Chapter 4

The BTnut Operating System

4.1 Introduction

In this chapter, we introduce the BTnut operating system (OS). In comparison to the exercises of the previous chapter, this has two main consequences:

- You do not have to read hardware schematics and spec sheets when you want to use resources, since we are now able to use library functions. In this chapter you will use such functions for accessing the LEDs and the terminal. Also the analog to digital converter that we have used in the last chapter would be accessible through such library functions – see the `btnut/btnode/include/dev/adc2.h` header for a description. There is even a function `btn_bat_measure` that does exactly what we have done manually (see `include/hardware/btn-bat.h`).
- Complicated programs can be divided into a set of threads. Programming a single thread is often much easier than programming the whole functionality in a single program. At the same time, however, programming with multiple threads becomes more complicated. Even though the execution coordination of these threads is done by the operating system, we still need to properly code concurrent threads, especially when it comes to two or more threads using a common resource (e.g., the radio, or the terminal). Threads will be described in the following chapter, where we introduce the API of the BTnut OS for creating, executing and terminating threads, as well as for the communication and coordination of such threads.

The following sections will explain the anatomy of a simple BTnut program using the example of LED control (section 4.2). Section 4.3 will then explain how we can use BTnut to provide output over a terminal connection (i.e., through the USB cable).

4.2 Anatomy of a BTnut Program

Recall from chapter 1 that BTnodes run an embedded systems OS from the open source domain, called Nut/OS. Nut/OS is designed (among others) for the Atmel ATmega128 microcontroller (which is used on the BTnodes), and is thus an excellent base on which the BTnut extensions provide additional device drivers to access BTnode-specific hardware. The actual compilation of your programs (i.e., the translation of C-code into machine-code) is done using `gcc-avr` (part of `WinAVR` on Windows), which is a freeware C compiler (and assembler) for the Atmel processor platform. We thus have three parts to our BTnode OS-experience: the rudimentary C-libaries as implemented by `gcc-avr`'s *avr-libc*; the higher-level OS routines built on top of *avr-libc* by Nut/OS; and the BTnode-specific device drivers provided by BTnut. In the following, we will simply call this layered OS architecture "BTnut", yet one should keep the differences in mind in order to better understand the overall system operation.

We first look at a minimal BTnut program.

Explanation *A minimal BTnut program*: BTnut programs are written in C (though we don't have access to all libraries that we are used from our PCs). Just as any other C-Program, they feature a `main` function as the initial focus of control, i.e., this is the first function that gets executed after power-up of your BTnode. However, in contrast to regular PC programs, our BTnut programs must *always* begin with initializing the BTnode hardware:

```
#include <hardware/btn-hardware.h> // for btn_hardware_init

int main(void)
{
    /* ALWAYS call this func at the beginning of main */
    btn_hardware_init();           /* initialize SRAM */

    for(;;)
    {
        // do something clever here
    }

    /* you should never reach this point */
    return 0;
}
```

Another peculiarity of a BTnut program is that it should never actually finish. In contrast to a PC program, one cannot return to the command line after the execution of a particular application is done – BTnodes are expected to continously execute their task! If their main program ends, the behavior of a BTnode is undefined (it might simply restart, or stop altogether, or . . .).

Obviously, an empty program is not very exciting. Let's see how the LED control described in the previous chapter can be implemented through BTnut OS function calls.

Explanation *Using the on-board LEDs*: BTnut OS offers through `<led/btn-led.h>` the functions `void btn_led_set (u_char nr)` and `void btn_led_clear (u_char nr)`, where `nr` denotes the LED in question, namely 0 through 3. Before the LEDs can be controlled this way, we need to initialize them first:

```
#include <hardware/btn-hardware.h> // for btn_hardware_init
#include <led/btn-led.h>           // for led-related functions

int main(void)
{
    btn_hardware_init();           /* initialize SRAM */
    btn_led_init(0);               /* initialize LEDs */

    btn_led_clear(LED0);    btn_led_set(LED1);
    btn_led_clear(LED2);    btn_led_set(LED3);

    for(;;);                       /* endless loop */

    /* you should never reach this point */
    return 0;
}
```

Notice the argument that `btn_led_init` takes – it indicates whether we want to activate an *LED heartbeat*, i.e., the periodic blinking of one or more LEDs to indicate that our BTnode is "alive" (as we didn't want a heartbeat in the above example, we used 0). For more information on LED heartbeats, see page 36.

Exercise 4.1. *Write a program that endlessly rotates through the four LEDs (i.e., it turns one after another on and off). Observe the output. Use* `for` *statements to slow the rotation down until it becomes easily visible.*

Optional Exercise 4.2. *Have your program from Ex. 4.1 terminate after a few rotations (you will need to add* `return 0` *to the end to get it to compile). What behavior do you observe?*

4.3 The Terminal

In order to communicate back to the user (or programmer), we are not restricted to using the on-board LEDs only. Through our USB-cable, we can setup our BTnode in such a way that `printf` statements provide output that can be printed in a terminal program on our PC, and use `fscanf` to read user input from within the terminal program and input it back into our BTnut program. This is where the ATmega128's UART ports – Universal Asynchronous Receiver Transmitter – come into play. Actually, the ATmega128 supports USART ports – Universal Synchronous/Asynchronous Receiver/Transmitter. Consequently, you will notice that some libaries and functions actually use `usart` in their names. However, as it does not make much of a difference, and as UART is the much more common interface, we will continue to use that term.

The ATmega128 has two UART interfaces – UART0 and UART1. While UART0 is used to connect the ATmega128 to the Bluetooth module, we can use UART1 to write ASCII text to the *terminal*, i.e., a program running on the host computer that uses well-known communication protocols to send and receive text from a remote computer. See the online documentation at `http://www.btnode.ethz.ch` for information on how to setup a terminal program under Linux (e.g., `minicom`) or Windows (e.g., Hyperterm).

Explanation *Setting up the terminal*: The `printf` function writes a formatted string to the standard output stream. But before using `printf`, we have to setup the standard output stream explicitly, i.e., we have to define that we want to link the standard output to the UART1. This can be done using a routine like to following:

```
#include <hardware/btn-hardware.h>  // always required
#include <stdio.h>                  // freopen, includes <io.h> for _ioctl
#include <dev/usartavr.h>           // NutRegisterDevice, APP_UART, UART_SETSPEED
#include <terminal/btn-terminal.h>  // also required

void init_stdout(void) {
    u_long baud = 57600;

    NutRegisterDevice(&APP_UART, 0, 0);
    freopen(APP_UART.dev_name, "r+", stdout);    // "r+": read+write
    _ioctl(_fileno(stdout), UART_SETSPEED, &baud);
}
```

Once we have established a terminal connection, we can write text to it using the well-known `printf` function (which is provided for the ATmega128 platform in *avr-libc*). Most standard conversion strings (e.g. `%d` for signed integers) and special characters (e.g. `\n`) can be used, but not all. For example, the float conversion (`%f`) is not implemented as the ATmega128 does not support floating point operations. **HINT:** Exit the terminal program again before trying to upload a new program to the BTnode, otherwise the bootloader's upload replies will be caught by the terminal program, not your upload tool (i.e., the upload will fail).

```
int main(void)
{
    btn_hardware_init();           /* initialize SRAM */
    init_stdout();
    int variable = 13;
    printf("Hello world, ");
    printf("my lucky number is %d\n",variable);
    for (;;);                      /* main should never return */
    return 0;                      /* required by gcc 4.x */
}
```

To read data from the terminal, you can use the function `fscanf`. However, in order to read input from the user, we can use an already defined library – `<terminal/btn-terminal.h>` – which offers convenient access to user input. Specifically, through the use of `btn-terminal`, a programmer can define a set of *commands* and optional arguments that a user can execute from a terminal prompt. Upon hitting the Tab-key, the BTnode terminal program lists all available commands.

Explanation *The Interactive Terminal*: The BTnode terminal is defined in `terminal/btn-terminal.h`. After initializing it with the stream to use and prompt to display, the programmer simply has to "run" it:

```
#include <hardware/btn-hardware.h>
#include <dev/usartavr.h>           // NutRegisterDevice, APP_UART, UART_SETSPEED
#include <terminal/btn-terminal.h>  // for interactive terminal, includes stdio.h and thus io.h

int main(void) {
    btn_hardware_init();
    init_stdout();                             /* as defined above */
    btn_terminal_init(stdout, "[btn3]> ");
    printf("\nHowdy!\n");
    btn_terminal_run(BTN_TERMINAL_NOFORK, 0); /* NOFORK never returns */
    return 0;                                  /* required by gcc 4.x */
}
```

After the usual initializations (for an explanation of `init_stdout`, see page 29), the terminal thread is initialized with `btn_terminal_init`, the first argument links it with the UART of the standard output stream, the second argument defines the prompt of the command line (you may use any string you like). Finally, the command `btn_terminal_run(BTN_TERMINAL_NOFORK, 0)` starts the terminal. The function never returns, unless you use `BTN_TERMINAL_FORK` as the first parameter, in which case the terminal is started in a separate *thread*. We will explain threads in chapter 5.

Using a terminal program such as *minicom* or *Hyperterm*, we can now interact with our BTnode program. However, except for a small message and a prompt, there isn't yet much we can do. Fortunately, BTnut already comes with a few commands that we can readily make available in our terminal.

Explanation *Predefined Terminal Commands*: The BTnut OS offers sets of predefined terminal commands. To use them, they have to be registered. Two of these sets, with the corresponding header file and the register function, are given below. Similar command sets also exist for the bluetooth radio (`bt_*`) and the debug logger (`log_*`).

```
#include <terminal/btn-cmds.h>
    btn_cmds_register_cmds();

#include <terminal/nut-cmds.h>
    nut_cmds_register_cmds();
```

The register commands have to be called after `btn_terminal_init` and before `btn_terminal_run`. For example, `btn_cmds_register_cmds` provides the *led* command, `nut_cmds_register_cmds` provides the *nut* command, which has several sub-commands. Hit the Tab-key for a list of available commands.

Exercise 4.3. *Incorporate the default BTnut commands into your small terminal application. Use the* nut threads *command to show the currently running threads. Now include* `led/btn-led.h` *and add a call to* `btn_led_init(1)` *right after initialization. Also, add* `NutSleep(1000)` *to the final* `for`*-loop so that it isn't empty (this needs* `sys/timer.h`*). Observe the output of* nut threads *now.*

Exercise 4.4. *Change the call to* `btn_terminal_run` *to use* `BTN_TERMINAL_FORK` *as its first parameter (you can leave the second argument "0" right now) and check the output of* nut threads *again.*

Explanation *Creating your own Terminal Commands*: You can also register your own commands with `btn_terminal`. You must provide a function with a standardized interface (a pointer to a single argument of type `char`), which can then be registered under an arbitrary command name:

```
void _cmd_square(char* arg) {
    int val;
    if (sscanf(arg,"%d",&val)==1) {
        printf("The square of %d is %d\n",val,val*val);
    }
    else { printf("USAGE: square <value>\n"); }
}

int main(void) {
    ...
    btn_terminal_init(stdout, "[btn3]> ");
    btn_terminal_register_cmd("square",_cmd_square);
    btn_terminal_run(BTN_TERMINAL_NOFORK, 0);
    for (;;); /* main should never return */
    return 0; /* required by gcc 4.x */
}
```

The command *square* is registered with `btn_terminal_register_cmd` *after* the initialization of the terminal. The first parameter is the string you will have to type to launch the function, whose identifier is given as the second argument. Note that functions that you want to register as a commands must have the signature `void <functionname>(char* arg)`.

Exercise 4.5. *Write a program that registers the command* echo*, which simply echos all the given arguments.*

Optional Exercise 4.6. *Write a program that registers the commands* myset *and* myclear*, which will take the numbers 0-3 as an argument, and set or clear the given LED.*

4.4 Timers

Explanation *Timers in BTnut*: Instead of using `for`-loops or `NutSleep` calls, you can also use one or more *timers* to schedule recurring function calls. Using timers, you can easily parallelize your program without the need for explicit thread-management: simply create a function for each required aspect of your program, and register different timers for each of them. BTnut will then take care of calling these functions in the given intervals.

```
#include <hardware/btn-hardware.h>
#include <sys/timer.h>

HANDLE hTimer;

static void _tm_callback(HANDLE h, void* a) { . . . }

int main (void)
{
    btn_hardware_init();
    hTimer = NutTimerStart( 3000, _tm_callback, NULL, 0 );
    for (;;) { NutSleep (1000); } /* never end */
    return 0; /* required by gcc 4.x */
}
```

You can use `NutTimerStart` and `NutTimerStop` to install or remove periodic timers. Using `TM_ONESHOT` as the last parameter to `NutTimerStart` will automatically remove that timer after it has run once (i.e., after one interval) – using “0” instead installs a periodic timer.

Exercise 4.7. *Write a program with two timed callback functions: one should repeatedly turn on the blue LED (using `btn_led_set(LED0)`) and switch off the red LED (using `btn_led_clear(LED1)`), the other shall do the opposite, i.e. turn on the red LED and switch off the blue LED.*

Optional Exercise 4.8. *Write a program that periodically calls a function to "shift" the current LED by one position (after the last position, it should begin again with the first).*

Optional Exercise 4.9. *Create a terminal application that allows to control the LED shifting functionality implemented in Ex. 4.8, i.e., use* `btn_terminal_register_cmd` *to create a command "toggle" that turns the shifting on or off.*

4.5 Dynamic Memory Management

Typically there is no need for dynamic memory management in your BTnode program. Simply create global or local variables, and the BTnut OS (together with `gcc-avr`) will take care for you to properly allocate the ATmega128 SRAM (of which we have 64kBytes) for the stack and the global and/or static variables.

However, should the need arise (e.g., you might want to limit the stack size of your current thread, or you have temporary data that is very large), BTnut also supports dynamic memory allocation using `malloc` and `free`.

Explanation *Dynamic Memory Allocation*: If you need to allocate memory directly, you can use Nut/OS's own `malloc` and `free`:

```
#include <hardware/btn-hardware.h>
#include <stdlib.h> // malloc, free
#include <string.h> // memset

u_int BUF_SIZE = 4096;

int main(void) {
    char *buffer;
    btn_hardware_init();

    buffer = malloc(BUF_SIZE);
    if (buffer != NULL) {
        /* fill buffer with data */
        memset(buffer, 0xFF, BUF_SIZE);
        /* and free it again (really useful) */
        free(buffer);
    } else { /* out of memory */ }

    for (;;);  /* endless loop */
    return 0;  /* required by gcc 4.x */
}
```

In addition to the 64kBytes of directly accessible SRAM, the BTnode also features three banks of *external data cache* – each having 60 kBytes – for a total of 180 kBytes of external storage. This memory cannot be allocated directly from within our BTnut program, as our ATmega128 processor can only address 64 kBytes of RAM. Instead, one has to briefly switch the upper 60 kBytes of our "regular" SRAM with one of the three available banks, in order to access data in there. This functionality is implemented in the `cdist/xbankdata.h` and can be tested by including the `terminal/xbank-cmds.h` commands.

See the sources of `terminal/xbank-cmds.c` in the `btnut`-sourcetree if you want to learn more about banked memory. In this tutorial, we will not further elaborate on this feature of the BTnut OS.

Chapter 5

Programming with Threads

5.1 Introduction

In this chapter, we introduce thread programming with BTnut. A *thread* is simply a function or a small program that can run concurrently to another thread. Using threads, we can actually write BTnode programs that listen to incoming commands over their radio, periodically measure sensor values, compute intermediate results, and send data to other nodes. All (virtually) at the same time!

This *multithreading* is handled by Nut/OS, as obviously our ATmega128 microcontroller can only handle one instruction at a time. In order to support multithreading (or multitasking) on a single core processor, the operating system needs to repeatedly start and stop individual threads (i.e., *schedule*) – in a completely transparent fashion. Two general approaches to multithreading exist:

- *Preemptive:* The OS has complete control over processes and can stop, pause, and restart them (almost) at will. Most modern PC operating systems (e.g., Linux, Windows 2000/XP) use preemptive multitasking. This is because it allows for a more reliable distribution of resources.

- *Cooperative:* Processes need to manually give up control of the CPU to a central scheduler, which then evaluates which thread or task should come next based on process priorities and queues. While cooperative multithreading simplifies resource sharing, and usually results in faster and smaller code (making it thus more suited to embedded systems programming), it runs the risk that a poorly designed or "hung" process can bring the entire system to a halt.

Nut/OS – and therefore also BTnut – employ *cooperative multithreading*. This means that in order to execute two or more threads at the same time, each process needs to periodically give up control to the OS scheduler. This will be described in more detail in section 5.3 below, right after we explain how to create our own threads in section 5.2. Sections 5.4 and 5.5 finally introduce the concepts of *mutexes* and *events*, respectively, which are means of coordination and communication between threads.

5.2 Creating Threads

Using BTnut, we can easily define and run our own threads. First we look at how threads are defined.

Explanation *Creating Threads*: Threads are functions. As mentioned earlier, the main routine itself is also a thread, which is started automatically after startup (fortunately, this is completely transparent to the programmer, thanks to Nut/OS). Additional threads have to be declared using the `THREAD` macro. An example defining the thread `my_thread` is shown below.

```
#include <sys/thread.h>

THREAD(my_thread, arg) {
    for (;;) {
        // do something
    }
}
```

Functions that are used as threads are supposed to never return, thus to loop endlessly (if you want to end a thread, you will need to manually call `NutThreadExit`). The second argument of the `THREAD` macro, called `arg` here, is a void pointer and can be used to pass an argument of arbitrary type to the thread when it is created (note that the actual *declaration* of `arg` as a void pointer is done by the macro). Don't forget to include `<sys/thread.h>` for working with threads!

The thread `my_thread` is now defined, but it has to be started before it becomes active.

Explanation *Running Threads*: A thread can be activated by any other thread, e.g. by the main routine. This is done using the command `NutThreadCreate`.

```
#include <sys/thread.h>

int main(void) {
    if (NutThreadCreate("My Thread", my_thread, 0, 192) == 0) {
        // Creating the thread failed
    }
    for (;;) {
        // do something
    }

    return 0; /* required by gcc 4.x */
}
```

The first parameter defines a name for the thread, the second parameter is the name of the function we have defined before. The third argument is a pointer that is passed to the thread function (compare with the second argument `arg` of the `THREAD` macro) – we do not use this feature here and thus an arbitrary value can be used. The last argument is the size of the stack that is allocated for the thread. This stack is used for local variables and for passing arguments when calling subroutines. If this value is chosen too large, the system may run out of heap memory. If it is chosen too small, the thread overwrites memory that is used otherwise, which results in unpredictable behavior. See page 30 for a method to check whether your stack size is correctly chosen. For now, just use 192 and you will be fine.

Exercise 5.1. *Write a program that creates a thread as explained above. This thread shall repeatedly turn on the blue LED (using `btn_led_set(LED0)`) and switch off the red LED (using `btn_led_clear(LED1)`) (**HINT:** `LED0` and* LED1 *are just macros for the numbers we were using before, i.e., "0" and "1", respectively. However, using symbolic names instead of numbers makes our programs more portable). The main routine, after having created the thread, shall do the opposite, i.e. turn on the red LED and switch off the blue LED. Which LEDs are switched on? Why? Add a single `NutThreadYield` such that the other LED is switched on. Add a second `NutThreadYield`, such that both LEDs are switched on by turns (you will see both LEDs switched on, because the main routine and the thread alternate very quickly).*

5.3 Thread Control

Exercise 5.1 has demonstrated the cooperative nature of the BTnut OS. In order to have two or more threads running, they need to repeatedly and continously *give up control* of the CPU and other resources, so that other threads may run.

In principle (we will see an exception later on), active (i.e., running) threads only yield the CPU to other threads if this is explicitly coded. The most simple way to do this is `NutThreadYield`, a function that has no parameters. This function causes the OS to check whether other threads with higher or equal priority are ready to run (we will explain thread priorities below). If this is the case, the current thread is suspended, i.e. `NutThreadYield` does not return and the thread with the highest priority among those that are ready to run is given the CPU. Otherwise, `NutThreadYield` returns immediately.

Explanation *Giving up Control*: In order to support concurrent threads on the BTnode, each thread, even the `main()` function, must periodically yield control. A call to `NutThreadYield` basically means "Is there any process that is more important than myself? If so, feel free to take over control. Otherwise, I will simply continue." Once control has been given away and is returned at a later point in time, the thread will continue to run right after the call of `NutThreadYield`.
If a thread has nothing to do, it can also force cease of control by calling `NutSleep(time)`. This function puts the current thread into SLEEPING mode and transfers control to any thread that is waiting for control (i.e., is READY, see below). If no thread is waiting, the *idle* thread takes over, which is always ready-to-run (but which has the lowest priority – see below).
Note that threads might also give up control involuntarily – in case of an *interrupt*. See the BTnut documentation for details.

Explanation *BTnut System Threads*: We have already encountered a number of threads before, created and run by the BTnut OS: the LED thread (see page 28 and excercise 4.3), the terminal thread (see section 4.3 and excerise 4.4), as well as the idle thread and the main thread (again from exercises 4.3 and 4.4). We can visualize the currently active threads using the `nut threads` command (see *Predefined Terminal Commands* on page 30):

```
[bt-cmd@00:d2]$ nut threads

Hndl Name      Prio Sta QUE  Timr StkP FreeMem
2057 T_TERMIN   150 RUN 0385 0000 200D   950 OK 2057 0D6A
14A4 LED        150 SLP 0000 2088 1481   989 OK
1087 main        64 SLP 0000 2074 1064   733 OK
0D6A idle       254 RDY 0385 0000 0D4E   356 OK 2057 0D6A
```

In order to have all of these "standard" threats running at the same time, we also need to make sure that *all* of them repeatedly yield the CPU to other threats. As the idle, LED, and terminal threat are all coded for us by the OS programmers (who thoughtfully made all of these threats yield every so often), we only need to make sure that the *main* threat (which is under our control) does so as well!

Exercise 5.2. *Use the minimal BTnut program as described on page 28 and add a LED heartbeat (see page 28) and the predefined Nut/OS terminal commands (see page 30). Start the terminal in a new thread using* `btn_terminal_run(BTN_TERMINAL_FORK, 0)`*, but leave the main routine empty (i.e., use only* `for(;;)`*). Do you see the LED heartbeat? Can you interact with the terminal? Fix your program so that both heartbeat- and terminal-thread can be executed concurrently to your (empty) main program.* **HINT:** *See Ex. 4.3 and 4.4.*

Explanation *Controlling the LED Thread*: Initializing our LEDs with `btn_led_init(1)` also starts a separate *LED thread*. The LED thread displays dynamic patterns on the LEDs, typically to indicate that the program is still running ("heartbeat"). Its pattern can be set with a single command, i.e., using `btn_led_add_pattern` or `btn_led_heartbeat`. See the BTnut system software reference for a detailed description of these commands. By default, the LED thread starts to blink with the blue LED after initialization.
Note that even when the heartbeat is active, we still can switch on and off LEDs individually using the commands `btn_led_set` and `btn_led_clear`. The LED thread will remember the pattern it was showing before LEDs are switched on manually and restart displaying the pattern after all these LEDs are cleared again manually.

Exercise 5.3. *Write a program with a main routine and an additional thread. Both threads repeatedly write a message to the terminal and then yield (using* `NutThreadYield`*). What do you observe? (***HINT:** *you can freeze terminal output in minicom using Ctrl-A) Do you have an explanation?* **HINT:** *Writing to the terminal is done with the speed of the UART, e.g., 57600 bits per second, which is slow in comparison to the speed of the CPU.* **HINT No. 2:** `printf` *does not directly write to the UART, instead it writes to a buffer with a limited capacity (default is 64 characters).*

Explanation *Thread Priorities and States*: Each thread in BTnut has a *priority* – a value from 0–254. The *idle* thread has the lowest possible priority, 254, while *main*, as well as all manually created threads, have a default priority of 64.
Priorities become important if several threads are competing for control. Each thread can be in three different states: RUNNING, READY, or SLEEPING. While only one thread can be RUNNING at any moment in time (this is managed by the BTnut OS), several can either be READY or SLEEPING. A sleeping process has ceded control either by calling `NutSleep(time)` (and is woken up by the OS after at least *time* milliseconds have passed) or is waiting for an *event* (e.g., an incoming radio signal, or a message from another thread – more about this in section 5.5 below).
Once the running thread cedes control using `NutThreadYield`, its state becomes READY, and BTnut transfers control to another ready-to-run thread – the one with the highest priority. If all other ready-to-run threads have a *lower* priority, control is returned to the yielding thread immediately (otherwise some unknown time later). Multiple threads with the same priority are executed in FIFO order.

Explanation *Setting Thread Priorities*: Remember that in the BTnut OS, threads have a priority in the range of $[0, 254]$, where a lower value means a higher priority. The default priority is 64. You may assign the current thread a higher priority, e.g. 20, using

```
THREAD(my_thread, arg) {
    NutThreadSetPriority(20);
    for (;;) {
        // do something
    }
}
```

Note that changing the priority of a thread implies a `NutThreadYield` (i.e., setting its state to READY), thus potentially yielding the CPU to another thread. This is the case if the running thread reduces its priority, and is thus no longer the thread with the highest priority that is ready to run.

Exercise 5.4. *Take the program from Ex. 5.1 and give the self-created threat a higher priority. Compare the output with the original program from Ex. 5.1. Repeat the experiment giving the self-created thread a lower priority. What do you observe?*

Optional Exercise 5.5. *Repeat Ex. 5.3 but give the additional thread a higher priority. Compare the output with what you received in Ex. 5.3. Repeat the experiment giving the additional thread a lower priority. What do you observe?*

Optional Exercise 5.6. *Repeat Ex. 5.3 but use* `NutSleep` *instead of* `NutThreadYield`*. Why is the output different from Ex. 5.3?*

Explanation *Terminating Threads*: A thread can terminate itself as shown below.

```
THREAD(my_thread, arg) {
    for (;;) {
        // do something
        if (some condition)
            NutThreadExit()
    }
}
```

There is no easy way for some thread A to kill another thread B. Nevertheless, you will implement this functionality in Ex. 5.13.

Combining thread creation with our means of writing command-line applications for the terminal (see section 4.3), we can experiment with thread creation and termination more conveniently:

Exercise 5.7. *Write a program that registers the command* create *as a terminal command. This command takes a string argument and creates a thread with this name. This thread periodically prints its name on the terminal and then sleeps for a second.* **HINT:** *A thread can access its own name using* `runningThread->td_name`*, e.g., using* `printf("%s ready\n",runningThread->td_name)`.

Optional Exercise 5.8. *Rewrite the program from Ex. 5.7 such that the first thread you start sleeps for one second, the second thread sleeps for two seconds, etc.* **HINT:** *For this purpose, you may use the third argument of the* `NutThreadCreate` *to pass the sleep time to the thread. Another alternative would be to use a global data structure.*

Optional Exercise 5.9. *Rewrite the program from Ex. 5.7 so that the* create *command takes a second parameter specifying the stack size of the thread that is created. Use this command and* nut threads *to figure out how much stack is actually used by the threads you create. Add some local variables to these threads and/or call some dummy functions from these threads to see how this increases the amount of used stack.*

5.4 Sharing Resources: Mutual Exclusion (Mutex)

Ex. 5.3 showed the problem of two or more threads trying to access the same resource (the terminal) concurrently. As a result, their output was garbled. One way to coordinate shared resources is the use of a *mutex* – a lock mechanism for *mutual exclusive* resource usage.

Explanation *Mutexes in BTnut OS*: Shared resources can be used exclusively by a thread by signaling current use over a mutex. After defining a mutex using `NutMutexInit(&myMutex)`, threads can use `NutMutexLock` and `NutMutexUnlock` to reserve the resource managed by the mutex:

```
#include <hardware/btn-hardware.h>
#include <dev/usartavr.h>          // NutRegisterDevice, APP_UART, UART_SETSPEED
#include <stdio.h>
#include <io.h>
#include <sys/thread.h>

#include <sys/mutex.h>

MUTEX myTerminal;

THREAD(my_thread, arg) {
    for (;;) {
        NutMutexLock(&myTerminal);
        printf ("This is Thread One\n");
        NutMutexUnlock(&myTerminal);
        NutThreadYield();
    }
}

void init_stdout(void) {  . . . }

int main(void) {
    btn_hardware_init();
    NutMutexInit (&myTerminal);
    init_stdout();                          /* as defined previously */

    if (0 == NutThreadCreate("ThreadOne", my_thread, 0, 192)) {
            printf ("Sorry, could not create ThreadOne. Stopping...\n");
            for (;;);
    }
    for (;;) {
            NutMutexLock(&myTerminal);
            printf ("This is the Main Thread\n");
            NutMutexUnlock(&myTerminal);
            NutThreadYield();
    }

    return 0; /* required by gcc 4.x */
}
```

Exercise 5.10. *Rewrite the program from Ex. 5.3 so that each thread first acquires a mutex lock before printing to the terminal.*

Exercise 5.11. *Write your own little "thread-safe"* `printf` *function that uses mutexes in order to exclusively acquire use of the terminal.* **HINT:** *You can either use a separate function call (e.g.,* `my_printf`*) or a preprocessor macro (e.g.,* `PRINTF`*). Using a macro should simplify argument handling, as* gcc *supports so-called* variadic macros*:*

```
#define PRINTF(...) {  \
  /* acquire mutex */  \
  printf (__VA_ARGS__) \
  /* release mutex */  \
}
```

5.5 Events

Another important part of a multithreaded OS is *communication between threads*. This allows the coordinated use of multiple processes, as threads can signal other threads when to wake up, or in general inform a number of threads that they have finished processing a particular data structure.

Explanation *Sending and Receiving Events*: The coordination (synchronization) of threads can be done using BTnut *events*. Consider the example shown below:

```
#include <sys/event.h>

HANDLE my_event;

THREAD(thread_A, arg) {
    for (;;) {
        // some code
        NutEventWait(&my_event, NUT_WAIT_INFINITE);
        // some code
    }
}

THREAD(thread_B, arg) {
    for (;;) {
        // some code
        NutEventPost(&my_event);
        // some code
    }
}
```

Here we see two threads. Thread `thread_A` executes some code and then blocks in the `NutEventWait` function. It only continues when either an event is posted or the timeout expires. The timeout is specified in milliseconds with the second parameter. In the example shown above, the timeout is disabled, i.e. an infinite time is specified with the macro `NUT_WAIT_INFINITE`.

Exercise 5.12. *Write a program with three threads (main and two additional threads) and a global variable with initial value 2. The three threads shall execute in turns, which you implement with events. One thread computes the square of the global variable, the second decrements it by one and the third multiplies it by two. All threads print the result on the terminal. When the global value has reached a value greater than 10000, all threads except the main routine terminate themselves. The main routine enters an endless loop.*

Exercise 5.13. *Extend the program from Ex. 5.7 with the terminal command* kill *that takes the name of a previously created thread as an argument. The terminal thread shall use an event to inform the selected thread that it is supposed to kill itself.*

Optional Exercise 5.14. *What happens if first an event is posted by some thread A and only afterwards some thread B does a* `NutEventWait` *? What happens if multiple events are posted before another thread is ready to receive them? Are the events stored or lost? Write a program to find out.*

Optional Exercise 5.15. *What happens if two threads are waiting for the same event? Are both threads woken up? Do thread priorities play a role? Write a program to find out.*

Exercise 5.16. *Modify the program from Ex. 5.13 so that each created thread mostly sleeps (e.g., using* `NutSleep(15000)`*) and only* briefly *listens for a kill signal (e.g., using* `NutEventWait(&killqueue, 125)`*). Observe what happens if you send a process a kill signal. Does it eventually terminate? Try changing the signal wait command to* `NutEventWaitNext(&killqueue, 125)`*. Can you still terminate a thread with your* kill *command? Why? Search the BTnut API for answers.*

Explanation *Events Signaling in Nut/OS*: When a signal is posted to an event queue (e.g., using `NutEventPost`), Nut/OS checks to see if any thread is waiting for a signal on this queue (using `NutEventWait`). If there are threads waiting, Nut/OS takes the *first* thread only (i.e., the one with the highest priority) and sets its status from SLEEPING to READY. Since posting events with `NutEventPost` implies a `NutThreadYield`, the posting thread will also become READY. Depending on the priority of the available ready-to-run-threads, Nut/OS might continue with the posting thread, switch to the signaled thread (i.e., the one that was waiting), or even execute a completely different ready-to-run-thread with an even higher priority than those two.
If you want to "wake up" all threads waiting on a particular queue, not just the one with the highest priority, you can use `NutEventBroadcast` instead.

Explanation *Asynchronous Events*: When a thread posts a signal to a particular event queue using `NutEventPost` or `NutEventBroadcast`, Nut/OS might switch control to another thread as these two function calls also imply a `NutThreadYield` (and thus a switch from the RUNNING state to the READY state). In order to continue running, a thread may use so-called *asynchronous* variants of those two functions – `NutEventPostAsync` and `NutEventBroadcastAsync`, respectively – in order to continue being in the RUNNING state. Both routines perform exactly the same signaling as their regular counterparts, yet without executing a context switch. This can be done later using, e.g., a `NutThreadYield` or `NutSleep`.

Exercise 5.17. *In order to gain understanding of threads and thread coordination, we will now write a ping pong game. We have two players standing at opposing ends of a table that hit a ball back and forth, and every time a player misses the table, the other player scores. The player that first reaches 11 points wins.*

In order to implement this, we will need two players (0 and 1), one ball, and one coordinator (referee) that keeps track of points. The sequence of events will look as follows:

1. *The current player will play the ball (initially player 0).*
2. *The ball will move across the table, and either hit or miss the table.*
3. *In case the ball misses the table, the referee scores one point for the opponent. If this player has reached 11 points, the referee ends the game and declares this player winner.*
4. *The sequence repeats at point 1, with player 0 and 1 alternating turns. If the last ball missed the table, the next player will serve, otherwise the point just continues until one player misses the table.*

Implement this algorithm on the BTnode. All the actors in the game will be implemented as a separate thread, coordination among threads is achieved through events. The coordinator records the score and outputs messages to the terminal. After the game finishes, the program ends, but make sure the `main` method never returns, otherwise the BTnode reboots and starts a new game!

HINT: *The ball will be responsible for deciding whether or not it hits the table, i.e. whether the player hits the ball well or not. In order to 'decide' whether a ball hits the table, we will use a random number generator. At the beginning of the program, you will need to include `stdlib.h` and initialise the random number generator by calling* `srand(u_int seed)`. *From then on, every call to `rand()` will give a number between 0 and `RAND_MAX`, so in order to get a random number r number between a and b ($a <= r < b$), use `a + rand() % (b-a)`. Experiment with different 'player qualities', i.e. using a higher probability of playing a good ball and see how it influences the result. Make ample use of* `NutSleep` *in order to make the game "watchable" on the terminal window.*