

Description

Deallocate the semaphore **sem**

(3) Semaphore acquisition

The semaphore_wait() function is used for a thread to acquire a semaphore.

void semaphore_wait(SEMAPHORE sem);

Parameters

sem : semaphore structure to acquire

Return Values

None.

Description

Acquire the semaphore **sem**

(4) Semaphore release

The semaphore_signal() function release the semaphore so that other threads can acquire it.

void semaphore_signal(SEMAPHORE sem); **Parameters** sem : semaphore structure to release **Return Values** None. **Description** Release the semaphore **sem**

3.1.5. Power Management

Nano OS controls sleep modes of MCU to reduce the energy comsumption.
Sleep modes are defined as macros shown below.

_IDLE : Set the MCU idle state _PWR_SAVE : Set the MCU power save state _EXT_STANDBY : Set the MCU external standby state

(1) Setting MCU sleep mode

Before calling NOS_SLEEP_MCU() to go into sleep state, the NOS_SET_SLEEP_MODE() function must be called to select the sleep mode of MCU.

#define NOS_SET_SLEEP_MODE(sleep_mode) **Parameters** sleep_mode : Sets the power mode before calling hal_sleep_mcu() function

(2) Enabling or disabling MCU sleep function

In case of ATmega128 MCU, the sleep bit of the sleep register should be active before setting the sleep mode. Nano OS provides the functions to enable/disable the MCU sleep function.

#define NOS_SLEEP_ENABLE() **Parameters** None.

#define NOS_SLEEP_DISABLE() **Parameters** None.

(3) MCU sleep

The MCU goes into the selected mode designated by the NOS_SET_SLEEP_MODE() function. The NOS_SLEEP_MCU() function should be called after NOS_SLEEP_ENABLE() function and before NOS_SLEEP_DISABLE() function.

#define NOS_SLEEP_MCU() **Parameters**

None.

3.1.6. Kernel Timer

Nano OS supports kernel timer functions. The software implemented timers can invoke a function whenever the timers are expired. Nano OS supports up to 8 timers (timer id is from 0 to 7)

(1) Timer creation

A timer is created with the nos_timer_create(…) function.

UINT8 nos_timer_create(void (*func)(void), UINT16 ticks, UINT8 opt);

Parameters

tmid : pointer of timer id (one within from 0 to7)

func : function to be executed when a timer is expired

ticks : ticks until expiration

opt : timer option. If opt is TIMER_ONE_SHOT, this timer executes the function only once. If opt is TIMER_PERIODIC, this timer executes the function whenever timer is expired.

Return Values

TIMER_DUPLICATE_ERROR : fail. The timer id is duplicated.

created timer ID : success

Description

Creates a timer with **tmid**

The ticks depends on the context switching time of the the kernel. So we provide the following functions for the input of time. They are redefined with the nos_timer_create(…) function.

#define nos_timer_create_ms(tmid, func, ms, opt)

Parameters

tmid : pointer of timer id (one within from 0 to7)

func : function to be executed when a timer is expired

ms : time in milli-second

opt : timer option. If opt is TIMER_ONE_SHOT, this timer executes the function only once. If opt is TIMER_PERIODIC, this timer executes the function whenever timer is expired.

#define nos_timer_create_sec(tmid, func, sec, opt)

Parameters
tmid : pointer of timer id (one within from 0 to7)
func : function to be executed when a timer is expired
sec : time in second
opt : timer option. If opt is TIMER_ONE_SHOT, this timer executes the function only once. If opt is TIMER_PERIODIC, this timer executes the function whenever timer is expired.

#define nos_timer_create_min(tmid, func, min, opt)
Parameters
tmid : pointer of timer id (one within from 0 to7)
func : function to be executed when a timer is expired
min : time in minute
opt : timer option. If opt is TIMER_ONE_SHOT, this timer executes the function only once. If opt is TIMER_PERIODIC, this timer executes the function whenever timer is expired.

(2) Timer destruction

A timer is destroyed with the nos_timer_destroy(…) function.

void nos_timer_destroy(UINT8 tmid);
Parameters
tmid : timer id (one within from 0 to7)
Return Values
None.
Description
Destroy the timer with **tmid**

(3) Timer activation

A timer begins to work after being activated. For activation, the nos_timer_activate(…) function is used.

UINT8 nos_timer_activate(UINT8 tmid);
Parameters
tmid : timer id (one within from 0 to7)
Return Values
TIMER_ACTIVATE_ERROR : fail.

TIMER_NO_ERROR : success

Description

Activate the timer with **tmid**

(4) Timer deactivation

The nos_timer_deactivate(…) function deactivates the activated timer.

UINT8 nos_timer_deactivate(UINT8 tmid);

Parameters

tmid : timer id (one within from 0 to7)

Return Values

TIMER_DEACTIVATE_ERROR : fail.

TIMER_NO_ERROR : success

Description

Deactivate the timer with **tmid**

3.2. Network

3.2.1. MAC

3.2.1.1. Nano MAC

(1) Data structures

The Nano MAC has the following structure.

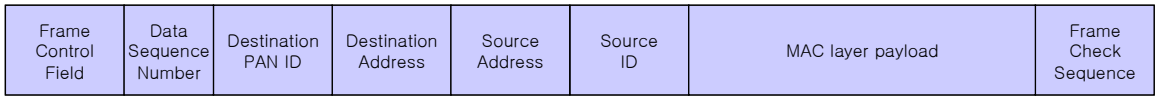

Figure 13. MAC frame structure

Frames are managed in the form of NMAC_TX_INFO, NMAC_RX_INFO and NMAC_RXQ_ENTITY structures.

```
typedef struct _nmac_tx_info
{
    UINT8   payload_length;   // length of frame payload
    WORD    dest_addr;        // Destination address
#ifdef ROUTING_M
    void    *routing_header_ptr;     // Network layer header
#endif
    void    *payload_ptr;         // payload
} NMAC_TX_INFO;
```

```
typedef struct _nmac_rx_info
{
    UINT8   payload_length;   // length of frame payload
    WORD      src_addr;         // Source address
    void        *payload_ptr;       // payload
} NMAC_RX_INFO;
```

```
typedef struct _nmac_rx_queue
{
    UINT8   payload_length;   // length of frame payload
    WORD      src_addr;         // Source address
    INT8        rssi;               //Received Signal Strength Indicator
    UINT8       corr;               //Correlation value (unsigned 7bit) for LQI
    BYTE        payload[NMAC_MAX_PAYLOAD_SIZE];       // payload
```

```
} NMAC_RXQ_ENTITY;
```

When the routing module is not used, the RXQUEUE structure is used to store the received message in MAC layer. The RXQUEUE is a circular queue, which has the head and tail of the queue, number of items, and actual data.

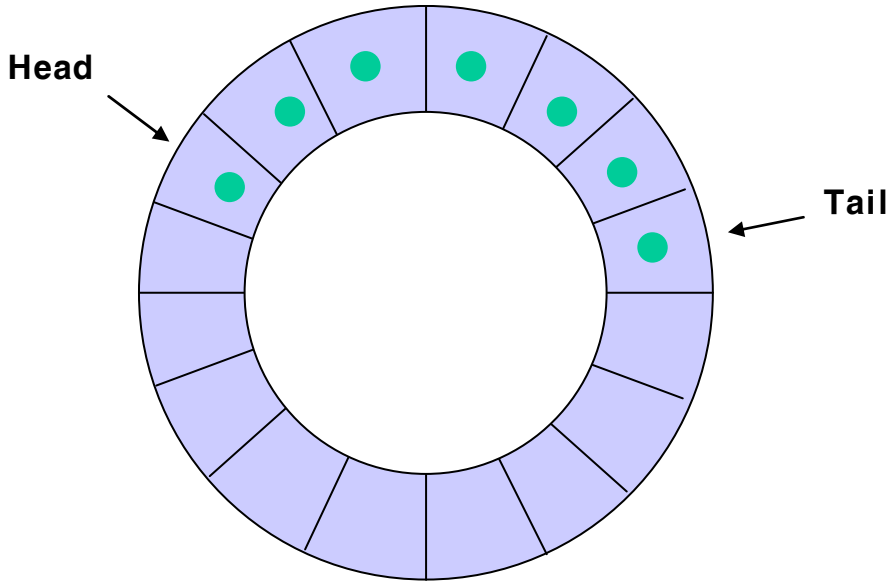

Figure 14. Receiver queue

```
volatile static struct _nmac_rx_queue
{
    UINT8       front, rear, nitem;          // front, rear of the queue
    NMAC_RXQ_ENTITY     data[NMAC_RXQ _LEN];    // queue
} NMAC_RX_QUEUE;
```

The RF_ENV structure is used to store some pieces of information in MAC layer.

```
static struct _nmac_env
{
    UINT8               tx_seq;     // Sequence number
    volatile UINT8      rx_seq;     // Sequence number
    WORD                panid;          // PAN ID
    WORD                address;        // Node address
    BOOL                rx_on_state;    // store RF recv state that user specified
#ifdef DEMO_M
    UIN16               min_permit_addr;
    UIN16               max_permit_addr;
```

```
#endif
} NMAC_ENV;
```

(2) Module initialization

In order to use the RF for communication, the nmac_init() function must be called. This function performs initializations of RF chip, rf_env and rx_queue, and then sets PAN ID and address of the node. Besides, it registers a handler to process the receiving data when interrupted and activates the interrupts.

```
void nmac_init(void)
Parameters
  UINT8 channel, UINT16 panid, UINT16 myaddr
Return Values
  None
```

(3) Sending packet

The nmac_tx() function is used to send packets through MAC layer from application or network layer. When this function is called, it is followed by the procedures.

```
BOOL nmac_tx(NMAC_TX_INFO* nmac_tx_info_ptr)
Parameters
  nmac_tx_info_ptr : data information to send
Return Values
  TRUE : success to send
  FALSE : no ACK received (fail)
```

1. If ack is requested, it turns on the receiver.
2. Empty the TXFIFO and then write new data into TXFIFO
3. Transmit a frame by CSMA-CA algorithm
4. If Ack is missing, attempts TX up to 3 times.
5. If the RF was being turned off, it turns off RF again and returns the result for sending.

(4) Receiving packet

The nmac_rx() function reads a frame from the MAC RX queue and returns TRUE if the MAC RX queue is not emtpy or returns FALSE if the MAC RX queue is empty.

```
void nmac_rx(NMAC_RX_INFO* nmac_rx_info_ptr);
```

Parameters

nmac_rx_info_ptr : pointer of receiver information buffer

Return Values

True if there is a packet received, which is in the queue

False if there is no packet in the queue

The nmac_rx_handler() is called from ISR and handles the received frame from RF chip. It is registered as a call back function of ISR when the RF module is initialized. This function handles the Ack frame and normal data frame. If an Ack has been arrived, it sets the RF_ENV.tx_ack_received. It means the TX frame has been transmitted successfully. If it is a normal data frame, it queues the whole frame in the MAC RX queue and call a callback function for MAC_RX interrupt.

void nmac_rx_handler()

Parameters

None.

Return Values

None

(5) Receiver module On/Off

To save power, it is possible to turn off the RF chip. The nmac_rx_on() function activates the receiver of RF chip while the nmac_rx_off() function deactivates the receiver of RF chip.

void nmac_rx_on(void);

Parameters

None

Return Values

None

void nmac_rx_off(void);

Parameters

None

Return Values

None

(6) Testing (limit RX range)

Nano OS provides a few functions for testing. In the development phase, all sensor nodes are placed within one hop range, in which all nodes are able to communicate among each

other directly. In such an environment, it is difficult to make sure multi hop routing function is performed properly. Thus, this problem can be handled by limiting the communication range.

void nmac_set_rx_range(UINT16 min_addr, UINT16 max_addr); **Parameters** min_addr : minimum node address that allows to receive (for testing) max_addr : maximum node address that allows to receive (for testing) **ReturnValues** None

When a frame has been received, the following function checks the range validation.

static BOOL nmac_rx_permit(UINT16 src_addr); **Parameters** src_addr : previous hop node address **ReturnValues** TRUE if a given node address is within allowable range, and FALSE otherwise

3.2.2. Routing

3.2.2.1. RENO Routing

(1) Data structure

The RENO protocol is a reactive and on-demand routing algorithm. The packet structure of this protocol is shown in Fig. 15.

Figure 15. Routing Message Structure

```
typedef struct reno_packet
{
    UINT8 msg_type;          // message type
    UINT8 msg_hop_limit;     // maximum number of hops that a message can be
delivered
    UINT8 msg_hop_count;     // number of hops that a message passes through
    UINT8 reserved;          // reserved.
    UINT8 dest_id;           // destination ID
    UINT8 src_id;            // source ID
    UINT8 payload_size;      // length of palyload
void   *payload_ptr;         // pointer of payload
} RENO_PACKET;
```

The msg_type variable describes the type of a message. There are four types in a message; DATA, RREQ(Route Request), RREP(Route Reply) and RERR(Route Error). The DATA message contains real data in application layer. In order to send the DATA message, a routing path to the destination needs to be determined in RENO algorithm. The RREQ message is the one that sets up a routing path in ad-hoc networks. If the RREQ message is broadcast, the neighbor nodes get the message and rebroadcast it. When the destination node receives the RREQ message, it notifies this fact to the sender node by sending a RREP message. Any node in routing path can send RERR to a source node when DATA message trasmission has been failed.

When a DATA message itself has arrived, the node stores the message in the RENO RX queue. The source ID, data length, and data are stored in the following circular queue.

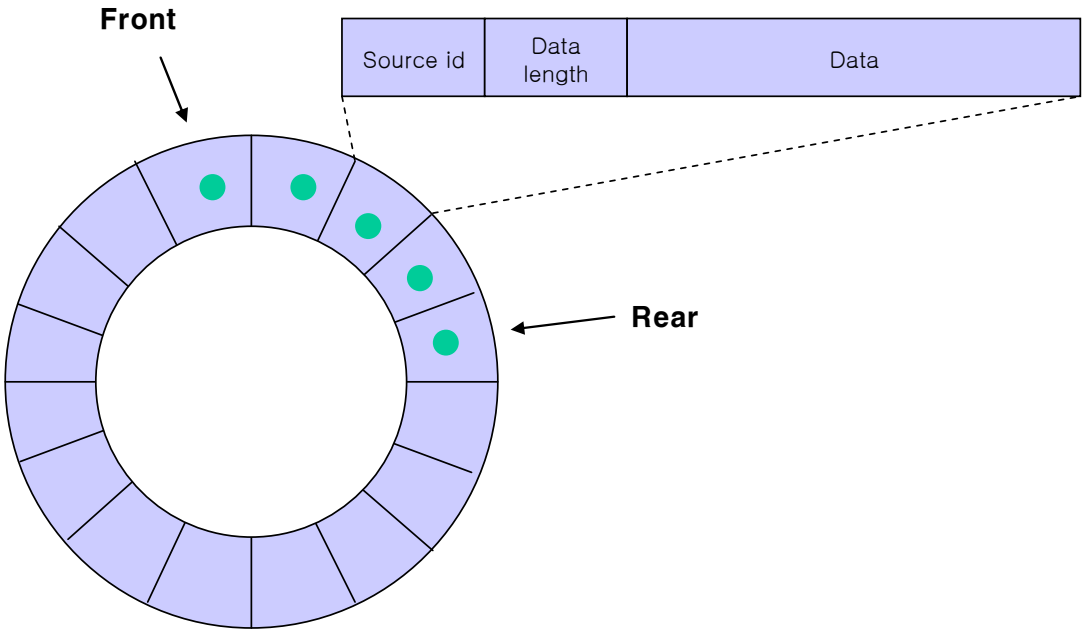

Figure 16. Data Queue on Receiver Side

```
typedef struct reno_recv_data
{
	UINT8 src_id;
	UINT8 data_length;
	UINT8 data[RENO_MAX_PACKET_PAYLOAD_SIZE];
}RENO_RECV_DATA;
```

```
struct reno_queue
{
	UINT8 front, rear;
	RENO_RECV_DATA item[RENO_QUEUE_LENGTH];
} RENO_QUEUE;
```

MAC frame is handled in Session Information base(SIB) structure.

```
typedef struct reno_session_information_base
{
	UINT8 prev_hop_id;		// previous node ID that send the packet
	UINT8 next_hop_id;		// next node ID to receive the packet
	RENO_PACKET packet;	// routing packet
} RENO_SIB;
```

Each node takes the responsibility for relaying packets of each neighbor nodes properly. Nodes have the following routing table for doing this.

```
typedef struct reno_route_entry
{
    UINT8    dest_id;              // destination node ID
    UINT16 dest_seq_num;       // to manage the last received data
    UINT8    next_hop_id;        //    next hop node ID to send data to
    BOOL     is_valid;            // the entry is valid?
    UINT8 link_fail_count;      // for RERR
} RENO_ROUTE_ENTRY;
```

Information at network layer (e.g. routing table, node ID and sequence number) is stored in Network Information Base (NIB) structure. The NIB structure is shown below. Each node has only one NIB.

```
struct reno_network_informaion_base
{
    UINT8    id;                  // node ID
    UINT16 seq_num;         // routing infomation message sequence number

    // routing table
    RENO_ROUTE_ENTRY route_table[MAX_ROUTE_ENTRY]; // routing table
    UINT8 route_table_index;              // index of last added entry

    // received data; not used
    UINT8 recv_data[RENO_MAX_PACKET_PAYLOAD_SIZE];
    UINT8 recv_data_length;
} RENO_NIB;
```

(2) Module initialization

To use routing module, the reno_init() function must be called. This function calls the nos_rf_init() function to initialize MAC layer, and initializes NIB, the receiver queue and so on.

```
void reno_init(void);
```

Parameters

None

ReturnValues
None

(3) Receiving packet

To receive data from the network layer, the application layer must call the reno_recv_from_nwk() function. It should specify the source node ID and a buffer pointer for received data as parameters.

BOOL reno_recv_from_nwk(UINT8* src_id, UINT8* data_length, void* data);
Parameters
src_id : source node ID
data_length : data size
data : pointer of data
ReturnValues
TRUE if succeeded, FALSE if failed

(4) Sending packet

To send data to the network layer, the application layer must call the reno_send_to_nwk() function. It fills the Session Information Base structure and calls the reno_handler() function.

void reno_send_to_nwk(UINT8 dest_id, UINT8 data_length, void *data);
Parameters
dest_id : destination node ID
data_length : data size
data : pointer of data
ReturnValues
None

(5) Receiving packet from MAC layer

The reno_recv_from_mac() function is used for relaying data from MAC layer to network layer. It writes a frame into Session Information Base structure and calls reno_hanler() function.

void reno_recv_from_mac(void);
Parameters
None
ReturnValues

None

(6) Packet sending to MAC layer

The reno_send_to_mac() function relays data from network layer to MAC layer. It increases the hop counter by 1, and then unicasts or broadcasts depending on whether broadcasting is needed(RREQ).

BOOL reno_send_to_mac(RENO_SIB *sib_ptr, BOOL is_broadcast);

Parameters

sib_ptr : Session Information Base to send

is_broadcast : broadcasting or not?

ReturnValues

TRUE if succeeded, FALSE if failed

(7) Packet handling

When packets are received or supposed to be sent, the reno_handler() function is called.

void reno_handler(RENO_SIB *sib_ptr);

Parameters

sib_ptr : Session Information Base to process

ReturnValues

None

The reno_handler() function checks the destination ID. If the destination ID is the node itself, it stores the received message in the receiver queue, inserts/deletes a routing enrty, or sends routing information. If the destination ID is not the node itself, it searches the destination ID through routing table. Depending on the result (success or fail), it calls the corresponding function. If the destination ID is found during searching through the routing table (in case of success), the reno_route_lookup_success() is called.

void reno_route_lookup_success(RENO_SIB* sib_ptr);

Parameters

sib_ptr : Session Information Base to process

ReturnValues

None

1. If the message type is DATA and the source node ID is not the node itself, it sends the packet to the next hop node (The next hop node is identified by sending a RREQ packet). If the packet sending is failed, it deletes the corresponding routing entry for the next hop. If the source node is not the node itself, it also sends a RERR message to previous hop to notify the source node of the failure of routing. If it even fails to send the RERR, it deletes the routing entry for the previous hop.

2. If the message type is RREQ, it inserts the entry information into the routing table, and broadcasts the packets if the result is TRUE.

3. If the message type is RREP, it inserts the entry information to the routing table.

4. If the message type is RERR, it simply removes the entry information from the routing table.

If the destination ID is not found in the routing search (in case of failure), the reno_route_lookup_fail() function is called.

void reno_route_lookup_fail(RENO_SIB* sib_ptr); **Parameters** sib_ptr : Session Information Base to process **ReturnValues** None

1. If the message type is DATA, it tries to send data to the next hop by calling the reno_ send_to_mac () function. If it fails, it deletes the routing entry from the routing table for the next hop and calls the reno_route_lookup_fail() function that is a handler invoked when no routing entry is found in the routing table.

2. If the message type is RREQ, it inserts the entry information into the routing table, and broadcasts the packet if the result is TRUE.

3. If the message type is RREP, it inserts the entry information into the routing table and sends it to the next hop.

4. If the message type is RERR, it simply deletes the entry information from the routing table and sends it to the next hop.

(8) Creating and sending packets

The reno_send_rreq(), reno_send_rrep() and reno_send_rerr() functions are used to send RREQ, RREP, and RERR packets, respectively.

The reno_send_rreq() function creates and sends an RREQ message. It periodically broadcasts the RREQ request and looks up the routing table after a given time. It returns if it

obtains the routing information or the number of tries is exceeded.

The reno_send_rrep() function creates and sends an RREP message for an RREQ message, while the reno_send_rerr() function creates and sends an RERR message. In both cases, after being writted into the Session Information Base, the message is sent to the requester node by calling the reno_send_to_mac() function.

UINT8 reno_send_rreq(UINT8 dest_id);
Parameters
dest_id : destination node ID
ReturnValues
routing table entry number to the destination if succeeded
ROUTE_TABLE_LOOKUP_FAILURE if failed

BOOL reno_send_rrep(UINT8 dest_id);
Parameters
dest_id : destination ID
ReturnValues
TRUE if succeeded FALSE if failed

BOOL reno_send_rerr(UINT8 dest_id, UINT8 prev_node_id, UINT8 err_node);
Parameters
dest_id : destination node ID
prev_node_id : previous node ID
err_node : node ID where error occured
ReturnValues
TRUE if succeeded FALSE if failed

(9) Routing table management

To manage the routing table, the following functions are provided.

The reno_scan_route_table() function searches such a routing entry that the given node ID is the destination, and returns the entry number. It searches routing entries from the last entry inserted into the routing table, and circulates after the first entry. It returns if it finds the routing entry for the given dest_id or circulates the routing table.

UINT8 reno_scan_route_table(UINT8 dest_id);
Parameters

dest_id : destination node ID

ReturnValues

routing entry number if dest_id is found in the search

ROUTE_TABLE_LOOKUP_FAILURE if the search is failed

The reno_insert_route_entry() function adds a new entry to the routing table. If the source ID is the node itself, it returns FALSE since it comes back to the node. Otherwise, it checks if the routing path from the source node is already in the routing table. If it is not in the routing table, it searches an empty routing entry, registers the routing information, and returns TRUE. If it is in the routing table, it compares the sequence numbers of the routing entries with the one of the packet. If the sequence numbers are same and the packet is supposed to be sent from the same node, it is recognized as retransmission. In this case, it does not update the routing table, but returns TRUE so that the RREQ message can be sent. If the sequence numbers are same but the packet is supposed to be sent from the different nodes, and further the sequence number is less than 2, it is regarded as an old packet and thus returns FALSE. Elsewhere, it updates the corresponding entry information in the routing table.

BOOL reno_insert_route_entry(RENO_SIB* sib_ptr);

Parameters

sib_ptr : Session Information Base to be inserted in the routing table

ReturnValues

TRUE if a new entry is added or the previous entry is updated

FALSE otherwise.

void reno_delete_dest_route_entry(UINT8 dest_id);

Parameters

dest_id : destination node ID to be deleted in the routing table

ReturnValues

None

The reno_delete_dest_route_entry() function deletes the routing entry of which node ID is the next_hop_id of the entry.

void reno_delete_next_route_entry(UINT8 next_hop_id);

Parameters

next_hop_id : Next hop ID to be deleted in the routing table

ReturnValues

None

(10) Debugging

The print_route_table() function prints routing table information on the UART terminal. The print_nib(), print_sib() and print_payload() functions print Network Information Base, Session Information Base and the content of payload on the UART terminal, respectively. All these simulation and debugging function can be used only when the kernel is set to be in the debugging mode.

void print_route_table(void); **Parameters** None **ReturnValues** None

void print_nib(void); **Parameters** None **ReturnValues** None

void print_sib(RENO_SIB *sib_ptr); **Parameters** sib_ptr : pointer of Session Information Base **ReturnValues** None

void print_payload(INT8 payload_unit, UINT8 payload_size, void *payload); **Parameters** payload_unit : payload unit (UINT16 : 16, else decimal: 8, String : s) payload_size : payload size payload : pointer of payload **ReturnValues** None

3.3. Nano Hardware Abstract Layer (nHAL)

The nano Hardware Abstract Layer(nHAL) makes operating systems independent by abstracting hardware details. It varies from hardware to hardware.

3.3.1. MCU dependent modules

(1) Critical Section

For critical sections, Nano OS provides two macros, NOS_ENTER_CRITICAL_SECTION() and NOS_EXIT_CRITICAL_SECTION().

#define NOS_ENTER_CRITICAL_SECTION() **Parameters** None

The NOS_ENTER_CRITICAL_SECTION() stores SREG (Status Register) into the stack and turns off all interrupt activities (context switching is disabled).

#define NOS_EXIT_CRITICAL_SECTION() **Parameters** None

The NOS_EXIT_CRITICAL_SECTION() restores SREG (Status Register) from the stack and enables interrupts (context switching is allowed).

(2) Delay Function

Nano OS provides two delay functions that are implemented by “busy waiting” operation.

void nos_delay_us(UINT16 timeout_usec); **Parameters** timeout_usec : delay time in microsecond **Return Values** None

The nos_delay_us() function is implemented by the following NOS_NOP() macro function. This NOS_NOP() macro performs no tasks, but it consumes a few CPU clock cycles.

#define NOS_NOP() **Parameters**

None

The nos_delay_ms() function delays the time in milli-second and implemented by the nos_delay_us() function.

void nos_delay_ms(UINT16 timeout_msec);

Parameters

timeout_ msec: delay time in milli-second.

Return Values

None

3.3.2. MCU internal peripherals

3.3.2.1. ADC

Sensor data is usually analog signal, which must be converted into the digital data by using ADC (Analog to Digital Converter). Atmega128 provides 8 ADC channels, each of which can be used to connect to different types of devices.

(1) Module initialization

For initialization, the nos_adc_init() function is used. It activates ADC channels.

void nos_adc_init(void); **Parameters** None **Return Values** None

(2) Analog to digital conversion

The nos_adc_convert() function is invoked by setting related registers, and returns the converted value as soon as the conversion is completed. The returned value is 10-bit data.

UINT16 nos_adc_convert(void); **Parameters** None **Return Values** Converted 10bit data

(3) ADC channel selection

An adc channel must be selected before the nos_adc_convert() function is called. This is done by the nos_adc_select_channel() function.

void nos_adc_select_channel(UINT8 channel);; **Parameters** channel : ADC channel to use **Return Values** None

3.3.2.2. UART

The UART is used for standard I/O interface or debugging purpose in embedded systems. Nano OS supports serial communication by UART.

(1) Module initialization

The nos_uart_init() function sets UART related registers for initialization. Note that the number of bits for communication, parity bit, and flow control parameters are already defined in the source code. It is nessary that you should modify the source code when you change the parameter values.

void nos_uart_init(void);

Parameters

None

Return Values

None.

(2) Sending data

The uart_putc() is the basic function to send 1 byte data. To send a string, the uart_puts() is implemented with the uart_putc() function by calling it as many as the number of bytes in the string. To send signed or unsigned values, the nos_uart_puti() and nos_uart_putu() functions are provided.

void nos_uart_putc(UINT8 port_num, INT8 byte);

Parameters

port_num : UART port number

byte : 1 byte data to send

Return Values

None.

void nos_uart_puts(UINT8 port_num, INT8 *str);

Parameters

port_num : UART port number

str : string to send

Return Values

None.

void nos_uart_puti(UINT8 port_num, INT16 val);

Parameters

port_num : UART port number
val : signed integer value

Return Values

None.

void nos_uart_putu(UINT8 port_num, UINT16 val);

Parameters

port_num : UART port number
val : unsigned integer value

Return Values

None.

(3) Receiving data

The nos_uart_gets() function receives one byte data from the UART port. This is implemented by an interrupt routine, which receives data until it encounters the 'carriage return'(_CR).

void nos_uart_gets(UINT8 port_num, INT8* str, UINT8 str_len);

Parameters

port_num : UART port number
str : buffer pointer of data to receive
str_len : maximum size of the data buffer

Return Values

None.

The UART receiving is perfomed by the uart_recv_handler() in UART0 RX and UART1 RX interrupt handler.

void nos_uart_getc_callback(UINT8 port_num, void (*func)(UINT8));

Parameters

port_num : UART port number

Return Values

None.

(4) Formatted output

The nos_uart_prinf() function prints a formatted string through the UART communication like the standard C library function.

#define nos_uart_printf(format, ...)

Parameters

format : formatted string

… : output parameters

Return Values

None.

3.3.2.3. SPI

SPI is a serial pheriperal interface for high speed communication between MCU and pheriperal devices. This interface uses 4 pins and supports full-duplex mode.

(1) SPI initialization

The SPI_INIT() function initializes the SPI interface by setting SPI related registers. According to the devices communicating with each other via SPI, some parameters such as working mode and clock mode can be set.

void SPI_INIT(is_master, clk, spi_mode);

Parameters

is_master : is spi mater or slave?

clk : spi clock selection

spi_mode : spi data mode selection

Return Values

None.

Macro	Description
SPI_MASTER	SPI works as a master
SPI_SLAVE	SPI works as a slave

If SPI works as a master, we must give clocks to slave devices. The clock is referenced by the system clock (or asynchronous clock) and can be set to one of the followings.

Macro	Description
SPI_CLK_DIV_2	1/2 of the reference clock
SPI_CLK_DIV_4	1/4 of the reference clock
SPI_CLK_DIV_8	1/8 of the reference clock
SPI_CLK_DIV_16	1/16 of the reference clock
SPI_CLK_DIV_32	1/32 of the reference clock
SPI_CLK_DIV_64	1/64 of the reference clock
SPI_CLK_DIV_128	1/128 of the reference clock

For SPI to recognize the received data, the polarity and phase of the clock signal must be defined and can be set to one of the followings.

Macro	Description
SPI_DATA_MODE0	sampling at leading edge(rising), setup at trailing edge(falling)
SPI_DATA_MODE1	setup at leading edge(rising), sampling at trailing edge(falling)
SPI_DATA_MODE2	sampling at leading edge(falling), setup at trailing edge(rising)
SPI_DATA_MODE3	setup at leading edge(falling), sampling at trailing edge(rising)

To communicate with CC2420, MCU is set to be a master, and the speed of SPI is 1/2 of the system clock with SPI data mode 0.

(1) SPI slave device activation/deactivation

When MCU is a SPI slave device, the SPI_ENABLE() or SPI_DISABLE() functions are necessary to communicate with slave devices. The SPI_ENABLE() function clears /SS pin (Active low, connected with CSn pin of CC2420) of SPI, which activates the slave device. The SPI_DISABLE() function sets /SS pin of SPI, which deactivates the slave device.

void SPI_ENABLE();

Parameters

None.

Return Values

None.

void SPI_DISABLE();

Parameters

None.

Return Values

None.

(2) Sending data

There are four available functions for data communication. The SPI_TX() function sends one byte of data via SPI interface. To send two bytes of data, the SPI_TX_WORD() and SPI_TX_WORD_LE() function are defined. The difference of the two is that the former sends the upper byte first while the latter the lower byte first. The SPI_TX_MANY() function is used to send more than 3 bytes of data.

void SPI_TX(UINT8 x);

Parameters
x : data to send
Return Values
None.

void SPI_TX_WORD_LE(UINT16 x);
Parameters
x : data to send
Return Values
None.

void SPI_TX_WORD(UINT16 x);
Parameters
x : data to send
Return Values
None.

void SPI_TX_MANY(BYTE *p, UINT8 c);
Parameters
p : data to send
c : data length
Return Values
None.

To send data through SPI interface, previous data transmission must be completed. Thus, if it was not completed yet, it is necessary to wait. The SPI_WAIT() macro will do the function.

#define SPI_WAIT()
Parameters
None.

(3) Receiving data

Likewise, there are four functions for receiving data. The SPI_RX () function receives one byte of data through SPI interface. To receive two bytes of data, the SPI_RX_WORD() and SPI_RX_WORD_LE() function are defined. The difference between the two is that the former receives the upper byte first, while the latter receives the lower byte first. The SPI_RX_MANY() function is used to receive more than 3 bytes of data.

BYTE SPI_RX(void);

Parameters

None.

Return Values

1 byte of data received

UINT16 SPI_RX_WORD_LE(void);

Parameters

None.

Return Values

two bytes of data

UINT16 SPI_RX_WORD(void);

Parameters

None.

Return Values

two bytes of data

void SPI_RX_MANY(BYTE *p, UINT8 c);

Parameters

p : variable to be stored

c : data length

Return Values

None.

The SPI_RX_GARBAGE() macro is used to remove the received data instead of using it. It is similar to the SPI_RX(), but simply returns without duplicating the received data into a variable.

void SPI_RX_GARBAGE(void);

Parameters

None.

Return Values

None.

3.3.2.4. EEPROM

Nano OS provides interfaces of writing data into EEPROM if the target platform has an EEPROM device. EEPROM can be used for storing configuration data, which is rarely changed through the lifetime of sensor node.

(1) Writing data

UINT16 nos_eeprom_write(UINT16 addr, const BYTE *buf, UINT16 len) **Parameters** addr : EEPROM address to write data to buf : data pointer len : data length **Return Values** actual length of data written

The nos_eeprom_write() function writes data into the designated address. The return value specifies the actual data length. This function is implemented, using nos_eeprom_write_byte() function.

UINT16 nos_eeprom_write_byte(UINT16 addr, BYTE data) **Parameters** addr : EEPROM address to write data : data to write **Return Values** None

(2) Reading data

UINT16 nos_eeprom_read(UINT16 addr, BYTE *buf, UINT16 len) **Parameters** addr : EEPROM address to read from buf : buffer pointer for data read len : length of data to read **Return Values** Actual length of data read

The nos_eeprom_read() function reads data from the designated address by the number of len bytes. The return value is the actual length of data read. The nos_eeprom_read() function

is implemented using the nos_eeprom_read_byte() function.

UINT16 nos_eeprom_read_byte(UINT16 addr, BYTE *data) **Parameters** addr : EEPROM address to read from data : buffer pointer for data to read **Return Values** None

3.3.3. Sensor

Nano OS provides 8 types of sensors. You need to select the type of sensor to turn on/off the sensor. The type of a sensor has a corresponding channel. In other words, it means that the channel can be used as the type of a sensor.

3.3.3.1. Gas sensor

(1) Sensor initialization

Before using gas sensor, the nos_gas_init() must be used.

void nos_gas_init(); **Parameters** None **Return Values** None

(2) Get sensor data

To get gas data, the nos_ gas_get_info() function is used. It returns the value converted from ADC.

UINT16 nos_gas_get_info(); **Parameters** None **Return Values** None

3.3.3.2. Humidity sensor

(1) Sensor initialization

Before using humidity sensor, the nos_hum_init() must be invoked. This function initializes timer/counter for measuring humidity, and pins connected to the humidity sensor.

void nos_hum_init(); **Parameters** None **Return Values** None

(2) Get sensor data

The humidity can be obtained from the frequency of ADC values of the humidity sensor. Thus, after getting humidity data, it must be converted into humidity values by using nos_hum_get_info() function.

UINT16 nos_hum_get_info(); **Parameters** None. **Return Values** Humidity value for a particular humidity frequency

The nos_hum_get_info() returns a humidity value, referring to a HumTable that shows humidity values vs humidity frequencies. The humidity frequency can be obtained by the nos_hum_get_freq() function. The get nos_hum_get_info() function returns the humidity value for one second by using timer/counter.

UINT16 nos_hum_get_freq(); **Parameters** None **Return Values** None

3.3.3.3. Light sensor

(1) Sensor initialization

Before using light sensor, the nos_light_init() must be invoked.

void nos_light_init(); **Parameters** None **Return Values** None

(2) Get sensor data

To get light data, the nos_light_get_info() function is used. It returns the value converted from ADC.

UINT16 nos_light_get_info(); **Parameters** None **Return Values** None

3.3.3.4. Temperature sensor

(1) Sensor initialization

Before using temperature sensor, the nos_temp_init() must be invoked.

void nos_temp_init(void); **Parameters** None **Return Values** None

(2) Get sensor data

To get gas data, the nos_temp_get_info() function is used. It returns the value converted from ADC.

UINT16 nos_temp_get_info(void); **Parameters** None **Return Values** None

3.3.3.5. Infrared Sensor

(1) Sensor initialization

Before using infrared sensor, the nos_pir_init() must be used. Since the infrared sensor tells whether an object is sensed by interrupt, the nos_pir_init() function initializes the pin that is connected to the corresponding interrupt.

void nos_pir_init(void); **Parameters** None **Return Values** None

(2) Sense an object

Sensing an object is an asynchronous event. Thus, the callback function must be registered so that it can be called when the event occurs. In the current implementation, the external interrupt 7 is connected to the PIR sensor. The nos_pir_callback() function defines a callback function to process something when interrupted.

ISR(INT7_vect);

void nos_pir_callback(void (*func)(void)); **Parameters** func : callback function to be invoked **Return Values** None

3.3.3.6. Ultrasonic sensor

(1) Sensor initialization

Before using ultrasonic sensor, the nos_us_init() must be invoked. The function initializes pins, interrupt, and timer for ultrasonic sensor.

void nos_us_init(void); **Parameters** None **Return Values** None

(1) Start sensor signal On/Off

In ultrasonic sensor, the NOS_US_TX_ON() function must be called to measure round trip time of ultrasonic signals. After the signal is transmitted back to the sensor, the NOS_US_TX_OFF() function must be called to turn off the sensor.

#define NOS_US_TX_ON(); **Parameters** None **Return Values** None

#define NOS_US_TX_OFF(); **Parameters** None **Return Values** None

(2) Measure round trip time

A timer is used to measure the round trip time of the ultrasonic signal. This is done by calling the nos_us_get_info() function for timer initialization, and the nos_us_get_info() function is called to measure the round trip time when the feedback signal is sensed.

void nos_us_get_info(void); **Parameters** None **Return Values**

None

UINT16 nos_us_get_info(void); **Parameters** None **Return Values** Measured counter value.

(3) Get sensor data

In the ultrasonic sensor, the distance is measured by sending a ultrasonic signal by using the nos_us_trigger() function and detecting the feedback signal by interrupt when the signal is transmitted.

void nos_us_trigger(void (*func)(void *)); **Parameters** func : callback function to be used for measuring the distance **Return Values** None

The ultrasonic sensor is connected to the external interrupt 7.

ISR(INT7_vect)

In the interrupt service routine, the nos_us_callback() function defines a callback function that calculates the distance. It reads the timer value by calling the nos_us_get_info() function, turning off the starting signal, and calculates the distance from an object by ultrasonic speed and timer value.

void nos_us_callback(void (*func)(void)); **Parameters** func : callback function to be invoked **Return Values** None

3.3.4. Actuator

3.3.4.1. Module initialization

To use the actuator, the following NOS_ACTUATOR_INIT() function must be called.

#define NOS_ACTUATOR_INIT(act_id); **Parameters** act_id : actuator number **Return Values** None.

3.3.4.2. Actuator control

Sensor nodes can be connected to various kinds of actuators. Like LEDs, they are connected to the general I/O ports in MCU. To control the actuators, the NOS_ACTUATOR_ON(); function and NOS_ACTUATOR_OFF() macro are provided.

#define NOS_ACTUATOR_ON(act_id); **Parameters** act_id : actuator number **Return Values** None.

#define NOS_ACTUATOR_OFF(act_id); **Parameters** act_id : actuator number **Return Values** None.

3.3.5. RF communication

3.3.5.1. RF chip driver (CC2420)

CC2420 is a RF chip from Chipcon. Nano OS provides CC2420 driver for controlling, initializing, channel selection, status check, etc.

(1) Module initialization

All RF related functions are called after the cc2420_init() function is called. The function performs the pin init, SPI init, interrupt init, and turning on crystal oscillator, and waits until the crystal oscillator is stable.

void cc2420_init(void) **Parameters** None. **Return Values** None.

(2) RF channel configuration

The channel number, PAN address and MAC Short Address are required for RF communication,. A RF channel can be selected as one of 16 channels that ranges from 11 to 26. The cc2420_channel_init() function is used to select a RF communication channel.

void cc2420_channel_init(UINT8 channel) **Parameters** channel : communication channel (11 ~ 26). **Return Values** None.

(3) Tranceiver activation/deactivation

The CC2420 tranceiver can be turned on/off to reduce the power consumption. The CC2420_SWITCH_ON() and CC2420_SWITCH_OFF() macro is used respectively.

#define CC2420_SWITCH_ON() **Parameters** None **Return Values** None

#define CC2420_SWITCH_OFF() **Parameters** None **Return Values** None

(4) Auto-ACK activation/deactivation

The CC2420 tranceiver has auto-ack functionality supported by hardware. The CC2420_AUTOACK_REP_ON() and The CC2420_AUTOACK_REP_OFF() macro is used to enable or disable this functionality respectively.

#define CC2420_AUTOACK_REP_ON() **Parameters** None **Return Values** None

#define CC2420_AUTOACK_REP_OFF() **Parameters** None **Return Values** None

(5) CC2420 registers reading/writing

The CC2420 has 3 kinds of registers : command strobe register, normal register, and FIFO register. To use those registers, you need to specify the address of the register. The CC2420_TX_ADDR() macro function specifies the address.

#define CC2420_TX_ADDR(a) **Parameters** a : register address

The CC2420_TX_ADDR() macro, like SPI_TX(), sends one byte of address data via SPI, and waits until the transmission is completed and exits.

The CC2420_RX_ADDR() is a macro function for reading address data.

#define CC2420_RX_ADDR(a) **Parameters** a : register address

The CC2420_RX_ADDR() macro is same as CC2420_TX_ADDR() macro, but reading or writing mode depends on the value of sixth bit of the CC2420 address. If this bit is one, it goes into reading mode.

(6) CC2420 command strobe registers

The command strobe register is a sort of control commands that controls CC2420 by specifying the address of the register. To access the command strobe register of CC2420, the CC2420_STROBE() macro is used.

#define CC2420_STROBE(s) **Parameters** s : command strobe

The CC2420_STROBE() macro simply sends a command strobe to CC2420 through SPI interface after selecting chip, which makes CC2420 do a particular work. To write data into a register, you have only to use the CC2420_SETREG() macro.

#define CC2420_SETREG(a, v) **Parameters** a : register address v : register value

#define CC2420_GETREG(a, v) **Parameters** a : register address v : register value

(7) Get CC2420 status

CC2420 sends the status bytes (the status of the chip) to MUC through SPI interface while receiving data. To read the status bytes, the CC2420_UPD_STATUS() macro is provided. The CC2420_UPD_STATUS() macro reads status bytes after SNOP command strobe is sent.

#define CC2420_UPD_STATUS(s)

Parameters
s : variable for storing the status of CC2420

(8) CC2420 FIFO reading/writing

CC2420 has two FIFOs, TXFIFO and RXFIFO. Since data is strored in RAM region in CC2420, it can be accessed by memory operation. But they are not distinctive in that the data read is automatically removed in FIFO. To write data into TXFIFO in CC2420, SPI_WRITE_FIFO() macro is used.

#define CC2420_WRITE_FIFO(p, c)
Parameters
p : pointer to the byte array to write to FIFO
c : the number of bytes to write

To read multiple bytes of data, the CC2420_READ_FIFO() and CC2420_READ_FIFO_NO_WAIT() can be used. The CC2420_READ_FIFO() and CC2420_READ_FIFO_NO_WAIT() looks similar, but the former checks the interrupt pin whenever it reads one byte of data, while the latter does not.

#define CC2420_READ_FIFO(p,c)
Parameters
p : pointer to the byte array to read
c : the number of bytes to write

#define CC2420_READ_FIFO_NOWAIT(p, c)
Parameters
p : pointer to the byte array to read
c : the number of bytes to write

To read only 1 byte of data in FIFO, the CC2420_READ_FIFO_BYTE() is used. To remove the stored data, the CC2420_READ_FIFO_GARBAGE() is used.

#define CC2420_READ_FIFO_BYTE(b)
Parameters
b : single data byte read from FIFO

#define CC2420_READ_FIFO_GARBAGE(c)

Parameters

c : the number of bytes to take away

(9) CC2420 RAM reading/writing

To write data into RAM in CC2420, the CC2420_WRITE_RAM() and CC2420_WRITE_RAM_LE() macros are used. The former sends the upper bytes first, while the latter sends the lower bytes first.

#define CC2420_WRITE_RAM(p, a, c, n)

Parameters

p : pointer to the variable to be written

a : the CC2420 RAM address

c : the number of bytes to write

n : counter variable which is used in for/while loops

#define CC2420_WRITE_RAM_LE(p, a, c, n)

Parameters

p : pointer to the variable to be written

a : the CC2420 RAM address

c : the number of bytes to write

n : counter variable which is used in for/while loops

To read data from RAM in CC2420, the CC2420_READ_RAM() and CC2420_READ_RAM_LE() macros are used. The former receives the upper bytes first, while the latter receives the lower bytes first.

#define CC2420_READ_RAM(p, a, c, n)

Parameters

p : pointer to the variable to be read

a : the CC2420 RAM address

c : the number of bytes to write

n : counter variable which is used in for/while loops

#define CC2420_READ_RAM_LE(p, a, c, n)

Parameters

p : pointer to the variable to be read
a : the CC2420 RAM address
c : the number of bytes to write
n : counter variable which is used in for/while loops

(10) CC2420 reset

The CC2420_RESET() function resets the MAIN register, resetting CC2420.

#define CC2420_RESET()

Parameters

None.

3.3.5.2. RF pin configuration

Each platform has one MCU and one RF chip. They communicate with each other via general port interface as well as Serial Peripheral Interface. RF pins interfacing to MCU are configured and initialized. (now, just for CC2420)

(1) Set, clear or check the status MACRO

Besides SPI-related pins, there are several pins between MCU and CC2420. They are pins for FIFO, FIFOP, RESET, VREG, SFD, and CCA.

```
#define RF_RESET_SET()
#define RF_RESET_CLR()
#define RF_VREG_SET()
#define RF_VREG_CLR()
#define RF_RESET_IS_SET()
#define RF_VREG_IS_SET()
#define RF_FIFOP_IS_SET()
#define RF_FIFO_IS_SET()
#define RF_SFD_IS_SET()
#define RF_CCA_IS_SET()
```

(2) RF initialize

nos_rf_init() initializes CC2420 and configures pins between RF and MCU.

```
void nos_rf_init(void);
Parameters
  None.
Return Values
  None.
Description
  Initialize RF.
```

(3) RF interrupt handler

The nos_rf_callback() defines a callback function that handles a received frame when FIFOP interrupt occurs(a frame received in CC2420 RXFIFO). The INT0_vect is FIFOP interrupt handler

```
void nos_rf_callback(void);
Parameters
  void (*func)(void)
Return Values
```

None.

Description

Register ISR call back function

ISR(INT0_vect)

3.3.6. Misc

3.3.6.1. LED

To use LEDs, the NOS_LED_INIT() function should be called for initialization. The _ function initializes each port pins connected to the LEDs and turns off them.

void NOS_LED_INIT(); **Parameters** None **Return Values** None

LEDs are connected to the general I/O port interface and can be turned on/off by writing zero or one into each port of the corresponding registers. The following functions are provided for controlling LED devices.

void NOS_LED_ON(n); **Parameters** n : LED number **Return Values** None. **Description** Turn on n-th LED

void NOS_LED_OFF(n); **Parameters** n : LED number **Return Values** None. **Description** Turn off n-th LED

void NOS_LED_TOGGLE(n); **Parameters** n : LED number **Return Values** None. **Description**

Toggle n-th LED

3.3.6.2. Battery power status

The battery sensor requires no initialization

(1) Get battery data

To get a battery data, the nos_bat_get_info() is provided.

UINT16 nos_bat_get_info(); **Parameters** None **Return Values** None

The nos_bat_get_info() function uses ADC. The value converted from ADC is returned after some calculations.

——————————————————— END ———————————————————