Volume 3
Nut/OS Threads, Events and Timers

Version 1.0

Contents

1 Introduction

Nut/OS internals.

This document discusses three basic mechanisms of Nut/OS:

- Threads
- Events
- Timers

2 Threads

Nut/OS implements cooperative multithreading.

C programmers usually start to write single-threaded programs, which sequentially execute on a single code path. As the name implies, a multithreaded program concurrently executes several threads.

Of course, a single CPU can't run more than a single thread at a time. But Nut/OS is able to switch the CPU from the context of one thread to the context of another thread. This gives the appearance of simultanously running threads.

Each thread has a priority which is used to determine how urgent it is. This priority ranges from 0 to 255, with the lowest value indicating the most urgent. The system works on the principle that the most urgent thread always runs if it is not waiting for any event.

All threads are executing in the same address space using the same hardware resources, which significantly reduces context switching overhead. Therefore it is important to stop them from causing each other problems. This is particularly an issue where two or more threads need to share a resources like memory locations or peripheral devices.

Nut/OS implements cooperative multithreading. That means, that threads are not bound to a fixed timeslice. Unless they are waiting for specific event or explicitely yielding the CPU, they can rely on not being stopped unexpectedly. One exception to this is if a CPU interrupt signal arrives.

In opposite to preemptive multithreading, coorperative multithreading simplifies resource sharing and results in faster and smaller code.

Declaring Threads

In Nut/OS thread starting points are defined as functions, which never return:

```
void threadfn(void *arg) __attribute__ ((noreturn));
```

The single void pointer argument can be used to pass specific information to the thread. The include file thread.h contains a macro to simplify the function declaration:

```
THREAD(threadfn, arg)
```

A special function named NutMain() must exist in every Nut/OS application.

```
THREAD(NutMain, arg)
```

This is equivalent to the standard C function main(), which is hidden in Nut/OS.

Starting Threads

NutMain() is the starting point of the main thread and is automatically called by the operating system during initialization. The main thread may then launch other threads by calling NutThreadCreate().

```
NutThreadCreate(u_char *name, void (*fn)(void *),
void *arg, u_short stackSize);
```

The first argument name specifies the symbolic name of this thread. Currently this string doesn't serve any other purposes than giving the thread a human readable name.

The second argument fn is the pointer to the function, which is used as the threads entry point.

The third argument arg is the void pointer being passed to this function. This may be set to null if not used.

The last argument, the threads stack size, which will be further explained now.

Determing the Stack Size

As stated previously, threads share all hardware resources like CPU registers and memory space. While switching from one thread to another, Nut/OS saves all CPU registers of the currently running thread and restores the previously saved register contents of the thread being started. This includes the stack pointer, as each thread needs its own stack.

The memory area used for the stack is allocated by NutThreadCreate(), which is unable to determine how much stack space may be needed by thread. Therefore this size is passed as an argument and must be specified by the caller, actually the programmer. But how can the programmer know?

Stack space is used for two purposes: Register storage during function calls and storage of auto variables. Auto variables are variables, which are locally declared within functions.

The stack space used for register storage is decided by the compiler and is hard to foresee. It depends on the optimization level, the register usage before, after and within the function call.

Nut/OS API functions called by the application may call other functions as well. In addition, interrupt routines are using the stack space of the interrupted thread and need to store all CPU registers.

Putting this all together, it will become clear, that determing the required stack space is at least difficult, if not even impossible because of the asynchronous nature of thread switching. Typically a maximum is estimated and some bytes are added for safety.

Nut/OS allocates 768 bytes of stack for the main thread, which is far more than most applications will use, if they follow certain rules. To save stack space, you should avoid two things. The first is using recursive function calls, unless you can guaranty a maximum nesting level. The second thing to avoid is declaring large arrays or structures within functions. Either declare them global or, to retain reentrency, declare a pointer and allocate the required memory from the heap.

Most application threads will be satisfied with 512 or even 256 bytes of stack. If enough memory is available, you should oversize the stack during development and reduce it later. Call NutHeapAvailable() to determine the number of bytes available.

Advanced users of Nut/OS may inspect the NUTTHREADINFOstructure to track the stack pointer.

Type	Element	Description
NUTTHREADINFO *	td_next	Link to the next thread info structure. This entry is used to create queues.
NUTTHREADINFO *	td_qnxt	Link to the next thread info structure in the list of all threads.
u_char[9]	td_name	Symbolic name of the thread.
u_char	td_state	Current state of the thread. Possible values are: TDS_TERM, not used. TDS_RUNNING, currently running. TDS_READY, ready to run. TDS_SLEEP, waiting for an event.
u_short	td_sp	Stack pointer contents used during context switch.
u_char	td_priority	Priority of the thread.
void *	td_memory	Pointer to the allocated heap memory, which is used for the thread's stack.
void *	td_timer	Points to the info structure of the timer the thread is waiting for. Set to null if the thread is not waiting for any timer.
void *	td_queue	Route entry of the queue where the thread is waiting. Set to null if the thread is ready to run or running.

Table 1: NUTTHREADINFO strucutre

Changing Priority

This priority of a thread can be set from 0 to 255, where the higher the number the lower the priority. Initially threads get a priority of 64 when created. In opposite to many other operating systems, Nut/OS threads can only change their own priority, not the priority of other thread. The declaration of the API function for doing this is:

```
u_char NutThreadSetPriority(u_char level);
```

The only parameter level specifies the new priority. The function returns the priority that had been previously set.

When calling this function, the context is switched to the thread with the highest priority, which is not waiting for an event. This may or may not be the calling thread.

Nut/OS and Nut/Net will create their own threads, which are running at hardcoded priorities.

Symbolic Name	Function	Priority	Stack Space
txi5	Ethernet transmitter	7	640
rxi5	Ethernet receiver	8	640
arpex	ARP table expiration	64	384
tcptmr	TCP state timer	64	384
dhcp	DHCP client	254	512
idle	Idle thread	255	384

Table 2: Nut/OS/Net internal threads

Yielding to Other Threads

Threads usually keep running until they are forced to wait for something to happen elsewhere. The non-preemptive scheduling mechanism used in Nut/OS relies on the individual thread to frequently yield control of the CPU. Otherwise all remaining threads are blocked and the system appears to hang.

Threads implicitly yield control when they have to wait for an event. A typical example is a thread, which receives data through an interface. When this thread calls a data input routine like NutDeviceGetLine(), it will be suspended if no data is available. The thread is marked not-ready-to-run and Nut/OS passes control to the next ready-to-run thread with the highest priority.

However, there are situations where a thread can't expect an event and has to poll, for example, a port bit. Such a thread must explicitly yield CPU control by calling

```
void NutThreadYield(void);
```

The same applies to threads doing lengthy calculations.

After calling NutThreadYield(), you can't make any assumptions about when a thread will be scheduled to run again. If you want to prevent a thread from running until a fixed amount of time has elapsed, you should call

```
void NutSleep(u_long ms);
```

The only argument that this function takes is a minimum number of milliseconds that must elapse before the thread will run again. In that sense NutThreadYield() is just a special case of NutSleep(), indeed NutSleep(0).

The Idle Thread

What happens, if all threads are waiting for an event? During initialization Nut/OS calls

```
NutThreadCreate("idle", NutIdle, 0, 384);
```

When called for the very first thread, this function never returns, but immediately jumps into the specified thread function. The idle function is most simple.

```
THREAD(NutIdle, arg)
{
    NutTimerInit();
    NutThreadCreate("main", NutMain, 0, 768);
    NutThreadSetPriority(255);
    for(;;) {
        NutThreadYield();
    }
}
```

It initalizes Nut/OS timer management as part of the initialization, creates the main application thread, lowers its priority to the minimum and enters an endless loop. Actually the idle thread loses control after setting the priority, because this API function switches to the higher priority main thread, which has just been created.

Note, that the endless loop contains a call to NutThreadYield(). If it wouldn't, the system will hang up as soon as the idle thread gains CPU control. And this is the case when all other threads are waiting for an event.

Terminating Threads

The current version of Nut/OS is not able to terminate a thread. Generally this is no problem, because rather than creating and destroying a thread to perform a temporary task, you can create a task and wait in loop for something to be done. However, the backdraw is that some occupied resources like stack memory can never be released.

Switching Context

Some of you might be interested in how context switching works inside Nut/OS. The magic is done by two routines:

```
void NutThreadEntry(void) __attribute__ ((naked));
NutThreadSwitch(void);
```

Both routines mainly consist of inline assembly statements.

NutThreadSwitch() pushes all 32 CPU registers on the stack and finally stores the stack pointer in the NUTTHREADINFO structure of the thread being stopped. Then it loads the stack pointer from the NUTTHREADINFO structure of the thread to be started and pops all 32 CPU registers from this stack.

Note, that NutThreadSwitch() is a C function. When entered, the program counter has already been pushed on the stack due to the call machine instruction generated by the compiler. The last machine instruction of the C function is a return statement, which will reload the program counter from the stack. But NutThreadSwitch() is different, because it modifies the stack pointer itself. So the function will not return with in a completely different context. We can say, that Nut/OS performs context switching by simply switching stacks.

When a thread is started for the first time, it had pushed something on the stack. You can imagnine, that simply calling NutThreadSwitch() will miserably fail. Therefore NutThreadCreate() prepares the stack in a way, that it looks like the thread had been previously switched already. For each thread this function allocates one memory from the heap. The size of this block is the stack size plus the size needed for the NUTTHREADINFO structure.

Lower memory
Free stack space
Switch frame
Entry frame
Thread info structure
Upper memory

Table 3: Usage of the thread's memory block.

The thread's info structure is located at the end of this memory block. Immediately in front of it NutThreadCreate() put a so called entry frame. The format of this frame is dictated by the compiler and will be used later by NutThreadEntry().

Lower memory
R25
R24
RAMPZ
SREG
R1
Return address high byte
Return address low byte
Upper memory

Table 4: Layout of the entry frame.

In front of the entry frame NutThreadCreate() puts the switch frame. This frame looks exactly like a stack would look like after a context switch. Remember that the stack pointer of a thread is stored in the NUTTHREADINFO structure when the thread is stopped during context switch. NutThreadCreate() initializes this item with the address of the byte in front of the switch frame. When the CPU executes a pop instruction, the stack pointer is incremented first and then the byte pointed to by the stack pointer is loaded.

Lower memory
R31
30 more CPU registers
R0
Return address high byte
Return address low byte
Upper memory

Table 5: Layout of the switch frame

Finally NutThreadCreate() inserts the NUTTHREADINFO structure to a special priority queue, which includes all threads that are ready to run.It doesn't really matter at this point, wether our newly created thread made it on top of this list or not. Sooner or later it will become the topmost entry picked up by NutThreadSwitch(). This function

will then find exactly the stack layout it expects from previously stopped threads. It will load all 32 CPU registers while incrementing the stack pointer.

When NutThreadSwitch reaches its return instruction, it "jumps back" to the other routine which we introduced above: NutThreadEntry(). Well, it doesn't really return, because it never had been called from there. But NutThreadSwitch put the associated return address on the stack.

NutThreadEntry() will find a prepared stack too. When it returns, it "jumps back" to the thread's starting point.

The very first thread ever started, the idle thread, is handled a bit different by NutThreadCreate(), because there's nothing to switch from. NutThreadSwitch() contains a second entry point, created by an assembly language label. This entry point has been put just before the stack will be switched to a new thread. NutThreadCreate use an assembly instruction to jump to this label. That's why NutThreadCreate never returns when creating the idle thread.

3 Events

The heart of Nut/OS.

As we read in the last chapter, threads are running as long as there's something to do. In typical applications most threads spend most of the time waiting for something.

Nut/OS provides an event queue mechanism. Threads can line-up in such queues when they are waiting for an event while other threads can post events to these queues to wake up waiting threads. This is also known as thread synchronization because of the interaction between running and waiting threads.

Let's look at a simple example in order to get a better idea of what's happening inside Nut/OS. An application thread may wait for a command line on a TCP/IP port and calls NutDeviceGetLine(). This API function or functions called by this API function will find an empty input buffer on the specified device and call

```
int NutEventWait(volatile HANDLE *qhp, u_long ms);
```

The first parameter is a handle of a waiting queue for all threads waiting for data from the Ethernet interface. The second parameter provides a timeout value given in milliseconds, which isn't important right now and may be set to zero to disable it. NutEventWait will mark the current thread not-ready-to-run and add it to the waiting queue. It will switch the context to another thread which is ready to run with the highest priority. Remember, that at least the idle thread is always ready.

Now nothing happens until an Ethernet frame is received by the Ethernet controller hardware. This will trigger a hardware interrupt and the CPU context is switched to the Ethernet interrupt routine, which in turn posts an event to the waiting queue of threads waiting for input from the Ethernet interface. This signal will wake up our application thread, which starts processing the data received. Finally it comes back, calling NutDeviceGetLine() again an the whole cycle repeats.

In fact, things are a bit more complicated, but the underlying principle should have become clear.

Priority Queues

The priority queues used by Nut/OS are most simple constructions. Nevertheless they represent the central structure for thread scheduling and synchronization.

At a first glance, a queue is nothing more than a void pointer. If the pointer is set to null, the queue is considered empty. A thread is added to the queue by setting the void pointer to the thread's info structure. Each info structure contains a void pointer itself, which can be used to point to another info structure. This way Nut/OS can set

up linked lists of info structures and use the void pointer of a wait queue to point to the root element.

Priority queues add a little extra by keeping the linked list sorted based on the priority of the thread associated with the info structure. This is quite easy, because the priority number is part of the thread info structure.

The Nut/OS event API makes use of two functions when handling priority queues:

```
void NutThreadAddPriQueue(NUTTHREADINFO *td,
NUTTHREADINFO **tqpp);

void NutThreadRemoveQueue(NUTTHREADINFO *td,
NUTTHREADINFO **tqpp);
```

The first parameter is the pointer to the info structure of the thread to be added to or removed from the queue. The second parameter is the pointer to the queue, which itself is a pointer to the first info structure in this queue.

Although possible, applications typically do not call these functions. Many applications will never use any event function directly. Every application, however, uses them indirectly when calling input, output or timer functions, if we consider NutThreadYield() as a special case of the timer function NutSleep().

Nevertheless, the Nut/OS event API is most useful for thread synchronization. In fact it's the only way to do it, because Nut/OS doesn't provide semaphores or message queues. If you feel you need these missing functionality, it could be build on top of the event API as a part of your application.

Posting Events

Events can be posted to a queue by calling

```
void NutEventPost(HANDLE *qhp);
```

If one or more threads are waiting on this queue, the first one will be removed from the queue and becomes ready-to-run. Because of the priority ordered queue, this is automatically the thread with the highest priority.

If the queue is empty, it will be switched to a special state called signaled. If a thread intends to wait on a signaled queue, it will remain ready-to-run.

Calling

```
void NutEventBroadcast(HANDLE *qhp);
```

will wake up all threads on the specified queue.

Interrupts

Remember, that interrupt routines somehow break the cooperative multithreading nature of Nut/OS. Even if the running thread is not willing to pass the CPU to another thread, interrupt routines are executing immediately if interrupts are enabled. On the other hand, threads in a cooperative multithreaded environment rely on the consistency of shared resources without the need to gain exclusive access.

Nut/OS solves this conflict by restricting access for interrupt routines. The following two API functions are almost the only ones, which are allowed to be called by interrupt routines:

```
void NutEventPostAsync(HANDLE *qhp);
void NutEventBroadcastAsync(HANDLE *qhp);
```

Both functions work the same way as their synchronous conterparts, but they do not perform the final context switch. They just modify the wait queue and make threads ready-to-run. The interrupted thread continues to run after the CPU finished executing the interrupt function.

This way it is possible, that a lower priority thread is running while a higher priority thread is ready-to-run. As soon as the running thread calls NutThreadYield(), either directly or indirectly, the CPU will switch to the higher priority thread.

Sometimes it is even necessary to execute parts of the code without being interrupted by a interrupt routine. For example, the two API functions above will probably corrupt the wait queue, if they are interrupted by an interrupt routine, which itself tries to modify the same queue by calling the same function.

Simply disabling the interrupt by clearing the interrupt enable flag in the status register of the CPU might do the trick. But re-enabling interrupts when done might confuse the calling function, if it relies on running with disabled interrupts too. Nut/OS offers to functions to retain the interrupt status among nested routines:

```
void NutEnterCritical(void);
void NutExitCritical(void);
```

NutEnterCritical() pushes the contents of the CPU status register on the stack and disables interrupts, while NutExitCritical() restores the contents of the CPU status register.

Waiting for Events

The next event function had been introduced already:

```
int NutEventWait(HANDLE *qhp, u_long ms);
```

As stated earlier, this function stops the calling thread and adds it to the queue, which is specified by the first parameter. The second parameter allows us to specify the number of milliseconds of a maximum waiting time. If this time elapses without any

event posted to the queue, then the thread will become ready to run anyway. In that case the function returns -1 instead of zero.

Internally Nut/OS starts a timer, which posts an event if it elapses. Timers will be handled in the next chapter.

4 Timers

Time triggered events.

This chapter is not completed.

Since version 2.2 the timer handling in Nut/OS had been changed significantly. While previous versions used an additional thread to handle an internal list of running timers, timer events are now processed in the timer interrupt routine. This was made possible by Marc Wetzel, who suggested several optimizations.

Two timer functions had been introduced already:

```
void NutTimerInit(void);
void NutSleep(u_long ms);
```

Do not even think of calling the first one unless you write your own initialization function. The second function, NutSleep() stops the execution of the calling thread for at least the specified number of milliseconds.

Starting and Stopping Timers

NutSleep() makes use of another function:

```
HANDLE NutTimerStart(u_long ms, void
(*callback)(HANDLE, void *), void *arg, u_char
flags);
```

This creates a new timer.

To remove a timer from the list and destroy it, call

```
void NutTimerStop(HANDLE handle);
```

Short Execution Delays

Sometimes very short delays are needed. Very often programmers use a simple loop, which does nothing else than incrementing a loop counter. However, be aware, that the compiler may find out, that this loop is irrelevant. In this case the loop is completely removed. Another approach is to include assembly statements containing just NOP codes. When they are declared volatile, the compiler will leave them untouched. This works fine for very short delays, but note, that the actual delay time depends on the CPU speed.

For delays in the range of some millisecond, Nut/OS offers a delay routine, which adjusts itself to the CPU speed:

```
void NutDelay(u_char ms);
```

In other situations the application might need to know the CPU speed, which is offered as a return value from

```
u_long NutGetCpuClock(void);
```

5 Links

Where to find additional information.

http://ethernut.sourceforge.net/

Ethernut developer forum.

http://www.ethernut.de/

Information about the Ethernut board.

http://www.egnite.de/

Home of egnite Software GmbH, the developer of the Medianut software and Ethernut hardware.

6 Index