Embedded Systems FS 2017

Solution to Lab 1: Setup, Make, Assembler, LEDs

Discussion Dates: 08.03.2017 / 15.03.2017

Introduction

The **BTnode** is an autonomous wireless communication and computing platform based on a Bluetooth radio and a microcontroller. It serves as a demonstration platform for research in mobile ad-hoc networks and wireless sensor networks. The BTnode has been jointly developed at ETH Zurich by the Computer Engineering and Networks Laboratory (TIK) and the Research Group for Distributed Systems.

![Figure 1: The BTnode rev3](image)

BTnodes run an open source embedded operating system called Nut/OS. The Nut/OS embedded operating system is designed for the Atmel ATmega128 microcontroller (which is used on the BTnodes) and is intentionally a light-weight operating system. The Nut/OS includes the following features:

- Non-preemptive cooperative multi-threading
- Events
- Periodic and one-shot timers
- Dynamic heap memory allocation
- Interrupt driven streaming I/O

In order to use Nut/OS on the BTnodes, a set of BTnode-specific drivers and a Bluetooth stack for its on-board Bluetooth radio have been implemented, together forming the BTnut system software. More information about the BTnode and BTnut is available at the following website: [http://www.btnode.ethz.ch](http://www.btnode.ethz.ch)
Goals of this Session

In this session, we will start with setting up the development environment consisting of the Eclipse IDE and the BTnode hardware development kit. Then, your first task will be to upload and run the existing bt-cmd application. The bt-cmd application opens a command-line terminal on a serial console that allows you to access the Bluetooth radio of the BTnode.

Throughout the remainder of the session, you will learn about programming the BTnode using low-level register-based control of its peripherals. In contrast to the BTnut operating system, we use low-level programming to directly control the BTnode hardware. While it may not be so comfortable to program compared to a high-level object-orientated programming language, it gives you an insight into what is actually going on inside the microcontroller and how its attached peripherals are controlled.

Task 1: Setup of the BTnode Development Environment

Task 1.1: Hardware Setup

To setup the hardware, please follow these steps:

1. Open your development kit and check that it includes all parts that are shown in Figure 2.

2. Connect the USB programming board onto the BTnode. The edges of the programming board must be aligned with the edges of the BTnode. Compare your setup with Figure 3 by looking at the location of the connectors relative to the BTnode.

3. Connect the microcontroller programmer to the BTnode programming board and to your PC using the supplied cable.

4. Connect the BTnode programming board to your PC using the supplied USB cable.
Figure 3: BTnode with connected USB cable and ISP with serial cable. **Attention:** The USB programming board can be inserted in two ways. Make sure that the edges of the USB programming board are aligned with the edges of the BTnode.

**Task 1.2: Software Installation**

During this lab, we will use Eclipse for editing source code, a Linux terminal for running tasks like compiling software, `make` for scripting the binary build process, and `minicom` for connecting to the serial console of the BTnode. A cross-compiler and a linker for the ATMega128 microcontroller are provided by the GNU AVR toolchain. All mentioned components are already installed on your lab PC.

Relevant sources for the Embedded Systems lab are located at the following URL: 
http://www.tik.ee.ethz.ch/tik/education/lectures/ES/lab/lab-exercises.zip

Setup your software environment:

1. Open a Linux terminal
2. Use `wget` URL to download the software sources to your home directory
3. Start Eclipse
4. Let Eclipse create a workspace directory, if you do not already have one. From now on, this location is called `$WORKSPACE`.
5. Check that you have opened the C/C++ perspective, switch to this perspective if not
6. Import the downloaded archive by running `File > Import > General > Existing Projects into Workspace > Select Archive File`
7. You have now imported the Eclipse project es. From now on, the location `$WORKSPACE/es/btnut` is called `$BTNUTDIR`.

**Task 2: Exploring the BTnode**

We will now explore the possibilities of the BTnode with the already existing `bt-cmd` demo application. The `bt-cmd` demo application is a brief example of how to use the Bluetooth radio and protocol stack.
First, we have to install `bt-cmd` on the BTnode:

1. Use your Linux terminal to navigate to `$BTNUTDIR/app/bt-cmd`
2. Run `make btnode3` to compile the binary image
3. Upload the binary image to the BTnode by running `make btnode3 upload`

Now, we use the `minicom` terminal application to connect to the command-line terminal running on the BTnode. The BTnode console is usually connected to `/dev/ttyUSB0` of your Linux machine. The serial terminal session must be configured to `57.6k, 8N1, no flow control`.

1. Run `minicom usb0` in the Linux terminal
2. Press the reset button on the BTnode (see Figure 4)
3. Hit `Tab` twice to get a selection of possible commands (see Figure 5)
4. You can leave Minicom by pressing `Ctrl A-X` (press `Ctrl A` followed by entering `X`)

---

**Figure 4: BTnode reset button**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bt</code></td>
<td>bluetooth radio commands</td>
</tr>
<tr>
<td><code>led</code></td>
<td>toggle LED patterns</td>
</tr>
<tr>
<td><code>bat</code></td>
<td>get the battery status</td>
</tr>
<tr>
<td><code>nut</code></td>
<td>show OS system information</td>
</tr>
<tr>
<td><code>log</code></td>
<td>BTnut logging features</td>
</tr>
</tbody>
</table>

**Figure 5: There are NutOS/BTnode and Bluetooth specific commands (if called without arguments they will show hints on the correct syntax, where applicable).**

You are now ready to solve the following questions:

a) Enable the four LEDs on the BTnode using the `led` command
b) Try to locate different BTnodes by issuing `bt inquiry sync`. 
Task 3:  Blink Application

Our first application for the BTnode will be Blink. As the name says, it does nothing more than periodically toggling the LEDs on the BTnode. The reason for this choice is that for any further work with the BTnodes, we need some kind of feedback from the programs we implement. In general, LEDs offer valuable feedback when programming embedded devices that often have no other form of hardware available for visual interaction with the programmer.

Task 3.1: Accessing LEDs on the BTnode

The LEDs are an off-chip resource. Unfortunately, accessing the LEDs is a bit tricky and requires some knowledge about the design of the BTnode: The address bus of the ATmega128 is 16 bit wide and it is mapped to the ports A (lower 8 bits) and C (upper 8 bits). You can see this on the “BTNODE CORE” page of the BTnode schematics. The address bus is mainly needed to access the external SRAM (AMIC_LP62S2048), but at the same time it is also connected to a latch (TI_SN74LVC573A) multiplexing the LEDs and other outputs (see “BTNODE POWER I/O” page of BTnode schematics). To set or clear LEDs, the bits that determine whether the LEDs should be on or off have to be put on the address bus. Then the latch is enabled, i.e. it samples the value on the address bus. After a while, the latch is disabled, i.e. it holds the previously sampled value. The C code and the corresponding Assembler program listed in Figure 6 do exactly this.

Take a look at the different accesses to hardware registers. First, the SRE (extern SRAM enable) bit in the MCUCR (MCU General Control Register) register must be set to tell the processor that there is an external SRAM bus. Before the latch can be used, the control line for controlling the write access must be configured. This is done by defining pin 5 as an output in the data direction register DDRB. While this is the minimum setup for accessing our off-chip LEDs, more initialization code is necessary for running more complex functions.

The full instruction set of the AVR microcontroller can be accessed at $WORKSPACE/es/doc/avr_instructionset.pdf

You can access existing source code templates under $BTNUTDIR/ex/lab1-blinkasm and $BTNUTDIR/ex/lab1-blinkc. Run make in these directories to compile the source code and make upload to upload the compiled code to the BTnode.

Please work on the following tasks:

a) Enable the blue LED on the BTnode. Read the BTnode schematics to find out which address must be set on the address bus. You need to change only one line of code in both source code examples.

b) In the $BTNUTDIR/ex/lab1-blinkasm folder, run avr-objdump -d blink.S.o. Open another terminal and run avr-objdump -d blink.S.elf in the same folder. Compare the both outputs. Which parts are equal, how do both outputs differ? Why?

c) Compare the output of avr-objdump -d blink.S.o from the $BTNUTDIR/ex/lab1-blinkasm folder with the output of avr-objdump -d blink.c.o from the $BTNUTDIR/ex/lab1-blinkc folder. Actually, both pieces of code should fulfill the same function. Why is the output of avr-objdump -d blink.c.o larger?

Solution a)

For enabling the blue LED, the address 0x0100 must be used.

---

1The schematics are available in $WORKSPACE/es/doc/btnode_rev3.22_schematic.pdf
```c
#include <avr/io.h>

void init(void) {
    // enable external memory bus
    MCUCR |= 1<<SRE;
    // set latch pin as output
    DDRB |= 1<<DDB5;
}

int main(void) {
    volatile char * pointer;
    char dummy;
    init();
    // compute the pseudo-address that
    // contains the values for the LEDs
    pointer = (char *) (((short)0x2) << 8);
    // force the compiler to write this
    // pseudo-address to the address-bus
    dummy = *pointer;
    // now enable the latch
    PORTB |= 1<<PB5;
    // wait a moment - volatile keeps the
    // compiler from removing this line
    asm volatile("nop");
    // disable the latch, i.e. hold the value
    PORTB &= ~(1<<PB5);
    return 0;
}
```

```assembly
.global main

init:
    ; enable interface for external memory
    ldi r24, 0x80 ; SRE
    out _SFR_IO_ADDR(MCUCR), r24
    ; set latch pin as output
    sbi _SFR_IO_ADDR(DDRB), DDB5
    ret

main:
    ; set pseudo-address on bus
    lds r24, 0x0200
    ; set latch
    sbi _SFR_IO_ADDR(PORTB), PB5
    ; wait for one cycle
    nop
    ; clear latch
    cbi _SFR_IO_ADDR(PORTB), PB5
    ret
```

Figure 6: C and Assembler code of the first Blink application.
lds r24, 0x0100

pointer = (char *) (((short)0x1) << 8);

Solution b)
The output of avr-objdump -d blink.S.o is equal to the written Assembler code. In contrast, blink.S.elf is already fully linked and also contains an interrupt vector that defines to which address the program counter has to be set when a specific interrupt occurs. The added instructions are provided by the AVR libraries.

Solution c)
The compiled C program needs already more space compared to the bare Assembler code. First, the C compiler added code for initializing the stack pointer. Second, the return value is saved to registers before the function finally returns.

Task 3.2: Period Toggling of BTnode LEDs

Our LEDs are now turned on, but they do not blink. Implement a blinking LED in both C and Assembler so that the LED periodically changes its state.

Figure 7: C and Assembler example for busy waiting.

a) Implement periodic toggling of the BTnode LEDs both in your C and your Assembler program. One can leverage the busy waiting code segments listed in Figure 7.

b) Estimate the period in which the LED is toggled in the Assembler version. Calculate the period length as accurate as possible by analyzing the Assembler code. BTnodes are running at 8 MHz, use the AVR Instruction Set Manual for reference on execution times.

c) The given Assembler code for busy waiting is based on three nested loops. What could be done to use less loops but still keeping the code size small? Think about the location of the counter variables in the example.
Solution b)

The following execution times are given from the AVR Instruction Set Manual:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>brne (taken)</td>
<td>2 cycles</td>
</tr>
<tr>
<td>brne (not taken)</td>
<td>1 cycle</td>
</tr>
<tr>
<td>ldi</td>
<td>1 cycle</td>
</tr>
<tr>
<td>rcall</td>
<td>3 cycles</td>
</tr>
<tr>
<td>ret</td>
<td>4 cycles</td>
</tr>
<tr>
<td>subi</td>
<td>1 cycle</td>
</tr>
</tbody>
</table>

This yields the following formula:

\[
\begin{align*}
    c_{\text{inner}} &= 1 + l_{\text{inner}} \cdot (1 + 2) - 1 \\
    c_{\text{middle}} &= 1 + l_{\text{middle}} \cdot (c_{\text{inner}} + 1 + 2) - 1 \\
    c_{\text{outer}} &= 1 + l_{\text{outer}} \cdot (c_{\text{middle}} + 1 + 2) - 1 \\
    c_{\text{total}} &= l_{\text{outer}} \cdot (3 + (l_{\text{middle}} \cdot (3 + l_{\text{inner}} \cdot 3))) + 7
\end{align*}
\]

Here, \( c_{\text{inner}} \) etc. denote the number of cycles, while \( l_{\text{inner}} \) etc. describe the number of iterations in the specific loop level. For the given parametrization from the source code, this results in 8,181,547 cycles or approximately 1.02 seconds for each call of the wait function.

Solution c)

In the given code, counter variables are stored in 8 bit registers. With two loops, the maximum number of iterations is \( 255 \cdot 255 = 65025 \). This is too small for waiting approximately 8 million cycles. Counter variables larger than 8 bit could be stored in the RAM. However, more assembler instructions are necessary for working with those counters.

Solution source code
```c
#include <avr/io.h>

void init(void) {
    // enable external memory bus
    MCUCR |= 1<<SRE;
    // set latch pin as output
    DDRB |= 1<<DDB5;
}

void write_led(short value) {
    volatile char * pointer;
    char dummy;
    // compute the pseudo-address that contains the values for the LEDs
    pointer = (char *) (((short)value) << 8);
    // force the compiler to write this pseudo-address to the address-bus
    dummy = *pointer;
    // now enable the latch
    PORTB |= 1<<PB5;
    // wait a moment - volatile keeps the compiler from removing this line
    asm volatile("nop");
    // disable the latch, i.e. hold the value
    PORTB &= ~(1<<PB5);
}

void waitabit() {
    int i, j;
    for(i = 0; i <= 1200; i++) {
        for(j = 0; j <= 1200; j++) {
            // wait a moment - volatile keeps the compiler from removing this line
            asm volatile("nop");
        }
    }
}

int main(void) {
    init();
    for(;;) {
        write_led(0x1);
        waitabit();
        write_led(0x0);
        waitabit();
    }
    return 0;
}
```

Figure 8: Code of extended Blink C application
Figure 9: Code of extended Blink Assembler application