

# Embedded Systems FS 2012

## Solution to Lab 3: BTnut and Threads

Discussion Date: 04. April 2012

In the first part of this lab session, you will get to know the BTnut operating system (OS). BTnut allows you to use convenient library functions for accessing BTnode resources (*e.g.*, the LEDs) and scheduling periodic as well as one-time operations (*e.g.*, to send an “alive” message once per hour). Afterwards, you will learn how to write programs using multiple threads in BTnut, which enables running several tasks concurrently (*e.g.*, to compute intermediate results from sensor readings while communicating with neighboring nodes).

### 1 BTnut

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 additional device drivers provide access to BTnode-specific hardware. Overall, the BTnut OS architecture consists of three layers: the rudimentary C-libraries as implemented by `avr-gcc`'s `avr-libc`; the higher-level OS routines built on top of `avr-libc` by Nut/OS; and the BTnode-specific device drivers.

BTnut programs are written in C and compiled using `avr-gcc`, a freeware C compiler for the Atmel processor platform. Fortunately, `make btnode3` and `make btnode3 upload` provide a convenient interface for compiling and uploading to a BTnode. Here is a minimal BTnut program:

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

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

    for(;;)                       /* endless loop */
    {
        // do something clever here
    }

    /* main should NEVER return */
}
```

The `main` function is the first function that gets executed after powering up a BTnode. The *mandatory* call `btn_hardware_init()` initializes the BTnode hardware. As BTnodes are expected to continuously execute their task, the `main` routine should *never* return; otherwise, the behavior of a BTnode is undefined.

## Task 1: On-board LEDs

The BTnut library `<led/btn-led.h>` offers two functions for setting and unsetting the four BTnode LEDs: `void btn_led_set (u_char n)` and `void btn_led_clear (u_char n)`. The parameter `n` specifies the LED using a number (0, ..., 3) or a symbolic constant (`LED0`, ..., `LED3`). First, however, the LEDs need to be initialized by calling `btn_led_init(0)`.

Write a program that continuously cycles through the four LEDs (*i.e.*, it turns one after another on and off). Observe the output. Use `NutSleep(time)` from library `<sys/timer.h>` to slow down the execution until it becomes easily visible, where the unit of `time` is milliseconds.

### Solution

```
#include <hardware/btn-hardware.h>    /* for hardware init */
#include <led/btn-led.h>              /* for LEDs */
#include <sys/timer.h>                /* for NutSleep */

int main(void)
{
    u_char led = 0;
    btn_hardware_init();              /* init hardware */
    btn_led_init(0);                  /* init LEDs */

    for(;;)                            /* endless loop */
    {
        led = 0;
        while (led < 4)                /* cycle through all 4 LEDs */
        {
            btn_led_set(led);          /* turn LED on */
            NutSleep(500);              /* delay to make LED switch visible */
            btn_led_clear(led);        /* turn LED off */

            led++;
        }
    }
}
```

## Task 2: Timers

Timers provide an elegant way to schedule recurring function calls. You use timers in BTnut as follows:

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

HANDLE hTimer;                        /* the timer */

static void _tm_callback(HANDLE h, void* a) /* callback function */
{
    // do something when timer expires
}

int main (void)
{
    btn_hardware_init();                /* initialize SRAM */
    hTimer = NutTimerStart(3000, _tm_callback, NULL, 0); /* install the timer */
    for (;;) { NutSleep(1000); }        /* never end main */
}
```

NutTimerStart installs a timer. In particular, it sets a time interval (3000 ms), registers a callback function (static void \_tm\_callback(HANDLE h, void\* a)) that is executed when the timer (hTimer) expires, provides arguments to be passed to the callback function (here NULL as nothing is passed), and specifies whether the timer is periodic (0) or fires only once (TM\_ONESHOT).

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 should do the opposite (*i.e.*, turn on the red LED and switch off the blue LED).

### Solution

```
#include <hardware/btn-hardware.h>           /* for hardware init */
#include <led/btn-led.h>                     /* for LEDs */
#include <sys/timer.h>                       /* for timers and NutSleep */

HANDLE hTimer1, hTimer2;                   /* the timers */

static void _blue_on_red_off(HANDLE h, void* a) /* callback function 1 */
{
    btn_led_set(LED0);
    btn_led_clear(LED1);
}

static void _blue_off_red_on(HANDLE h, void* a) /* callback function 2 */
{
    btn_led_clear(LED0);
    btn_led_set(LED1);
}

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

    hTimer1 = NutTimerStart(1000, _blue_on_red_off, NULL, 0); /* install timer 1 */
    NutSleep(500);                          /* timer offset */
    hTimer2 = NutTimerStart(1000, _blue_off_red_on, NULL, 0); /* install timer 2 */

    for (;;) { NutSleep(1000); }            /* never end main */
}
```

### Task 3: Terminal

When testing a program, it can be extremely helpful to display debugging information via printf. To do this with a BNode, we have to route the output of printf through the BNode's serial port (*i.e.*, USB cable) to our host PC, where we observe the output in a so-called terminal program. On a Linux machine, you simply run minicom usb0 to setup a terminal connection to a BNode via port usb0; make sure the terminal session is configured to 57.6k, 8N1, no flow control. On the BNode, you have to route the standard output appropriately:

```
#include <stdio.h>
#include <dev/usartavr.h>

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);
}

int main(void)
{
    btn hardware_init(); /* initialize SRAM */
    init_stdout(); /* link printf output */
    int variable = 13;
    printf("Hello world, "); /* print something */
    printf("my lucky number is %d\n",variable);
    for (;;) /* main should never return */
}

```

After initializing the BTnode hardware, void `init_stdout(void)` does the desired output routing. Keep two things in mind: *i)* `printf` supports most standard conversion strings (e.g., `%s` and `%d`) and special characters (e.g., `\n`) but not float conversion (`%f`); *ii)* exit the terminal program (`minicom`) on the host PC before uploading a new program to the BTnode.

Augment your program from Task 2 by printing a debug message whenever one of the callback functions alters the state of a LED. Check whether the output correlates with the state of the BTnode's LEDs using the `minicom` terminal program.

## Solution

```

#include <hardware/btn-hardware.h> /* for hardware init */
#include <led/btn-led.h> /* for LEDs */
#include <sys/timer.h> /* for timers and NutSleep */
#include <stdio.h> /* for printf */
#include <dev/usartavr.h> /* for UART */

HANDLE hTimer1, hTimer2; /* the timers */

static void _blue_on_red_off(HANDLE h, void* a) /* callback function 1 */
{
    printf("Blue ON - Red OFF!\n"); /* debug output */
    btn_led_set(LED0);
    btn_led_clear(LED1);
}

static void _blue_off_red_on(HANDLE h, void* a) /* callback function 2 */
{
    printf("Blue OFF - Red ON!\n"); /* debug output */
    btn_led_clear(LED0);
    btn_led_set(LED1);
}

void init_stdout(void) /* link stdout to UART1 */
{
    u_long baud = 57600;

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

int main (void)
{
    btn hardware_init(); /* initialize hardware */
    btn_led_init(0); /* initialize LEDs */
    init_stdout(); /* route printf output */

    hTimer1 = NutTimerStart(1000, _blue_on_red_off, NULL, 0); /* install timer 1 */
    NutSleep(500); /* timer offset */
    hTimer2 = NutTimerStart(1000, _blue_off_red_on, NULL, 0); /* install timer 2 */
}

```

```
for (;;) { NutSleep(1000); } /* never end main */  
}
```

## 2 Programming with Threads

In BTnut, a *thread* is a function that runs concurrently to other threads. To support multithreading on a single core processor, the OS needs to repeatedly start and stop (*i.e.*, schedule) individual threads in a completely transparent fashion. BTnut uses *cooperative multithreading*; that is, each thread manually gives up control to the OS scheduler, which then decides which thread to run next based on priorities. While cooperative multithreading simplifies resource sharing and typically results in faster and smaller code (making it thus well suited to embedded systems programming), it entails the risk that a poorly designed thread brings the entire system to a halt.

Threads are functions. The main routine itself is also a thread that is started automatically after powering up the BTnode. You can define and create additional threads as follows:

```
#include <sys/thread.h>  
  
THREAD(my_thread, arg) /* define thread */  
{  
    for (;;) /* endless loop */  
    {  
        // do something  
    }  
}  
  
int main(void)  
{  
    if (NutThreadCreate("My Thread", my_thread, 0, 192) == 0) /* create and start thread */  
    {  
        // creating the thread failed  
    }  
  
    for (;;)   
    {  
        // do something  
    }  
}
```

The `THREAD` macro defines thread `my_thread`, where the `arg` parameter can be used to pass an argument of arbitrary type to the thread when it is created. As thread functions should never return, they loop endlessly (but you can explicitly call `NutThreadExit` to end a thread). The function `NutThreadCreate` creates and starts thread `my_thread`, gives it a name ("My Thread"), passes nothing to the thread (0), and specifies the size of the stack that is allocated for the thread (simply use 192 for now in your own code and you will be fine).

### Task 4: Running and Yielding Threads

Active threads only yield the CPU to other threads if they explicitly do so. Calling `NutThreadYield()` means: "Is there any thread more important than myself (*i.e.*, has higher priority)? If so, feel free to take

over control. Otherwise, I will continue.” Once control has been given away and is returned later on, the thread continues to run right after the call of `NutThreadYield()`. Alternatively, a thread can call `NutSleep(time)`. This puts the current thread to sleep and transfers control to a waiting thread. If no thread is waiting, the *idle* thread takes over (which is always waiting but with lowest priority).

Write a program that creates a single thread. This thread should repeatedly turn on the blue LED (LED0) and switch off the red LED (LED1). The main routine, after creating the thread, should do the opposite (*i.e.*, turn on the red LED and switch off the blue LED). Which LEDs are switched on? Why? Add a `NutThreadYield()` to the main routine such that the other LED is switched on. Add a second `NutThreadYield()` to your own thread such that both LEDs are switched on in turns. (You will see both LEDs switched on, because the main routine and the thread alternate very quickly.)

### Solution

```
#include <hardware/btn-hardware.h>           /* for hardware init */
#include <sys/thread.h>                       /* for threads */
#include <led/btn-led.h>                     /* for LEDs */

// without any NutThreadYield the red LED is on

THREAD(my_thread, arg)                       /* the thread */
{
    for(;;)                                  /* endless loop */
    {
        btn_led_set(LED0);                  /* turn on blue LED */
        btn_led_clear(LED1);               /* turn off red LED */
        NutThreadYield(); // second to add (both LEDs are on)
    }
}

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

    NutThreadCreate("My thread", my_thread, 0, 192); /* create and run the thread */
    for(;;)                                  /* endless loop */
    {
        btn_led_clear(LED0);                /* turn off blue LED */
        btn_led_set(LED1);                 /* turn on red LED */
        NutThreadYield(); // first to add (only the blue LED is on)
    }
}
```

## Task 5: Thread Priorities

In BTnut, each thread has a *priority* ranging from 0 (highest) to 254 (lowest). The *idle* thread has priority 254, whereas the main routine and 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**, several can either be **READY** or **SLEEPING**. A sleeping thread has ceded control either by calling `NutSleep(time)` or is waiting for an *event* (*e.g.*, an incoming packet or a message from another thread). 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 or equal priority, control is returned to the yielding thread immediately. Multiple threads with the same priority are executed in FIFO order.

You can assign the current thread a different priority as follows:

```

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

```

Take the program from Task 4 and give the self-created thread a higher priority. Compare the output with the original program from Task 4. Repeat the experiment giving the self-created thread a lower priority. What do you observe?

### Solution

```

#include <hardware/btn-hardware.h>           /* for hardware init */
#include <sys/thread.h>                     /* for threads */
#include <led/btn-led.h>                   /* for LEDs */

// without any NutThreadYield the red LED is on

THREAD(my_thread, arg)                     /* the thread */
{
    NutThreadSetPriority(20);               /* higher priority: red LED on */
    //NutThreadSetPriority(80);            /* lower priority: blue LED on */

    for(;;)                                /* endless loop */
    {
        btn_led_set(LED0);                 /* turn on blue LED */
        btn_led_clear(LED1);              /* turn off red LED */
        NutThreadYield();
    }
}

int main(void)
{
    btn hardware_init();                   /* initialize hardware */
    btn_led_init(0);                       /* initialize LEDs */

    NutThreadCreate("My thread", my_thread, 0, 192); /* create and run the thread */
    for(;;)                                /* endless loop */
    {
        btn_led_clear(LED0);               /* turn off blue LED */
        btn_led_set(LED1);                 /* turn on red LED */
        NutThreadYield();
    }
}

```

## Task 6: Events

Using *events*, threads can communicate with each other and coordinate their operation. For example, a thread can signal other threads to wake up after it has finished processing a particular data structure. The following code snippet demonstrates how threads post and wait for events in BTnut:

```

#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
    }
}

```

Thread `thread_B` executes some code and then posts an event via `NutEventPost(event)`. Thread `thread_A` also executes some code but then blocks in the `NutEventWait(event, timeout)` function. It only continues after `thread_B` has posted an event or the timeout expires. Here, the timeout is disabled by specifying an infinite time with the `NUT_WAIT_INFINITE` macro.

Write a program with three threads (main and two additional threads) and a global variable with initial value 2. The three threads should 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. After each computation, a thread prints its name and the result of the computation 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. Print appropriate messages to see when a thread terminates and when the main routine enters the loop. (**Hint:** A thread can access its own name using `runningThread->td_name` and can terminate itself through `NutThreadExit(void)`. To repeat the experiment, simply press the RESET button on the BTnode.)

## Solution

```

#include <hardware/btn-hardware.h>           /* for hardware init */
#include <led/btn-led.h>                     /* for LEDs */
#include <sys/timer.h>                       /* for NutSleep */
#include <sys/event.h>                       /* for events */
#include <sys/thread.h>                     /* for threads */
#include <stdio.h>                           /* for printf */
#include <dev/usartavr.h>                   /* for UART */

HANDLE event;                               /* the event */

static int value = 2;                        /* the globale variable */

THREAD(square_thread, arg)                  /* compute the square thread */
{
    for(;;)
    {
        NutEventWait(&event, NUT_WAIT_INFINITE); /* wait for event */

        if (value > 10000)                    /* should terminate itself? */
        {
            printf("%s: terminating\n", runningThread->td_name);
            NutThreadExit();
        }
        value = value*value;                  /* compute square */
        printf("%s: new value = %d\n", runningThread->td_name, value);
    }
}

```

```

        NutSleep(1000);                /* wait for 1 second to make
    }                                  output visible */
}

THREAD(decrement_thread, arg)         /* decrements by one thread */
{
    for(;;)
    {
        NutEventWait(&event, NUT_WAIT_INFINITE);    /* wait for event */

        if (value > 10000)                /* should terminate itself? */
        {
            printf("%s: terminating\n", runningThread->td_name);
            NutThreadExit();
        }
        value = value - 1;                /* decrement by one */
        printf("%s: new value = %d\n", runningThread->td_name, value);

        NutSleep(1000);                /* wait for one second to make
    }                                  output visible */
}

void init_stdout(void)                /* link stdout to UART1 */
{
    u_long baud = 57600;

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

int main(void)
{
    btn_hardware_init();                /* initialize hardware */
    init_stdout();                       /* route printf output */

    NutThreadCreate("Square", square_thread, 0, 192);    /* create & run square thread */
    NutThreadCreate("Decrem", decrement_thread, 0, 192); /* create & run decrement thread */

    while (value <= 10000) {            /* should enter endless loop? */
        value = value*2;                /* multiply by two */
        printf("%s: new value = %d\n", runningThread->td_name, value);

        NutSleep(1000);                /* wait for one second to make
    }                                  output visible */

        NutEventPost(&event);          /* post event to first thread */
        NutEventPost(&event);          /* post event to second thread */
    }

    NutEventPost(&event);                /* post event to let remaining
    }                                  thread terminate */

    printf("%s: entering endless loop %d\n", runningThread->td_name);
    for(;;)                              /* endless loop */
    {
        NutThreadYield();
    }
}

```