BTnode Programming
— An Introduction to BTnut Applications

Number 1.3

Jan Beutel, Philipp Blum, Matthias Dyer, Clemens Moser
Computer Engineering and Networks Laboratory
ETH Zurich
8092 Zurich, Switzerland
{beutel,blum,dyer,moser}@tik.ee.ethz.ch

with contributions of

Marc Langheinrich, Jonas Wolf
Institute for Pervasive Computing
ETH Zurich
8092 Zurich, Switzerland
{langhein,wolfj}@inf.ethz.ch

The BTnode Project
May 29, 2006

Contents

Date	**Section**	**Who**	**Changes**
Mar 2, 2005		jb	Initial Version 0.1
Mar 24, 2005	2, 3	jb, pb	Initial drafts of 2st and 2nd exercise done
Mar 30, 2005	3	jb	Added input from beta-testers, ready for distribution
May 10, 2005	1, 6	jb, cm	Edits after Clemens first import, added BTnode developer kit to introduction
May 12, 2005	all	jb,cm,md	Fixed graphics and rest of 5 and 6, prep for first release
Mar 21, 2006	1	jb	Minor updates
Mar 24, 2006	1,2,appendix	jb	Finished chapter 1 and 2 draft for SS2006
Apr 5, 2006	2	jw	Added bootloader
Apr 5, 2006	1,2,4,appendix	jb	Minor changes to chapter 2. Checked all solutions and moved some files around
Apr 6, 2006	1	ml	Added Marc's superb introduction modification
Apr 19, 2006	2	jb	Replaced 1152000 by 57600 (default baudrate)
May 29, 2006	6	jb,cm	Bluetooth updates for 2006, created version 1.3

Table 1: Revision History

Chapter 1

Introduction

1.1 The BTnodes and the BTnut System Software

Figure 1.1: The BTnode rev3.

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 and ad-hoc connected networks (MANETs) and distributed 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. Currently, the BTnode is primarily used in the NCCR-MICS research projects[1]

In addition to its Bluetooth radio, the latest BTnode revision (rev3) also features a low-power radio identical to the one used on the Berkeley Mica2 Motes[2], allowing it to interact with both Mica2-based nodes and previous, Bluetooth-only revisions of the BTnode. Both radios can be operated simultaneously or be independently powered off completely when not in use, considerably reducing the idle power consumption of the device.

BTnodes run an embedded systems OS from the open source domain, called Nut/OS[3] Nut/OS is designed for the Atmel ATmega128 microcontroller (which is used on the BTnodes) and intentionally kept very simple. According to the Nut/OS homepage, it features:

- Non preemptive cooperative multi-threading
- Events
- Periodic and one-shot timers

---

[1] See `www.mics.org`.

[2] See `www.xbow.com/Products/Product_pdf_files/Wireless_pdf/6020-0042-05_A_MICA2.pdf`.

[3] See `www.ethernut.de/en/software.html`.

- Dynamic heap memory allocation
- Interrupt driven streaming I/O

In order to use `Nut/OS` on the BTnodes, a set of BTnode-specific drivers have been added, and in particular a Bluetooth stack for its on-board Bluetooth radio. These three pieces form together the *BTnut system software.*

In this tutorial, we will learn how to use the BTnut system software to deploy sensor node applications on the BTnode wireless sensor node platform.

1.2 Intended Audience

This tutorial originated in the Embedded Systems lecture, a graduate course taught at the Department of Information Technology and Electrical Engineering, ETH Zurich. It requires basic knowledge of C-programming and embedded systems and should give an overview of the capabilities of networked embedded systems and their key properties. However, apart from its usage in the lecture, this tutorial provides a basic introduction to programming on the BTnode platform, so it should also be beneficial to the occassional computer scientist not versed in all things electrical.

Each chapter comes with a set of exercises that are supposed to get you accustomed to basic, everyday tasks of an embedded engineer. The order in which the exercises are performed is not of crucial importance, and whole chapters can be left out to suit the individual needs (e.g., computer scientists might want to skip those concerning hardware issues). However, we suggest that you perform the exercises in the order given to minimize unforeseen complications.

1.3 Hard- and Software Requirements

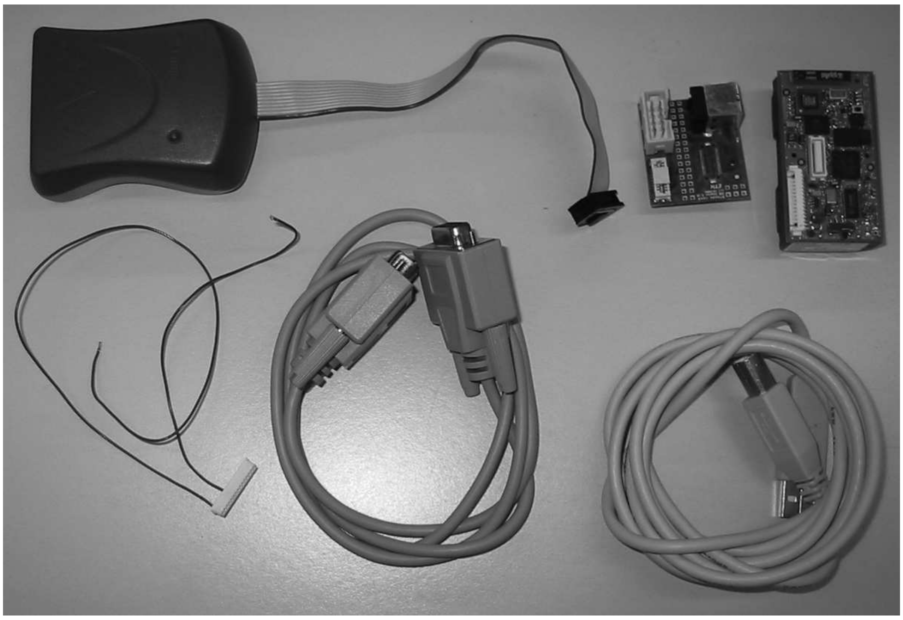


Figure 1.2: The BTnode development kit. The minimal set of tools consists of the three items on the very right: a BTnode, a USB programming board, and a USB cable. Additionally, some exercises require the use of an ISP programmer, a serial cable, and a 15-Pin Molex breakout cable (left half).

To be able to do all of the practical exercises in this tutorial, you will need a complete BTnode developer kit (see Figure 1.2) consisting of: a BTnode rev3; a usbprog USB programming adapter; an ISP programmer (we suggest the Atmel ATAVRISP or alternatively the ATAVRISP MK2 programmer); serial and USB cables; a 15-Pin Molex breakout cable; and the software, documentation and tools contained on the BTnode CDROM (see Figure 1.3). However, a number of exercises can also be performed with a minimal subset of these tools, namely a BTnode, the USB programming adapter, and a USB cable.

For a complete listing of software tools and their versions used in this tutorial, please see appendix A. The tutorial assumes that the necessary development tools (avr-gcc toolchain, avr-libc, an ISP programming utility if you use the ISP programmer, eclipse and CDT) are installed and working correctly. For details on the installation and configuration of the development tools see the BTnode online resources available at `www.btnode.ethz.ch`.

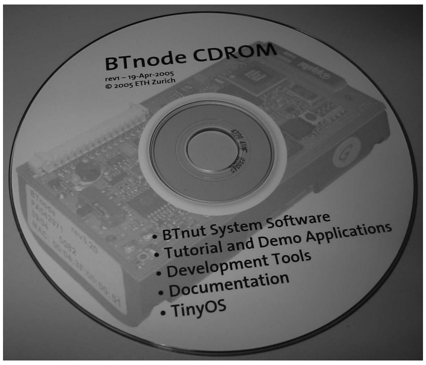


Figure 1.3: The BTnode CDROM.

1.4 Reference Documents

Should you ever need more information than what is given here in this tutorial, feel free to browse the following sites for details on the individual pieces of the puzzle:

- **The BTnode platform reference** – with support documents, installation instructions for the development tools and source software, mailing lists and various links.

  `www.btnode.ethz.ch`
- **The home of Nut/OS** – the BTnut operating system core.

  `www.ethernut.de`
- **Open source development tools for the AVR platform**

  `www.openavr.com`
- **Open source tools for the development on Atmel AVR, Windows platform installer**

  `winavr.sourceforge.net`
- **Atmel AVR product family**

  `www.atmel.com/products/avr`
- **Atmel AVR related developer information** – application notes, links and tools.

  `www.avrfreaks.net`
- **A nice avr-gcc tutorial** (in german)

  `www.mikrocontroller.net/wiki/AVR-GCC-Tutorial`

- **Bluetooth Special Interest Group** – all about the standardization, applications and reference documents.

  `www.bluetooth.org`

- **Technical BTnode/BTnut support** – For technical questions concerning BTnut and the BTnode platform please inquire to the mailing list:

  `mailto:btnode-development@list.ee.ethz.ch`

Chapter 2

First Steps in BTnode Programming

2.1 Introduction

In this chapter, we will step you through the basic knowledge about development tools, software structure and reference documentation necessary to start developing your own applications on the BTnode platform. This is explained, using a pre-configured toolchain setup on Windows, although other host platforms and tool setups are possible too (Linux and MacOS X). For detailed instructions on the tool installation, please refer to the online documentation and links listed under section 1.4 and the software versions listed in appendix A.

2.2 Development Tools

For basic software development you will need an editor, a compiler-assembler-linker toolchain, a standard library and an in-system programming software to upload the compiled program to your embedded target. There are many other tools that can make life easier when projects are getting larger and debugging more difficult. The selection of tools introduced here should provide you with a basic overview and understanding to define the right set of tools for your personal project needs.

2.2.1 Compilation

The tools introduced here are freely available and are based on GNU GCC and the AVR libc which is a Free Software project whose goal is to provide a high quality C library for use with GCC on Atmel AVR microcontrollers. Together, avr-binutils, avr-gcc, and avr-libc form the heart of the Free Software toolchain for the Atmel AVR microcontrollers. They are further accompanied by projects for in-system programming software (uisp, avrdude), simulation (simulavr) and debugging (avr-gdb, avr-insight, AVaRICE).

These tools are available packaged as a Windows installer in the WinAVR project which we will use as a reference. There are numerous other distributions of the avr-gcc toolchain available as well as different (commercial) compilers for the Atmel AVR family.

A thorough introduction to the internals of such a compiler toolchain as used in embedded systems can be found in Appendix A: Assemblers, Linkers and the SPM Simulator of [3]. Manuals for the avr-binutils, avr-gcc and avr-libc are packaged with the respective distribution or available online (see section 1.4).

The following example illustrates a sample compilation, linkage with startup code and libraries as well as transformation into a machine uploadable format of a sample application called test.c:

```
avr-gcc -c -mmcu=atmega128 -D__BTNODE3__-I../../include test.c -o test.btnode3.o
avr-gcc test.btnode3.o  ../../lib/btnode3/nutinit.o -L../../lib/btnode3 -mmcu=atmega128 -o test.btnode3.elf
avr-size test.btnode3.elf
   text    data     bss     dec     hex filename
  36920    1708     314   38942    981e test.btnode3.elf
avr-objcopy -O ihex test.btnode3.elf test.btnode3.hex
```

2.2.2 Simulation and Debugging

When project size increases and especially in critical situations specialized simulation and debugging tools can be of great benefit. There are numerous tools available (avr-gdb, JTAG tools, Atmel AVR Studio, GNU dwarf parser, avr-insight, Avrora, simulavr) serving different purposes, of which a selection will be introduced in chapter 5.

2.2.3 Project Management

The basic utility used in most build environments is GNU make. The make utility automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them. This is a very convenient way to avoid retyping long lines of parameters on the command line.

Different editors with syntax higlighting and project management features can be used for C based AVR development. The most common are Eclipse, Emacs, Programmers Notepad and AVR Studio. Especially Eclipse in conjunction with CDT (C/C++ Development Tools) is a very powerful tool that allows C-indexing, project management, integration of a make build environment, debugging, version control and much more.

Version control such as with CVS (Concurrent Version System) or Subversion is helpful for keeping track of changes and sharing source code among team members.

2.2.4 Embedded Target Connection

The software on an embedded system is typically programmed once during manufacturing onto a resident internal memory from where it is then executed. Software changes are frequent during development but infrequent during the lifetime of a product.

For uploading code to the flash memory of the ATmega128l (in-system programming) a serial uploader software (uisp, avrdude, uploader tools in AVR Studio) and an appropriate programmer (hardware) is necessary.

Although basic debugging can be performed via general purpose IOs and LEDs, verbose terminal output is generally preferred. For this a RS-232C serial connection is necessary between the embedded target (BTnode) and a PC. This can be done using a serial level shifter (e.g. Maxim MAX3232) or a USB-serial converter (e.g. Silabs CP2101).

In addition to uploading code using in-system programming as described above, the ATmega128l features a bootloader section as well as JTAG uploading and debugging support (see chapter 5 for further information on JTAG). The bootloader section in the flash memory can be used to re-program the user section of the flash memory once such a bootloader has been installed. See exercise 14 for further information.

2.2.5 Documentation Tools

The primary source for information for any hard- or software system are its manuals, typically accompanied by release information, changelogs, readme file and known errata.

The internet is a general resource for developers and project management. More specific mailing lists and archives offer discussion forums on specific topics, such as the avr-libc library usage and development or on BTnode specific issues.

Large online project management such as http://www.sourceforge.net offer many services such as electronic bug tracking systems, version control, web visualization, nightly builds, software distribution and general project management.

Single projects typically extract documentation from source code. This can be done by tools such as javadoc or doxygen to automatically generate up-to-date online documentation.

2.3 Notes on the BTnode Hardware Architecture

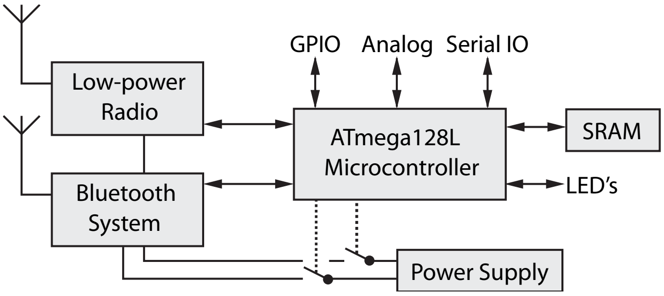


Figure 2.1: BTnode rev3 hardware overview.

System Core – The BTnode System Core consists of an Atmel ATmega128l microcontroller, clocks and SRAM memory.

- Atmel ATmega128l – 4 kB EEPROM, 64 kB SRAM, 128 kB Flash
- System clock – 32 kHz real time clock and 7.3728 MHz system clock
- 5 processor power modes
- External data cache – 3x60 kByte low power SRAM
- Four LED's for easy debugging
- In-system programming through serial ISP programmer, JTAG or resident bootloader

Bluetooth Radio – Zeevo ZV4002 Bluetooth radio running HCI firmware. It is connected to the ATmega128l through a UART interface.

Low-Power Radio – Chipcon CC1000 radio operating at 868 MHz. Other operating frequencies can be used according to the CC1000 documentation (433-915 MHz). Both an integrated monopole antenna, an external wire and an external coaxial connector (MMCX type) are possible though assembly options. The default assembly variant is the internal monopole antenna and operation in the 868 MHz ISM band.

Power Supply – The standard power supply are 2-cell AA batteries. The common range for these is 2-3 V DC when either primary or rechargeable batteries are used. The primary boost converter has a nominal input range of 0.5-3.3 V DC. Alternatively 3.6-5 V can be supplied through the VDC_IN pin available on the external connectors J1 and J2.

- Primary supply – Linear Technologies LTC3429, 600mA max., input 0.5-3.3 V to 3.3 V
- Alternate supply – Linear Technologies LT1962, 300mA max., input 3.6-5.0 V to 3.3 V
- Switchable power-groups for IO, Bluetooth and LPR radio
- Battery charge indicator
- On/Off switch for the primary power supply

A detailed hardware reference is available though the BTnode website (see section 1.4).

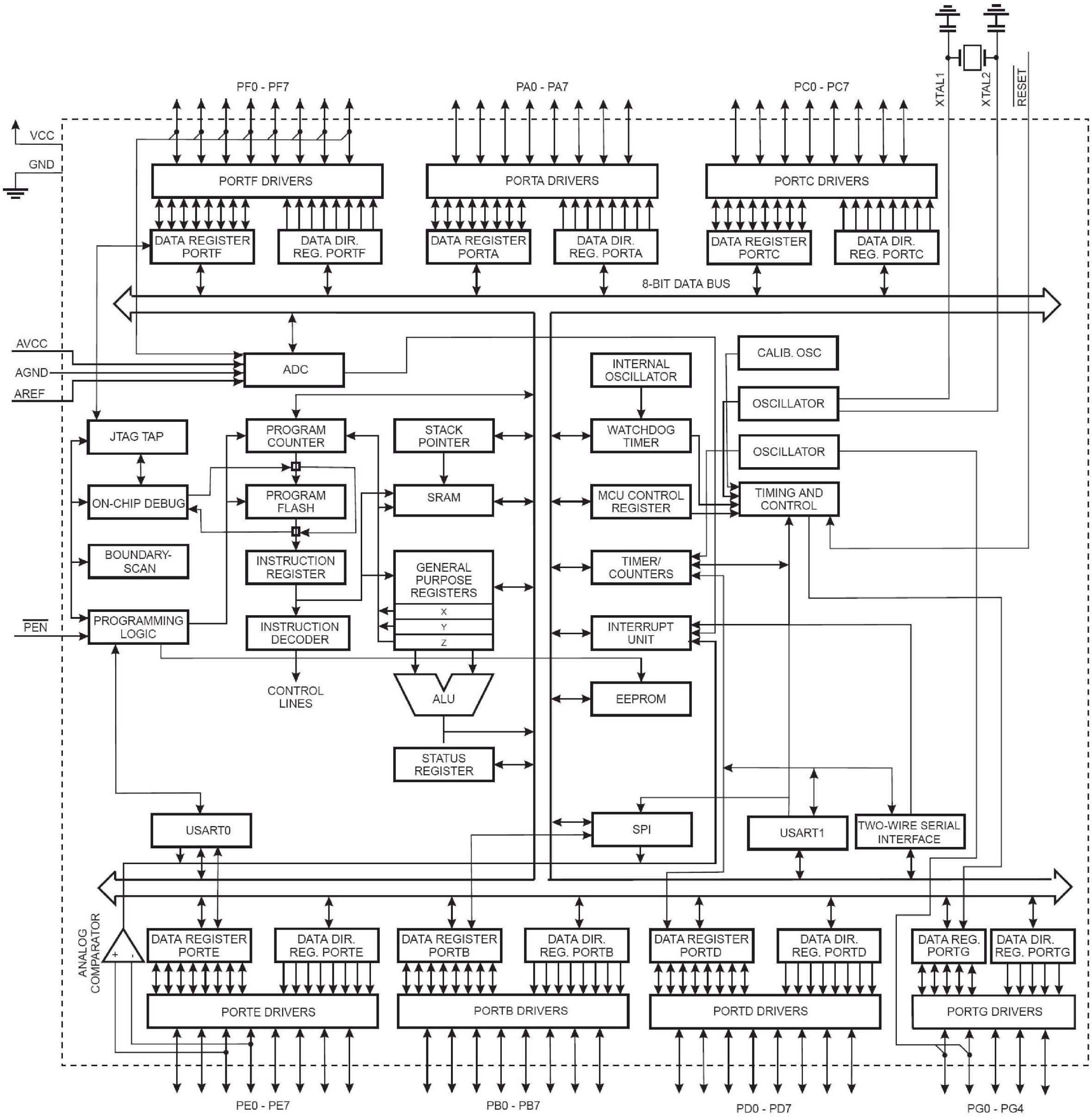


Figure 2.2: Atmel ATmega128l microcontroller core and peripheral block diagram.

Exercise 1 *Find the BTnode rev3.20 Schematic and the ATmega128l Processor Manual pdf files [1]. Browse the schematic and find the latch (Texas Instruments SN74LVC573A) used to multiplex the extended SRAMs (AMIC LP62S2048) data and address bus. Which ports of the processor are used to connect to the latch? Which ports are used to connect to the memory?*

Browse for the second latch used to multiplex the LEDs and switchable power supplies. Which port/pin on the ATmega128l maps to which function (LED/power switches) here? Which are the control lines used for the latch? Draw a sample output waveform for the microcontroller pins used, that switch the LEDs on and off.

What are the problems arising from this hardware setup for a software system, especially in the case of an operating system with concurrency (multiple drivers/tasks/threads)? How would you implement a software driver for this functionality? Why is SBI/CBI (set bit and clear bit) not sufficient in this case?

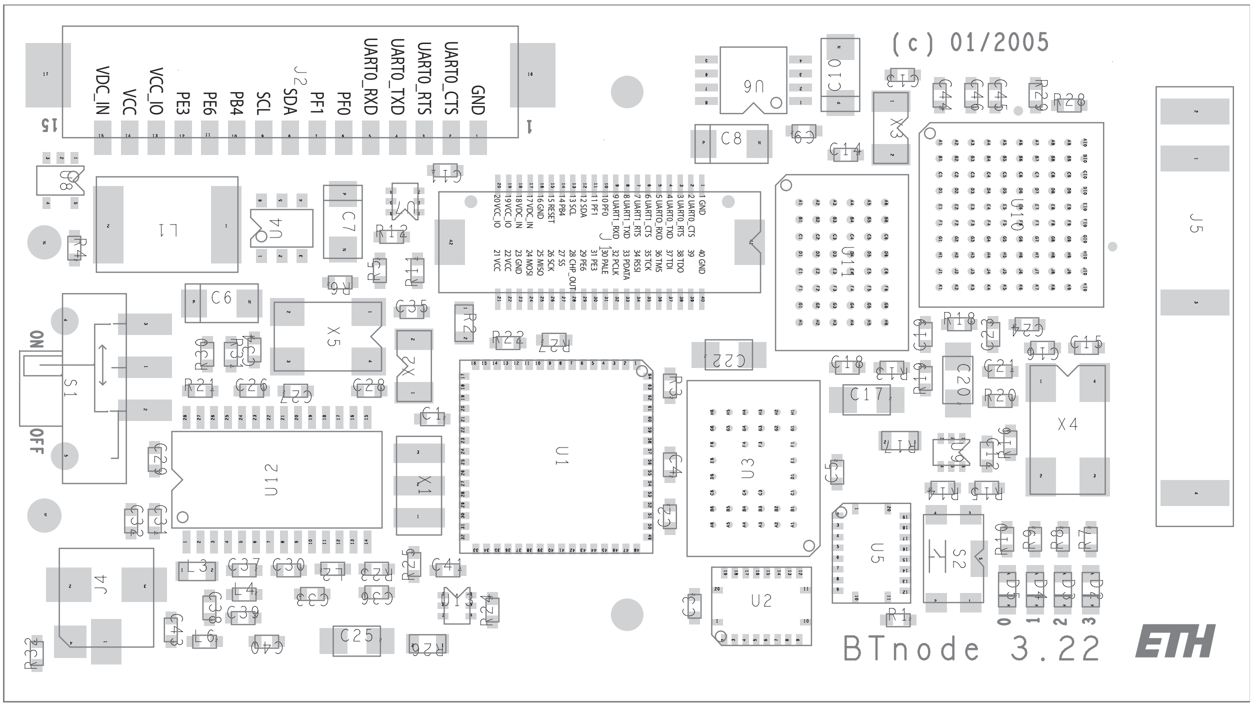


Figure 2.3: BTnode rev3 top assembly and connector pinout.

2.4 BTnut System Software Resources

First, we will make you familiar with the development environment and the tool flow. The exercises in this section are based on using Eclipse and CDT, yet they can also be performed using other project management environments and editors.

Explanation ***Getting to know the BTnut system software release*****:**
The BTnut software is released in both a binary snapshot and sourcecode format. The most recent releases can be downloaded from sourceforge.net.

- The `btnode_snap_btnode3_binary` contains an out-of-the-box pre-compiled library package for AVR binary and documentation, ready for usage with the avr-gcc toolchain and the demo applications included.
- The package `btnut_system` contains all BTnut and Nut/OS sources. It requires to compile the BTnut system software and install the documentation prior to the compilation of applications.

The releases are numbered even and are based on the following CVS tag and date:

```
BTnut snapshot and release -- REL_VERSION = 1.6

Nut/OS -- NUT_SNAPSHOT = 200X-XX-XX
```

and compiles against the following avr libc:

```
AVR Libc -- avr-libc 1.4.3
```

The BTnut pre-compiled snapshot contains 5 directories, `app` for the applications, `doc` for documentation, `extras` for hardware specific drivers other than the BTnode, `include` for all headerfiles and `lib` for the pre-compiled libraries.

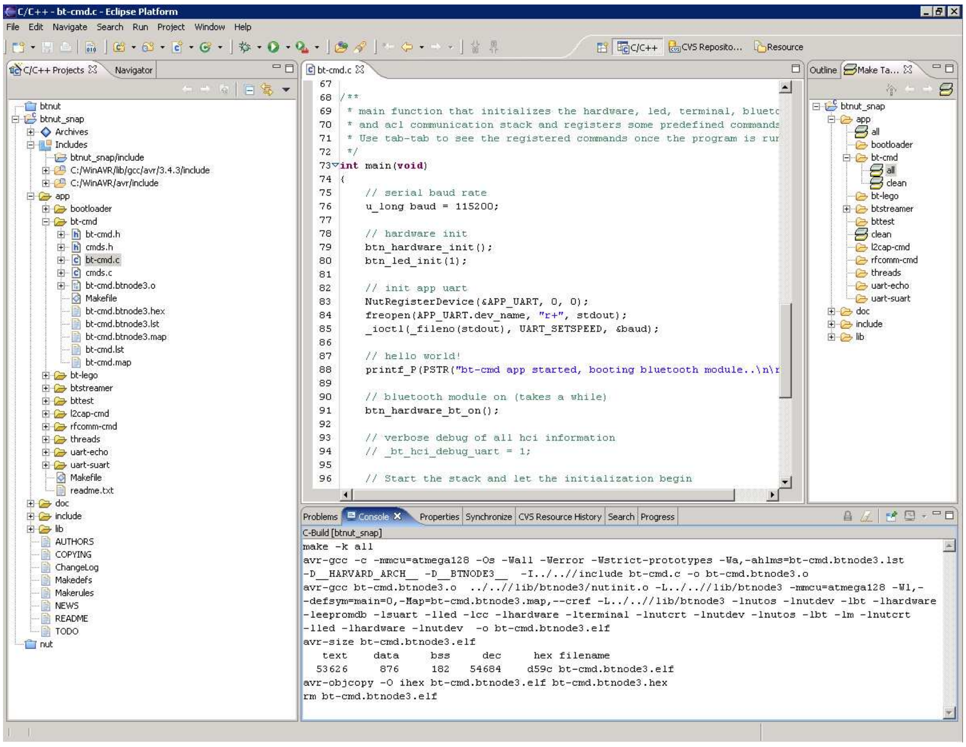
The first task to be performed on the BTnut system software will be to set up a working environment within Eclipse.

Exercise 2 *Open the C/C++ perspective in Eclipse. Create a new project called* `btnut_snap_X.X` *by selecting "Standard Make C Project" from the pull-down menu. Be sure to set the correct binary parser on the second screen of the new project wizard (select ELF parser and GNU ELF parser, and enter* `avr-addr2line` *and* `avr-c++filt`*) and set the correct compiler (*`avr-gcc`*) in the discovery options tab to select the correct cross-development tools for the AVR platform.*

Now import the `btnode_snap_btnode3_binary` *package by selecting* `Import`, `Archive File` *into this project. As a final task, you will need to configure the project with the correct include paths for the C/C++ parser: Open the project properties and insert the* `btnut_snap/include`, `$(PATH_TO_AVR_GCC_TOOLS)/avr/include` *and* `$(PATH_TO_AVR_GCC_TOOLS)/lib/gcc/avr/3.4.5/include` *to the projects include paths.*

Exercise 3 *Open the* `bt-cmd.c` *file in the* `app/bt-cmd` *folder and go to the line where* `btn_led_init(1);` *is called. Highlight the function name, then press* `F3` *to open the functions* `Declaration` *from the appropriate header file. Right click the function name again and search for* `All References` *in the* `Workspace`*.*

Be sure to switch to the `C/C++ Perspective` *in Eclipse and open the* `C/C++ Projects View` *(see figure 2.4).*



```
/**
 * main function that initializes the hardware, led, terminal, bluetc
 * and acl communication stack and registers some predefined commands
 * Use tab-tab to see the registered commands once the program is rur
 */
int main(void)
{
    // serial baud rate
    u_long baud = 115200;

    // hardware init
    btn_hardware_init();
    btn_led_init(1);

    // init app uart
    NutRegisterDevice(&APP_UART, 0, 0);
    freopen(APP_UART.dev_name, "r+", stdout);
    _ioctl(_fileno(stdout), UART_SETSPEED, &baud);

    // hello world!
    printf_P(PSTR("bt-cmd app started, booting bluetooth module..\n\r

    // bluetooth module on (takes a while)
    btn_hardware_bt_on();

    // verbose debug of all hci information
    // _bt_hci_debug_uart = 1;

    // Start the stack and let the initialization begin
```

```
make -k all
avr-gcc -c -mmcu=atmega128 -Os -Wall -Werror -Wstrict-prototypes -Wa,-ahlms=bt-cmd.btnode3.lst
-D__HARVARD_ARCH__ -D__BTNODE3__  -I../..//include bt-cmd.c -o bt-cmd.btnode3.o
avr-gcc bt-cmd.btnode3.o  ../..//lib/btnode3/nutinit.o -L../..//lib/btnode3 -mmcu=atmega128 -Wl,-
-defsym=main=0,-Map=bt-cmd.btnode3.map,--cref -L../..//lib/btnode3 -lnutos -lnutdev -lbt -lhardware
-leepromdb -lsuart -lled -lcc -lhardware -lterminal -lnutcrt -lnutdev -lnutos -lbt -lm -lnutcrt
-lled -lhardware -lnutdev  -o bt-cmd.btnode3.elf
avr-size bt-cmd.btnode3.elf
   text    data     bss     dec     hex filename
  53626     876     182   54684    d59c bt-cmd.btnode3.elf
avr-objcopy -O ihex bt-cmd.btnode3.elf bt-cmd.btnode3.hex
rm bt-cmd.btnode3.elf
```

Figure 2.4: The C/C++ perspective with the C/C++ Projects view on the left, a file editor in the top, console view in the bottom middle and the Make Target view open on the right.

Exercise 4 *Open the BTnut System Software Reference (online version: BTnut API on http://www.btnode.ethz.ch, or local in* `doc/html`*) in a web browser and open the file*

`btnode/include/led/btn-led.h`

from the `File List`*. Read the documentation provided for the* `btn_led_init()` *and* `btn_led_add_pattern()` *functions.*

Exercise 5 *Go back to the* `bt-cmd.c` *file and add a new led pattern for the LED heartbeat using* `btn_led_add_pattern` *in line 103. While typing the function name* `btn_led_add_pattern` *press* `CTRL-SPACE` *to invoke Eclipse's* `Content Assist` *function and complete the line with the correct arguments to create a dual blinking LED pattern using these parameters:*

```
pattern = BTN_LED_PATTERN_HALF
arg = 0
speed = 10
nr = BTN_LED_INFINITE
```

Exercise 6 *Check the documentation available in the datasheets, application notes, mailing list archives, Nut/OS webpage, Avrfreaks forum, tool resources, etc... to get an overview on the different compilers, libraries, programming variants and hardware programmers available for the Atmel AVR family.*

Exercise 7 *Open the avr-libc Manual (online version available on the avr-libc webpage). Find the mathematics functions in the avr-libc and check what functions are supported. Compare this selection to the CPU description found in the ATmega128l Manual and the instruction set of the ATmega128l found in the AVR Instruction Set Manual. Don't forget to read the available footnotes to learn about device specific options.*

Think about what functions you would like to use to implement certain algorithms. Why are function such as `tan()` *present, but simple multiply and divide operations are missing? How would you implement a fixed point division or even floating point operations for the AVR?*

In addition check the `FAQ` *found in the avr-libc Manual* `Related Pages` *documentation (especially entry 2) and the* `General Utilities Module` *of the avr-libc Manual for information on further functions like* `div()`, `qsort` *and* `rand()`.

Are there other libraries and languages available for the AVR family? Search for possible solutions on the web.

Optional Exercise 8 *When linking an application for a microcontroller a startup or initialization code needs to be integrated that controls the bootup and initialization procedure and sets the system into a default state after power-on. This behavior can be specifically controlled by a memory map and init sections. For an introductory documentation of the most common compiler flags and build steps, read through the* `Demo Projects Module` *in the avr-libc Manual.*

This topic is very complex. So we will generally use a pre-configured set up from the BTnut build system to integrate the (hardware dependant) correct startup code and memory map.

Optional Exercise 9 *In addition to the* `ChangeLog` *and* `README` *files provided with the BTnut System Software, the project management environment on http://sourceforge.net/projects/btnode has a* `Tracker` *and* `Tasks` *section to track bugs, requests for enhancements (RFEs), support requests etc. Check these locations to learn more about development issues and possible caveats. If you discover a bug either enter it into sourceforge.net or post them on the BTnode mailing list.*

Now you have gained an overview of the BTnut System Software, developing in Eclipse and know how to navigate code and search for documentation.

Explanation *BTnut Configuration Options*:
The BTnut System Software uses a GNU make based build system. The basic configuration is done in a file `Makerules` and parameters are defined in `Makedefs` and can be overridden by setting them as environment variables:

```
BURN = avrdude
BURNPORT = /dev/ttyS0
BURNFLAGS = -pm128 -cavrispv2 -P$(BURNPORT) -s
```

Alternatively you can use uisp with the settings:

```
BURN = uisp
BURNPORT = /dev/ttyS0
BURNFLAGS = -dprog=stk500 -dpart=atmega128 -dserial=$(BURNPORT) \\
--wr_fuse_e=0xFF --wr_fuse_h=0x00 --wr_fuse_l=0xBF

# Defines for btnode3 platform
MCU.BTNODE3 = atmega128
ARCH.BTNODE3 = avr
HWDEF.BTNODE3 = -D__HARVARD_ARCH__ -D__BTNODE3__
#DEFS.BTNODE3 = $(HWDEF.BTNODE3)
DEFS.BTNODE3 = -DNUTTRACER $(HWDEF.BTNODE3)
#DEFS.BTNODE3 = -DNUTTRACER -DNUTTRACER_CRITICAL $(HWDEF.BTNODE3)
#DEFS.BTNODE3 = -DNUTDEBUG $(HWDEF.BTNODE3)
```

Here, you can select parameters for the default programming interface and define debugging verbosity. We will make use of these features in later chapters of this tutorial.

2.5 First steps in BTnode programming – Using the avr-gcc toolchain

We will now use the tools to compile and upload a first program to the BTnode.

Explanation *ISP Programming Variants*: There are numerous software and hardware components that allow ISP programming of an Atmel AVR microcontroller.

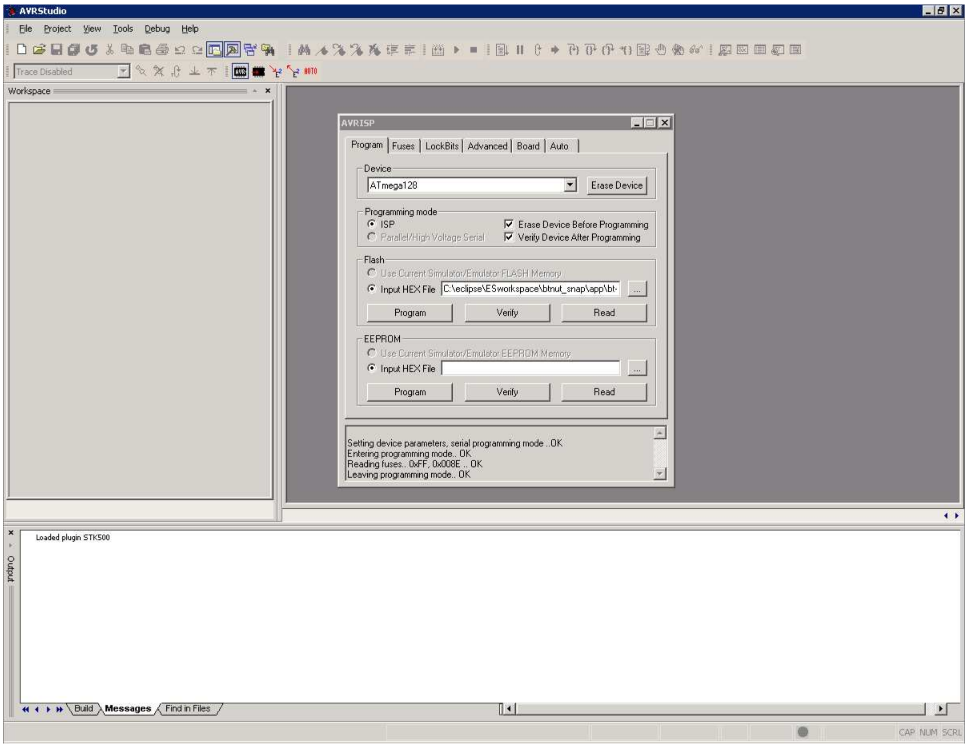
The default tool supported by Atmel is AVR Studio which offers a graphical user interface, simulation and project management capabilities. To use it for programming of an AVR only, open the tool and select the `Program AVR` entry from the `Tools Menu`. Be sure to select the correct device (ATmega128) in the `Program Tab`, do not change the fuse bit settings and select the right `Communications Settings` (Auto) in the Advanced Tab (see figure 2.5). When continuing from the command line be sure to close AVR Studio.

There are numerous command line tools for ISP programming as well. These are often more convenient than the GUI based tools. You have already used `avrdude` which also supports a GUI on windows.

For further informations such as using the bootloader function read the Atmel Applications Notes *AVR109: Self Programming*, *AVR910: In-System Programming* and *AVR911: AVR Open-source Programmer.*

Exercise 10 *Open a command line shell and check if your avr-gcc toolchain is installed and working correctly. First check the versions of the avr-gcc toolchain by entering* `avr-as --version`, `avr-gcc -v` *and* `avr-ld -v`. *Furthermore we will test* `avrdude -v` *(optional also* `uisp --version`*) that we will later use to upload code to the ATmega128l.*

Optional Exercise 11 *To see specific hints and help on the toolchain, execute the tools with the* `--help` *parameter from the command line or the man pages (unix) to get detailed online help.*



Device: ATmega128

Programming mode
● ISP
○ Parallel/High Voltage Serial
☑ Erase Device Before Programming
☑ Verify Device After Programming

Flash
○ Use Current Simulator/Emulator FLASH Memory
● Input HEX File: C:\eclipse\ESworkspace\btnut_snap\app\bt-

EEPROM
○ Use Current Simulator/Emulator EEPROM Memory
● Input HEX File:

```
Setting device parameters, serial programming mode ..OK
Entering programming mode.. OK
Reading fuses.. 0xFF, 0x008E .. OK
Leaving programming mode.. OK
```

```
Loaded plugin STK500
```

Figure 2.5: AVR Studio offers a graphical frontend to programming, simulation and project management functions.

Exercise 12 *Now connect a BTnode to your PC using a usbprog board and a USB cable (see figure 2.6). Further connect an Atmel ATAVRISP programmer to the usbprog board and to a serial port on your PC. The default settings are* `/dev/ttyS0` *for programming through an ATAVRISP and* `/dev/ttyUSB0` *for debugging through the serial port of the BTnode. (If in doubt about the right serial port for debugging use the* `List_USB2UART` *script on windows or check* `/var/log/messages` *on linux.).*

Try to communicate with the ATAVRISP and the ATmega128l on the BTnode:

```
avrdude -pm128 -cavrispv2 -P/dev/ttyS0
```

Explanation *Using the USB-UART adapter board*: The usbprog rev2 board is used for a breakout of all pins available on connector J1. Furthermore it contains a USB to UART converter (Silabs CP2101) that is used to connect the debug UART of the ATmega128l to a PC (default usage). A dedicated connector for ISP programming is also available on the usbprog board. Also when using the USB connection, the BTnode is remotely powered from the PC to save battery power.

Be sure to orient the usbprog board correctly as shown in figure 2.6. The board goes above the power switch of the BTnode with the two mounting holes matching those on the BTnode. If in doubt about the right serial port for debugging use the `List_USB2UART` script on windows or check `/var/log/messages` on linux.

Exercise 13 *Now upload a first pre-compiled application to your BTnode. Download the newest example application file* `bt-cmd.btnode3.hex` *from the BTnode project sourceforge.net file release page. Open a command line shell. In this step you will two use* `avrdude` *commands that are executed by the ISP programmer:* `erase` *and* `upload`*. First erase any programs present in the flash memory of the ATmega128l using* `erase`*:*

```
  avrdude -pm128 -cavrispv2 -P/dev/ttyS0 -e
```

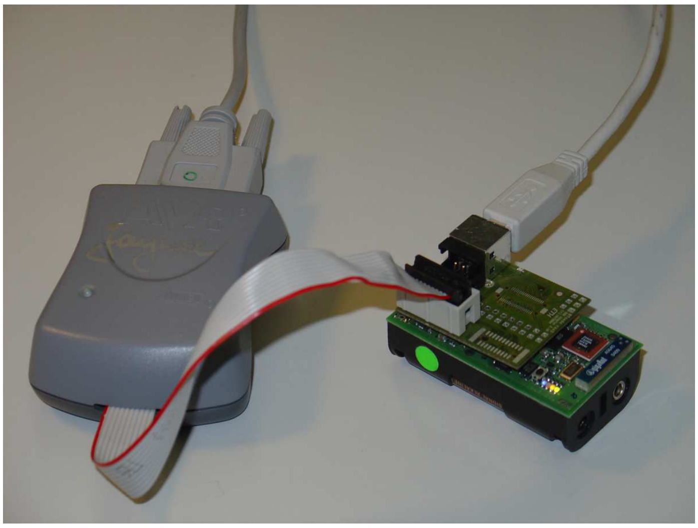


Figure 2.6: Debugging a BTnode using a USB connection to a serial port and ISP programming with the Atmel ATAVRISP.

Then program the new application code from an Intel Hex file format to the BTnode using `upload`*:*

```
avrdude -pm128 -cavrispv2 -P/dev/ttyS0 -D -V -s -U flash:w:bt-cmd.btnode3.hex:i
```

The `-D` *flag disables the auto-erase function, the* `-V` *flag disables auto-verify and the* `-s` *flag requires safemode. You can add the* `-v` *flag to receive more verbose output. Observe the LEDs on the BTnode for output from your first uploaded program.*

Explanation *Installing the bootloader*: Download the newest bootloader file `bootloader.btnode3.hex` from the BTnode project sourceforge.net file release page. To install the bootloader, proceed to upload this program code to the BTnode using the ISP programmer as described analogously for `bt-cmd.btnode3.hex` in exercise 13. Now your BTnode is ready to receive software flash reprogramming instructions.
To compile your own bootloader, navigate to the `btnut_system/btnut/app/bootloader` folder. Compile the bootloader by executing `make btnode3`. You should now have a file called `bootloader.btnode3.hex`.

Optional Exercise 14 *An alternative to using the ISP programmer is using a bootloader on the BTnode that can emulate ISP behaviour. The bootloader may or may not be installed on your BTnode but can be built from source code and installed using the method introduced in the previous exercise. See the explanation box below for more information.*

As of now [April 2006], the bootloader is not fully compatible with `avrdude`*, so we will need to use the* `uisp` *tool. Again, open your shell to the location where you have placed* `bt-cmd.btnode3.hex`*.*

Now, to upload the program code:

1. *press* ***and hold*** *the reset button on the BTnode*
2. *execute the upload command below*
3. *release the reset button on the BTnode*

```
uisp -dprog=stk500 -dpart=atmega128 -dserial=/dev/ttyS0 --upload if=bt-cmd.btnode3.hex
```

If you can't find the reset button, see figure 2.7. If you get strange error messages while programming, try to disconnect and reconnect the USB cable.

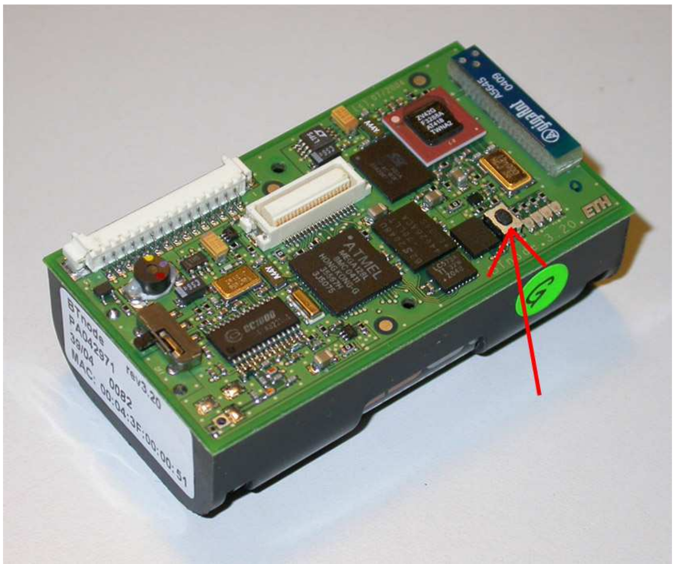


Figure 2.7: BTnode reset button

Exercise 15 *Erase the* `bt-cmd` *application on the BTnode. Open a terminal programm to the serial port you have connected your usbprog board with* `57.6k`, `8N1`, `no handshake` *to observe the terminal output from the BTnode.*

Upload the simple application `uart-echo.btnode3.hex` *with uart output to the BTnode. As soon as the uart-echo application responds, you can type and see the response on the LEDs. This time use the auto-erase function and auto-verify on avrdude:*

```
avrdude -pm128 -cavrispv2 -P/dev/ttyS0 -s -U flash:w:uart-echo.btnode3.hex:i
```

WARNING: DO NOT USE OTHER LOW-LEVEL COMMANDS WHEN IN-SYSTEM PROGRAMMING UNLESS YOU KNOW WHAT YOU ARE DOING AS IT COULD DAMAGE THE MICROCONTROLLER!.

Exercise 16 *Now go back to the* `bt-cmd` *application in Eclipse that we modified earlier and save the changes we have made. Open a command line shell on this directory. Compile the* `bt-cmd` *application by entering:*

```
make btnode3
```

Then upload the newly compiled application to the BTnode with:

```
make btnode3 upload
```

Observe the different LED heartbeat compared to the pre-compiled `bt-cmd.btnode3.hex` *we uploaded earlier. Check the terminal program for output. Hit* `Tab` *twice to get a selection of commands possible in the* `bt-cmd` *application. Explore the different functions available in this demo application. Try to locate different BTnodes by issuing* `bt inquiry sync`.

Explanation ***The bt-cmd demo application*****:** The `bt-cmd` demo application is a brief example of how to use the Bluetooth radio and protocol stack. Once the application has booted and is ready on a serial terminal with `57.6k`, `8N1`, `no handshake` you can check the list of available commands by hitting `Tab` twice.

```
# ----------------------------------------------------
# Welcome to BTnut (c) 2006 ETH Zurich
# bt-cmd program version: 20060405-1206
# $Id: firststeps.tex,v 1.10 2006/05/12 20:45:19 beutel Exp $
# running @ 7.3628 MHz, NutFreq=10241 Hz
# ----------------------------------------------------
booting Bluetooth module...
Bluetooth MAC address: 00:04:3f:00:00:d2
HCI version: 2 00C9 2 0012 003D
LMP features: 03 10 00 FF FF 05 F8 1B
Local name: 'ZeevoEmbeddedDevice'
hit tab twice for a list of commands
[bt-cmd@00:d2]$
bt      led     bat     nut     log
[bt-cmd@00:d2]$
```

There are NutOS/BTnode and Bluetooth specific commands (if called without arguments they will show hints on the correct syntax, where applicable).
`bt` – bluetooth radio commands
`led` – toggle LED patterns
`bat` – get the battery status
`nut` – show OS system information
`log` – BTnut logging features

For reference on Bluetooth [2] see the support documents and links provided on the BTnode web-page (see section 1.4).

Exercise 17 *To simplify the building and uploading we will now create* `Make Targets` *in Eclipse that you can execute with a single click.*

Open the `Make Targets View` *(Window → Show View → Other → Make) and navigate to the* `app/bt-cmd` *folder. Right click onto this folder and select* `Add Make Target`*. Alternatively you can create different targets by entering make arguments such as* `all`*,* `btnode3`*,* `version` *or* `clean`*.*

Then use these `Make Targets` *to automatically build and upload selected applications from within Eclipse. You can observe the progress and console output from the respective views.*

Exercise 18 *Right click onto the* `bt-cmd.c` *file in the* `C/C++ Projects View` *and select* `Compare With Local History` *to see the changes you have made earlier.*

Exercise 19 *Create a new folder in the* `app` *directory and copy the* `bt-cmd/Makefile` *to this folder. Create (or alternatively copy and rename) a new* `application.c` *file in this folder. Be sure to edit the project name in the* `Makefile`*.*

Now you are ready to program your first own project using BTnut.

Explanation ***Resetting the work environment to initial conditions*****:** The pre-compiled BTnut snapshot used in this tutorial can be obtained from http://sourceforge.net/projects/btnode, section Files. Download the `btnut_snap_btnode3_binary_x.x.tar.gz` file and unpack it to a location of your choice. Now create a new `Standard C/C++ Project` in Eclipse and import the files from the `btnut_snap_avrbinary` archive.

Optional Exercise 20 *In order to stay up to date on the bleeding edge development codebase of BTnut you will need to check out the most current version from the CVS repository on sourceforge.net. Open the* `CVS Repository Exploring` *perspective in Eclipse and create a new CVS repository:*

```
Host: btnode.cvs.sourceforge.net
Repository path: /cvsroot/btnode
User: anonymous
Connection type: pserver
```

The check out the CVS `HEAD` *of the module* `btnut` *as a* `Standard Make C Project`*. You can check for changes to the most current CVS tag* `HEAD` *or to other dates and tags by seleting* `Compare With...` *or* `Replace With...`*.*

Before building the demo applications in the `app` *directory you will need to check out a release of Nut/OS either by executing* `make nut_cvs_sources` *in the btnut directory or by checking it out from CVS into a parallel project as described above (host* `btnode.cvs.sourceforge.net`*, repository* `/cvsroot/ethernut`*, module* `nut`*. The build the BTnut libraries first by executing* `make clean` *and* `make all` *in the* `btnut` *directory.*

Chapter 3

Device-Level Programming

3.1 Introduction

The goal of this session is to familiarize the reader with some peculiarities of programming microcontrollers that have a rich set of peripherals. After going through the tutorials and exercises, you should be able to understand and write simple drivers which allow you to use these peripherals efficiently. To work through the whole chapter takes you approximately four hours, without the optional exercises about two hours.

In this session, we will avoid using library functions and operating system support as far as possible. The reason is that you should be able to really understand what is going on instead of using some black-box functionality. Clearly, this type of programming is often a bit cumbersome. But you will enjoy the comfort and convenience of an operating system that you will learn to use in the next session all the more.

In Section 3.2, the use of off-chip resources is explained using the example of the LEDs on the BTnodes. In Section 3.3, the reader learns how to use the analog-digital converter of the ATmega128 as an example for an on-chip resource. In Section 3.4, we introduce interrupts. The final Section 3.5 deals with critical sections that are required to protect shared data.

3.2 Off-chip resource: Setting and Clearing LEDs

As a first example, we now use 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. The LEDs are an *off-chip* resource. Unfortunately, accessing the LEDs is a bit tricky and requires some “hacks”, which are explained in the following.

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). The address bus is mainly needed to access the external SRAM (AMIC_LP62S2048), but at the same time it is also connected to the LEDs via a latch. 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 samples value. The following function does exactly this:

```
void write_led(u_char value) {
    volatile u_char * pointer;
    u_char dummy;

    // compute the pseudo-address that contains the values for the LEDs
    pointer  = (u_char *) ( ((u_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
    asm volatile  ("nop" ::);
    // disable the latch, i.e. hold the value
    PORTB &= ~(1<<PB5);
}
```

Explanation *volatile*:
Note the keyword `volatile` before the declaration of `pointer`. It tells the compiler that code lines containing `pointer` should not be optimized at all. This is necessary because the compiler does not know anything about external off-chip resources like the LEDs. Thus it cannot understand why we compute the variable `dummy`, which is never used afterwards. If `volatile` were omitted, the compiler would simply ignore such “nonsense” statements.

Explanation *Accessing special purpose registers*:
The names of special purpose registers are defined in the `hardware/btn-hardware.h` header file and in header files included therein. These names can be used like variables. For example you may read the content of the `PORTB` register using

```
u_char current_portb = PORTB;
```

Similarly, you can write to such a register in the same way as you write to a variable, e.g.

```
PORTB = 0xff;
```

sets all bits of the `PORTB` register to one.
Most often however, you only want to read or write a single bit of a special purpose register. This can be done by using the bitwise *and* / *or* operators. The names of individual bits are also defined in the header files. But these names cannot be used like variables, they are simple aliases for the position of the corresponding bit within a register. For example `PB5` is an alias for 5 since the `PB5` bit is the fifth bit within the `PORTB` register (counted from the left starting with 0). Examples:

```
if (PORTB & (1<<PB5)) // checks whether the PB5 bit is set
PORTB |= 1<<PB5;      // sets the PB5 bit to one
PORTB &= ~(1<<PB5);   // clears the PB5 bit
```

Exercise 21 *To check whether you have understood how LEDs are controlled, use the BTnode schematics to figure out the* `value` *needed to switch on the blue LED. Explain the computation of* `pointer`*.*

As a start, we write a program that blinks with the blue LED. The main routine thus looks as follows:

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

int main(void) {
    DDRB |= 1<<DDB5;
    while (1) {
        // toggle the blue LED
        // wait a second
    }
    return 0;
}
```

Explanation *Configuring the direction of IO ports*:
The line before the infinite loop configures the fifth bit of the `DDRB` register. DDRB stands for Data Direction Register of Port B and this operation declares the fifth pin of port B to operate as an output pin. After this line, you are free to use the `write_led` function shown above. See pages 63ff in the ATmega128 manual for a detailed explanation.

Exercise 22 *Complete now the program sketched above. In order to see what your program does, you will have to implement a* `pause` *function. Do this using a loop that increments a counter variable.*

Optional Exercise 23 *Once your program is running, try to estimate the clock frequency of the ATmega128. Do this by counting the operations in the loop of your pause function.* **HINT:** *Look at the list file (*`<program name>.lst`*) which has been created by the compiler. Even without understanding any assembler at all you can find your function by searching for its name. You can identify the loop by looking at the labels ("*`.L6:`*", for example) and the branch instructions ("*`brlo .L6`*", for example). Assume that all assembler instructions take one cycle to execute.*

3.3 On-chip resource: The Analog to Digital Converter

The ATmega128 microcontroller contains an on chip analog to digital converter (ADC), whose detailed description can be found on pages 231 to 247 of the ATmega128 manual. As for all on-chip resources, the ADC can be configured by writing to special purpose registers, its status and the conversion result can be accessed by reading from special purpose registers. In the case of the ADC, the two 8 bit registers called ADMUX and ADCSRA are used for configuration and status. The two 8 bit registers called ADCH and ADCL are used to deliver the conversion result.

As we now know how to use the LEDs, we can start writing more complex programs. We now want to sample the battery power and show the result using the LEDs. The solution should look as follows:

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

int main(void) {
    int battery_power;
    DDRB |= 1<<DDB5;
    while (1) {
        battery_power = get_battery_voltage();
        // if battery_power below 1000mV, switch on red LED
        // if battery_power between 1000mV and 2000mV, switch on yellow LED
        // if battery_power above 2000mV, switch on green LED
        // wait a second
        // switch on blue LED
        // wait a second
    }
    return 0;
}
```

We now have a more detailed look at the function `get_battery_voltage`. Its skeleton looks as follows:

```
int get_battery_voltage(void) {

    // configure ADMUX
    ADMUX |= 1<<MUX0;
    ADMUX |= 1<<MUX1;

    // configure ADCSRA register such that the conversion
    // is as slow as possible and the ADC is enabled

    // start conversion and wait for result

    // read (and convert ?) result
}
```

In a first step, the `ADMUX` register is configured. As you can see in the manual, page 244, all bits are cleared at startup and we only have to write the bits which we want to be one. Looking at BTnode schematics, we see that the `BAT_SENSE` signal is connected to pin 3 of port F. From the manual, page 239 we know that this pin is the third channel of the ADC and table 98 on page 244 tells us that we have to set the bits `MUX1` and `MUX0` from the `ADMUX` register to sample the voltage from channel three. We leave the `ADLAR` bit cleared. The `REFS1` and `REFS0` bits are left cleared because we use the external voltage reference connected to the `AREF` pin of the ATmega128.

WARNING: DO NOT USE OTHER SETTINGS FOR THE REFSx BITS, IT COULD DESTROY THE MICROCONTROLLER!.

Exercise 24 *Now its your turn to configure the* `ADCSRA` *register. For maximal precision, we want the slowest conversion speed. We do not use interrupts and we want to do a single conversion.*

After having configured the ADC, the conversion can be started. This is done setting the `ADSC` *bit of the* `ADCSRA` *register. This bit is automatically cleared when the conversion is completed. Wait for this condition and then read the result from the* `ADCL` *and the* `ADCH` *register.*

Determine the values you expect from the ADC for a battery voltage of 1 volt and 2 volts, knowing that the reference voltage is 3300 millivolts, the ADC delivers 10 bit values and the `BAT_SENSE` *signal is half the battery voltage (see schematics).*

HINT: *If your conversion result is always zero, make sure that (i) you either have batteries in your BTnode or you have connected the battery contacts to an external power supply and that (ii) the power switch is on (if connected to the USB cable, the BTnode is also powered if this switch is off, but then the* `BAT_SENSE` *signal is 0).*

3.4 Writing interrupt routines: Hardware Timers

In this section, the program from the previous section is modified such that it periodically samples the battery voltage in a timer interrupt routine. The advantage is that now the microcontroller can do other work in parallel. The processor load created by the timer interrupt is measured using an IO pin and the oscilloscope.

Explanation *Hardware Timers*:
Another type of on-chip resources are timers. In principle, timers are counters that are incremented automatically. By the use of configuration registers, the speed of incrementing the timers can be adjusted and whenever the timers overflow or reach a specified value, they trigger an interrupt.

Explanation *Interrupt Service Routines (ISR)*:
Interrupts are used to execute a function, the so-called interrupt service routine. The normal program flow (the main function, in our case) is interrupted and the interrupt service routine is executed. As soon as it terminates, the normal program flow is resumed exactly at the position where it was interrupted.

Timer interrupts can thus be used to execute some periodic functionality without having to spend the whole processing time on waiting. An example is shown here:

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

static void timer3IRQ(void *arg) {
    // switch on green led
}

int main(void) {

    // register interrupt service routine
    NutRegisterIrqHandler(&sig_OVERFLOW3, timer3IRQ, 0);
```

```
    // configure the speed of the timer
    TCCR3B |= 1<< CS30;
    TCCR3B |= 1<< CS32;
    // enable the interrupt at overflows of the timer
    ETIMSK |= 1<< TOIE3;

    while (1) {
        // toggle the blue led
        // wait a second
    }
    return 0;
}
```

In addition to the main routine, the interrupt service routine (ISR) `timer3IRQ` is defined. At the very begin of main, `timer3IRQ` is registered as the service routine for the `sig_OVERFLOW3` interrupt, that is for the event that timer 3 overflows.

After registering the ISR, the timer is configured. The `CS30` and `CS32` bits of the `TCCR3B` register are set to configure the speed of the timer. In this case, the timer is incremented every 1024 clock cycles (see page 135 of the manual). The timer does not have to be started, it is always active. However, the generation of interrupts when the timer overflows has to be enabled. This is done by setting the `TOIE3` bit of the `ETIMSK` register.

Exercise 25 *We now will modify the previous program, such that the battery power is sampled in a timer ISR. Use timer 3 in such a way that the battery power is sampled approximately once every two seconds. The ISR displays the sampled result on the LEDs, but in contrast to the previous program, it does not wait and switch on the blue LED.* **HINT:** *To adjust the interval of the ISR, you can change the prescaler (*`CS3x` *bits) and/or set the timer manually to a non-zero value after every overflow.*

Optional Exercise 26 *Modify the program from the previous exercise using the* clear timer on compare match (CTC) *mode of the hardware timer, which is described on page 121 and 131ff. Also use the ISR to display the result of the battery power sampling using the LEDs as in the previous example.*

In a real-world program, often a large number of different interrupts are used to service multiple peripherals at the same time. By default, interrupts are blocked while an ISR is executing, thus different interrupts can block each other. Therefore the careful programmer aims at keeping ISRs as short as possible.

Optional Exercise 27 *Measuring the execution time of an ISR can be done as follows: On a free IO pin of the ATmega128, we generate a rising edge at the begin of the ISR and a falling edge at the end. The time that the IO pin is high can then be measured on an oscilloscope. For example we may use pin 0 of port F, which is a good choice since it is accessible as pin 6 on the 15-pin-connector of the BTnode, as you can verify on the BTnode schematics. Connect this pin and ground (e.g. from pin 1 of the 15-pin-connector) to the oscilloscope. Set up pin 0 of port F as an output pin using the* `DDRF` *register. How long takes your ISR to execute? How much of the processing power is thus used for sampling the battery power every two seconds?* **HINT:** *If you only have an analogue oscilloscope, you may have to decrease the interval of the ISR drastically (e.g. 10ms is a good value) in order to display the generated waveform properly.*

3.5 Protecting shared data and resources

In this section, the program from the previous section is extended to write measured data to the terminal. It is explained why this should not be done from interrupt context. Thus the sampled data has to be shared by the ISR, which determines the battery voltage and the main routine, which prints it to the terminal. It is explained why this shared data has to be protected from uncoordinated concurrent access by multiple flows of control and how this can be done.

Explanation *Using the terminal*:
The ATmega128 has also two serial interfaces, so called Universal Asynchronous Receiver Transmitter (UART) units. The UART0 is used to connect the ATmega128 to the Bluetooth module. The UART1 can be used to write ASCII text to the terminal, which is a program running on the host computer. Writing text to the terminal can be done using the well-known `printf` function from the 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.

```
int variable = 13;

printf("Hello world, ");
printf("my lucky number is %d\n",variable);
```

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, that is 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>
#include <stdio.h>                  // freopen
#include <io.h>                     // _ioctl
#include <dev/usartavr.h>           // NutRegisterDevice, APP_UART, UART_SETSPEED

void init_stdout(void) {
    u_long baud = 57600;

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

To read data from the terminal, you can use the function `fscanf`.

Optional Exercise 28 *Write a program, that samples the battery voltage once every two seconds using a timer ISR. Instead of displaying the result on the LEDs, print it to the terminal from within the ISR. Measure the execution time of the ISR using the oscilloscope.*

The measurement of the execution time of the ISR shows that `printf` takes a lot of time. We have discussed before that ISRs should be as short as possible. Therefore we want to do the printing of the sampled battery voltage from the main routine. Of course we want to print every measurement result exactly once.

Exercise 29 *Rewrite the program from Ex. 28 such that the battery voltage is sampled in the ISR but that the printing of the result is done in the main routine. To do this, you have to think about some communication mechanism between the two flows of control.*

Optional Exercise 30 *Instead of printing the result from reading the* `ADCL` *and* `ADCH` *registers directly, print it in millivolts.* **HINT:** *Remember that an unsigned short variable overflows at 65536, thus be careful about the data types you use.*

The program you have written probably works just fine. But if you would have a lot of time to observe its behavior (or if you are “lucky”), you would notice that sometimes strange values are printed on the terminal.

Explanation *Corruption of Unprotected Data*:
If two flows of control, e.g. the main routine and an ISR access a piece of data, its value can become corrupted. Assume that the ISR writes to a 16 bit variable which is read by the main routine. Assume that its value at some point of time is `0x00ff`. Now the main routine first reads the upper byte, that is `0x00`, and then the ISR is executed. The ISR may increment the variable to `0x0100`. After the ISR has terminated, the main routine continuous reading the variable and reads the lower byte as `0x00`. Now the main routine has read the variable as `0x0000`, which is far off the real value of either `0x00ff` or `0x0100`.

This problem can be solved by using *critical sections*, that is by protecting the access of a shared variable in the main routine from being interrupted by an ISR. The other way round is no problem, since an ISR cannot be interrupted by the main routine.

Explanation *Enabling and Disabling Interrupts*:
To protect a piece of code from being interrupted, you can disable interrupts globally using the function `cli()`. To reenable interrupts, you can use the function `sei()`. These instructions clear and set the I-bit of the `SREG` register, which is the main status register of the ATmega128 microcontroller.

Exercise 31 *Protect the shared data that is used in your program from Ex. 28. Do this by implementing the functions `EnterCritical` and `ExitCritical`. Make sure that `ExitCritical` does not enable interrupts if they were disabled before `EnterCritical`.*

Optional Exercise 32 *Not all data access conflicts are so easily visible as the shared variable from Ex. 28. For example our implementation of the `write_led` function has a problem of this kind too. Explain why and fix it.*

Chapter 4

Programming with Threads

4.1 Introduction

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

- Complicated programs can be divided into a set of threads. Programming a single thread is much easier than programming the whole functionality in a single program. The coordination of the execution of these threads is done by the operating system. It is the main focus of this chapter to introduce the API of the BTnut OS for creating, executing and terminating threads, as well as for the communication and coordination of such threads.
- You do not have to read hardware schematics and manuals when you want to use resources since we now can use library functions. In this chapter you will use such functions for accessing the LEDs and the terminal. Also for the analog to digital converter we have used in the last chapter such library functions would be available, see the `dev/adc.h` header for a description. There is even a function `btn_bat_measure`, doing exactly what we have done manually (see `hardware/btn-bat.h`).

Section 4.2 deals with the creation of threads. Section 4.3 introduces a special thread provided by the BTnut OS, called “terminal”. This thread is used to allow interactive control of a BTnut application. In Sect. 4.4, *events* are introduced as a means of coordination and communication between threads.

4.2 Creating Threads

First we look at how threads are defined.

Explanation *Creating Threads*:
Threads are functions. For example, the main routine is a thread, which is started automatically after startup. Additional threads have to be declared using the `THREAD` macro. An example defining the thread `my_thread` is shown below.

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

Functions that are used as threads are supposed to never return, thus to loop endlessly. 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.

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>
#include <sys/timer.h>

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

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, which is passed to the thread function (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 31 for a method to check whether your stack size is correctly chosen. For now, just use 192 and you will be fine.

Some threads are already defined by the operating system. For example there is a thread that controls the LEDs.

Explanation *LED Thread*:
Instead of controlling the LEDs directly as we have done in the Ch. 3, we can use the LED API of the BTnut OS. To do this, we have to include the `led/btn-led.h` header file and then we can initialize the LEDs using `btn_led_init`. This function has a single argument and if this is not 0, then it starts the LED thread. The LED thread allows you to display dynamic patterns on the LEDs 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.
We still can switch on and off LEDs individually using the commands `btn_led_set` and `btn_led_clear`. Both functions have the number of the LED as their single argument. 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.

Explanation *NutThreadYield*:
The BTnut OS is a cooperative multi-threading OS. In principle (we will see an exception later on), threads that run only yield the CPU to other threads when 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 priority are ready to run. 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. If no thread with a higher priority than the current thread is ready to run, `NutThreadYield` returns immediately.

Exercise 33 *Write a program, that creates a thread as explained above. This thread shall repeatedly turn on the blue LED (using `btn_led_set(0)`) and switch off the red LED (using `btn_led_clear(1)`). 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).*

Explanation *NutSleep*:
There are other ways to yield the CPU to other threads than `NutThreadYield`. It is quite a common situation, that a thread has finished some work and now wants to pause for a while. Remember that in the last chapter, we have implemented the `pause` function for this purpose. But this solution had the disadvantage of blocking the CPU during the whole pause. Thus you preferably use the `NutSleep` function as shown below

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

THREAD(my_thread, arg) {
    for (;;) {
        // do something
        NutSleep(1000);
    }
}
```

The `NutSleep` function has a single parameter, which determines the number of milliseconds after which the execution of the thread shall be resumed. `NutSleep` yields the CPU to other threads, which can do useful work during the sleep period.

Exercise 34 *Write a program with a main routine and an additional thread. Both threads repeatedly write a message to the terminal and sleep for one second. What do you observe? What did you expect? Do not worry if the two answers do not match, you have just discovered a bug of the BTnut OS (which will be fixed soon, hopefully).*

Explanation *Thread Priorities*:
In the BTnut OS, threads have a priority in the range of $[0, 254]$, 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
    }
}
```

The thread priorities are used to decide which of several ready to run threads shall be executed. When all ready to run threads have the same priority, the threads are processed in FIFO order.
Note that changing the priority of a thread may implicitly yield the CPU to another thread. This is the case if the running thread reduces its priority and then is no longer the thread with the highest priority that is ready to run.

Optional Exercise 35 *Repeat Ex. 34 giving the additional thread a higher priority. Compare the output with what you received in Ex. 34. Repeat the experiment giving the additional thread a lower priority. What do you observe?*

Optional Exercise 36 *Write a program with two threads that permanently write to the terminal using* `printf` *without sleeping. Describe the observed behavior and explain, why it is different from what you would expect from the theory of cooperative multi-tasking.* **HINT:** *Writing to the terminal is done with the speed of the UART, i.e. 115kBits 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 *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. 41.

4.3 The Terminal

We have introduced the `printf` and `scanf` functions already in the last chapter. Here we present a more convenient way to use the terminal.

Explanation *The Terminal Thread*:
The BTnut OS helps you to interact with a user via the terminal. To this purpose, a thread is created that receives input from the UART that is linked with the standard output stream (see 24) and echoes received characters. This allows you to see what you type in the terminal application running on the host computer. In summary, this thread implements a simple command line interface to the BTnode. The following program is an example for using this facility:

```
#include <stdio.h>
#include <dev/usartavr.h>
#include <sys/thread.h>
#include <sys/timer.h>
#include <hardware/btn-hardware.h>
#include <terminal/btn-terminal.h>


int main(void) {
    btn_hardware_init();
    btn_led_init(1);
    init_stdout();
    btn_terminal_init(stdout, "[es-ex3]$");
    btn_terminal_run(BTN_TERMINAL_NOFORK, 0);
    return 0;
}
```

After the usual initializations (for an explanation of `init_stdout`, see page 24), 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, since it reuses the main routine (which is also the main thread) as the terminal thread.

Explanation *Creating your own Terminal Commands*:
The terminal thread also parses the received string after you press enter and executes a function, if the string matches to a registered terminal command.

```
void square(u_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, "[es-ex3]$");
    btn_terminal_register_cmd("square",square);
    btn_terminal_run(BTN_TERMINAL_NOFORK, 0);
    return 0;
}
```

After (this is important) the initialization of the terminal thread, the command *square* is registered with `btn_terminal_register_cmd`. The first parameter is the string you will have to type to launch the function, which is given as the second argument. Note that functions which you want to register as a command must have the signature `void <functionname>(char* arg)`. The function receives the string `arg` as an argument. It contains the remainder of the string parsed by the terminal thread, e.g. if you type `"square 7"`, `arg` is a pointer to `"7"`.

Exercise 37 *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`*, which is a string, i.e. has type* `u_char*`*.*

Optional Exercise 38 *Rewrite the program from Ex. 37 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.*

Explanation *The Nut OS commands*:
The BTnut OS also 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 is given below:

```
#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`. `btn_cmds_register_cmds` provides the *led* command, `nut_cmds_register_cmds` provides the *nut* command, which has several sub-commands. For example with *nut threads*, you can print a list of all threads on your BTnode.

Optional Exercise 39 *Rewrite the program from Ex. 37 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.*

4.4 Events

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 40 *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 41 *Extend the program from Ex. 37 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 42 *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 43 *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.*

Chapter 5

Embedded Debugging

5.1 Introduction

The goal of this tutorial is to get to know the different tools and techniques for embedded debugging considering the BTnode platform as example.

One of the most compelling problems for anyone programming an embedded system, is to understand what your system is doing, what resources it's using and how it interacts with the external world. Bugs occur. Fixing them is usually easier than finding them! The problem is that embedded code cannot be easily executed under a debugger, nor can it be easily traced, because of the following circumstances:

- Embedded systems are **resource constrained**. Some debugging techniques might cause too much overhead (processing, communication and memory). Applying debugging may obscure the real problem (Heisenberg effect).
- The embedded processor is connected to **peripheral hardware components** such as A/D-converters, timers, communication interfaces, interrupt controllers and general purpose I/O pins. The embedded programm closely interacts with those components which makes it hard to trace.
- Embedded system often provide very **limited access** to the resources. If all you have is four LEDs, debugging will be very hard.

5.2 Tools

Good mechanics have many tools; you can't fix a car with just a hammer. Like good mechanics, good programmers need to be proficient with a variety of tools. Each has a place; each has a Heisenberg effect; each has power.[1]

Explanation *Simulator with source-level debugger*: A simulator allows for early debugging and execution of algorithmic code. It does not require any target hardware. A source-level debugger lets you step through your code, stop it, and then examine memory contents and program variables.

Explanation *In-circuit emulator (ICE) and JTAG debugger*: An Emulator *emulates* the behavior of the real chip. ICEs allow you to replace the real chip that interacts with I/O components for better insight. JTAG debuggers directly connect to the real chip instead of replacing it. ICEs and JTAG debuggers can be used for source-level debugging.

---

[1] The ten secrets of embedded debugging: http://www.embedded.com/showArticle.jhtml?articleID=47208538

Explanation *Simple printf statements*: This is perhaps the most flexible and primitive tool. Printing out variable values and function entry/exit points allows you to discover how your program is operating. Unfortunately `printf` is both clumsy to use (requiring code changes and recompiling) and quite intrusive because it greatly slows execution.

Explanation *Operating system monitors*: Operating system monitors display events, such as task switches, semaphore activity and interrupts.

Others: Profilers, memory testers, execution tracers, coverage testers.

5.2.1 Debugging techniques for the BTnode

Following tools can be used to debug an AVR microcontroller. Some techniques require additional special hard- and software:

	Technique	**Hardware**	**Software**
1	Simulator	–	AVRStudio / AVaRICE + GDB / SimulAVR + GDB
2	ICE	ICE40/ICE50	AVRStudio
3	JTAG debugger	JTAGICE (mkII)	AVRStudio / AVR insight
4	printf	UART	Terminal
5	OS monitor	UART	Nut OS Tracer, Terminal

Optional Exercise 44 *Open the AVRStudio and consult the AVR Studio Tools and User Guide.*

1. *Compare the features and limitations of an Emulator (ICE50) with the ones of a JTAG debugger (JTAGICE).*
2. *What is on-chip-debugging (OCD)? Which hardware is required for OCD on the BTnode?*
3. *What happens with the peripheral components of the μC (UART, Timers, A/D Converter) when you enter stop-mode for source-level debugging (e.g. when a breakpoint is hit)?*

Optional Exercise 45 *Consider following table. Which tool(s) is/are most appropriate, in your opinion, for the given problems? Sometimes all tools can be applied in order find and fix a bug. Some with more, others with less effort. Justify your answer.*

Problem	***Tool:*** *Simulator/ JTAGICE/printf*	***Reason***
An algorithm that operates from memory to memory does not behave as expected.		
The μC communicates over one of its hardware UART with the Bluetooth module. In general, the μC sends a command sequence and parses the reply from the module. The implementation of this protocol on the μC is erroneous and needs debugging...		
You are implementing a network stack. A series of function is called (for each network layer) to process an incoming packet. In your current implementation, when a packet is received, the μC freezes somewhere in the processing. You want to find out where.		

5.3 AVR Simulation

In this section you will learn how to use AVR simulation for prototyping and source-level debugging. We introduce two simulators: *simulavr* and the *AVR Studio Simulator*. Unfortunately, both of them are quite limited, i.e. they only simulate a subset of the AVR peripherals (timers, etc.). As a consequence, applications that use Nut/OS can not be simulated. The simulator usually breaks in one of the timer interrupt routines.

Exercise 46 ***Simulavr + AVR-Insight*** *In this exercise you will learn how to start* simulavr *and how to connect the AVR-Insight debugger to it.*

1. *Create a new C file* `simpleio.c` *with the following code:*

```
#include <io.h>

void delay(void){
        int i;
        for(i = 0; i < 1000; i++);
}

int main(void){
        DDRF |= _BV(0);
        for(;;){
                PORTF |= _BV(0);
                delay();
                PORTF &= _BV(0);
                delay();
        }
}
```

2. *Compile the file manually with the debug-symbols option (-g):*

```
avr-gcc -mmcu=atmega128 -g -I/usr/pack/btnode-1.0-mo/avr/include/avr simpleio.c -o simpleio.elf
```

3. *Start AVR-Insight:*

```
avr-insight&
```

4. *Start* simulavr *as a* gdb-server*:*

```
simulavr -g -d atmega128
```

The simulator should print out something like: `Waiting on port 1212 for gdb client to connect...`

5. *In order to connect Insight with the simulator, open the gdb-console (View→Console) and enter following commands:*

```
file simpleio.elf
target remote localhost:1212
load
break main
continue
```

6. *Congratulations: now you can step through your code, set breakpoints and watches.*

Optional Exercise 47 ***AVR Studio Simulator***

1. *AVR Studio needs a different debug format. Compile the code from the last exercise with* `-gdwarf-2`*.*

2. *Open AVRStudio and open your* `.elf` *file from the* `File->Open File` *menu. A project wizard appears. Select* AVR Simulator *as debug platform and* ATmega128 *as device.*

3. *The simulator initializes and stops at the first instruction. Go to the* `AVR Simulator Options` *from the* `Debug` *menu, and set the frequency to* 8.00 *MHz.*
4. *In the I/O workspace window on the left side you find all the simulated resources of the AVR. Take some time to browse through the individual items. Expand the* `PORTF` *item.*
5. *Congratulations: now you can step through your code (F10), set breakpoints and watch how the ports and registers change in the workspace.*

Optional Exercise 48 ***Profiling printf with AVR-Studio***
Printf statements are often used for debugging. Printing out variable values and function entry/exit points allows you to discover how your program is operating. In this exercise we measure the cycle count of an example printf statement in order to get the feeling of the overhead.

1. *Edit your "*`nutsim.c`*" file:*

```
#include <io.h>
#include <stdio.h>
#include <dev/usartavr.h>
#include <hardware/btn-hardware.h>

int main(void) {
    u_long baud = 57600;

    btn_hardware_init();
    NutRegisterDevice(&APP_UART, 0, 0);
    freopen(APP_UART.dev_name, "w", stdout);
    _ioctl(_fileno(stdout), UART_SETSPEED, &baud);

    printf("UART baudrate = 57600\n");

    for(;;);
}
```

2. *Recompile the* `.elf` *file. The simulator should automatically restart and load the new file.*
3. *Step through the code until the yellow arrow is on the printf statement. Expand the processor item in the I/O workspace. Remember the cycle count.*
4. *Proceed one step (step over). Compare the cycle counter with the previous values. How many cycles did it take?*
5. *Normally printf statements have formatted output. Replace the existing printf statement with:*

```
printf("UART baudrate = %u,%u kbaud\n",
       (int)(baud / 1000UL),
       (int)((baud - (baud / 1000UL)*1000UL)/100));
```

6. *Compare the cycle count of the formatted printf with the unformatted one.*

5.4 The OS-Tracer

Printf is often used for debugging. However, in the previous section, we have seen that this method has a relatively large overhead. Thus, it is not suitable for tracing frequent events such as interrupts or thread switches. For such events, the *tracer tool* is more appropriate.

Explanation *Tracer Tool, Interactive Mode*:
The tracer tool stores information about important OS events in memory and prints this information later on the terminal for analysis purposes. Important OS events include thread switches (due to sleeps, yields, priority changes, etc.) and interrupts. In addition to the type of event, the exact system time (microsecond resolution) and additional information (e.g. which thread did a sleep) is stored.
The tracer tool can be used in various different ways. The most simple is the interactive terminal mode. To activate the tool, use

```
#include <sys/tracer.h>
    btn_terminal_register_cmd("trace",NutTraceTerminal)
```

as it has been explained in the previous section.

Exercise 49 *This exercise is a step-by-step tutorial for using the trace tool. First write a program that starts the LED thread, then registers the* trace *terminal command and then starts the terminal thread. Run this program and continue as follows:*

1. *Type* `trace`*, you will get the output:*

```
[es-ex3]$trace
 TRACE STATUS
 Mode is OFF
 Size is 0
 contains 0 elements
SYNTAX: trace [print [<size>]|oneshot|circular|size
<size>|stop|mask [<tag>]]
```

2. *Type* `trace oneshot` *and then type* `trace` *again. If you have not waited too long between the two commands, you will get something like this:*

```
[es-ex3]$trace oneshot TRACE mode ONESHOT, restarted
[es-ex3]$trace TRACE STATUS
 Mode is ONESHOT
 Size is 500
 contains 77 elements
SYNTAX: trace [print [<size>]|oneshot|circular|size
<size>|stop|mask [<tag>]]
```

Typing `trace` *again will give you a similar status except that the* `contains XX elements` *shows an increasing number. When it has reached* 500*, the* `Mode` *changes to* `OFF` *again as it was before we typed* `trace oneshot`*, but now* `contains 0 elements` *is replaced by* `is full`*.*

```
[es-ex3]$trace TRACE STATUS
 Mode is OFF
 Size is 500
 is full
SYNTAX: trace [print [<size>]|oneshot|circular|size
<size>|stop|mask [<tag>]]
```

3. *In the previous step, we have filled the trace buffer with events. We now can have a look at them by typing* `trace print 10`*, which gives you an output like this:*

```
[es-ex3]$trace print 10

TRACE contains 500 items,  printing 10 items. TAG
PC/Info      Time [s:ms:us]
-----------------------------------------------
Thread Yield          idle         13:524:336 Thread Sleep
LED          13:524:604 Thread Yield          idle
13:581:857 Thread Sleep          LED          13:582:125 Thread
Yield          idle         13:639:392 Thread Sleep          LED
13:639:659 Thread Yield          idle         13:696:909 Thread
Sleep          LED          13:697:205 Thread Yield          idle
13:754:442 Thread Sleep          LED          13:754:710
```

In the `TAG` *column, you see the type of the recorded events. In the case shown above, all events are of type* `Thread Yield` *or* `Thread Sleep`*, the column* `Info` *shows you the name of the thread which has done a sleep or a yield and the* `Time` *column indicates at what time this was done. The time is* 0 *when the BTnode is booted.*

4. *The list of events does not allow you to quickly understand what is really going on. Therefore we now use the terminal program on the host computer to capture the terminal output in a file. Then we postprocess the trace file we have created in the previous step using Matlab. Thus start Matlab now. Type* `show_trace('<the filename of the captured terminal output>')`*, which opens a figure like the one shown in Fig. 5.1. Be sure to use a separate log file for each trace captured. In this figure, you can see time on x-axis and three threads on the y-axis.*

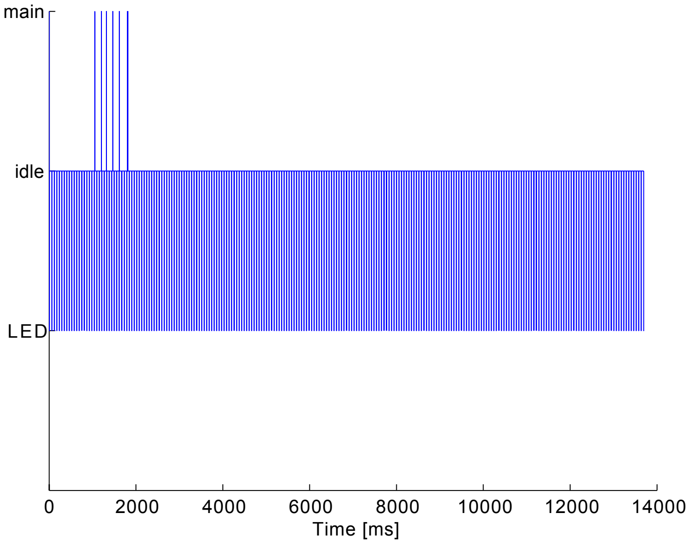


Figure 5.1: Execution of threads for the program listed in Exercise 49.

What you can see is that the BTnode spends most of the time in the idle thread. Periodically, it switches to the LED thread and a few times, the main thread was active. The LED thread is responsible for the periodic blinking of the LEDs, it becomes active approximately every 60ms *and takes about* 300μs *to execute. The main thread is responsible for capturing terminal input and launching the corresponding commands. Thus if you did not type* `trace` *while the buffer was filling, you will see only one spike to the main line at the very beginning of the trace. Otherwise (as shown in Fig. 5.1) you can see a spike for every letter of* `trace` *plus one when you pressed return. Parsing a keystroke takes about* 600μs*, executing the command after pressing enter takes much longer, approximately* 6ms. *When looking closely at the last spike, you may note that it actually consists of several spikes. This is due to the fact, that the* `trace` *command prints the status of the trace buffer to the terminal, but cannot do so in a single shot. It fills the UART buffer until it is full, then yields execution to the idle thread and is woken up when the buffer has become empty again to write the rest of the output.*

5. *In the previous step we have seen, that even when the BTnode seems to do nothing really useful, several threads are executed. To understand a little bit better how this actually works, we now do another trace capturing in addition to the threads also the occurrence of interrupts. To this purpose type* `trace mask`.

```
[es-ex3]$trace mask TRACEMASK
 0 Critical  Enter OFF
 1 Critical  Exit OFF
 2 Thread    Yield ON
 3 Thread    SetPrio ON
 4 Thread    Wait ON
 5 Thread    Sleep ON
 6 Interrupt Enter OFF
 7 Interrupt Exit OFF
 8 Trace Start ON
 9 Trace Stop ON
 10 User * ON
```

You get a numbered list of event types followed by either `ON` *or* `OFF`. *Typing* `trace mask 6` *redisplays this list, but now the event type 6, which is the begin of an interrupt service routine is set to* `ON`. *Repeat* *this for the event type 7. Now take a trace as explained in the previous steps, capture the event list in a file and display it using Matlab.*

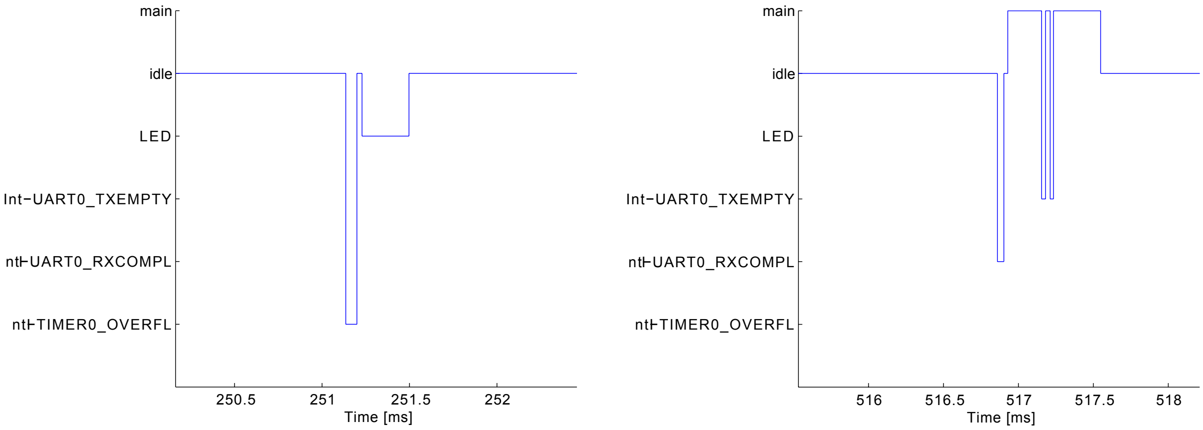


Figure 5.2: LED thread woken up by the timer interrupt. *Left:* LED thread woken up by the timer interrupt. *Right:* Main thread woken up by UART receive interrupt, causing transmit complete interrupts by echoing the terminal input.

Looking at Fig. 5.2, left side, you can see how the LED thread is triggered by the timer interrupt (`Int_TIMER0_OVERFL`*). On the right side of Fig. 5.2, it is shown that the main thread is activated after the occurrence of a UART receive interrupt (*`Int_UART0_TXCOMPL`*). Since the main thread echoes all received characters to the terminal, two UART transmit complete interrupts occur immediately after the activation of the main thread.*

Exercise 50 *The traces captured in the previous example show that most of the time is spent in a thread called* `idle`*, which was not started by our program. What is the purpose of this thread?*

Exercise 51 *When the tracing of interrupts is enabled, you can see timer interrupts. You can also see that a thread that sleeps always awakes immediately after these timer interrupts. Figure out the interval of these timer interrupts and think about what kind of restriction this implies for the* `NutSleep` *function.* **HINT:** *Remember that you can specify the sleep time in milliseconds.*

Explanation ***Tracing a Particular Piece of Code*:**
The interactive mode of the tracer tool is very simple to use but it does not allow to trace a particular piece of code in which you are interested. To do this, it has to be used in a different way.
You may be interested in what a particular function call does. Therefore you would like to start tracing immediately before this function is executed. You can do this as shown here:

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

int main(void) {

    // initializations
    NutTraceMaskSet(TRACE_TAG_INTERRUPT_ENTER);
    NutTraceMaskSet(TRACE_TAG_INTERRUPT_EXIT);

    // some code

    NutTraceInit(1000,TRACE_MODE_ONESHOT);
    // code you want to trace

    // some code
}
```

At the begin of the main routine you set the trace mask using the functions `NutTraceMaskSet` and `NutTraceMaskClear`. You find the macros that describe the types of events you want to trace in the `sys/tracer.h` header file. Then you start the trace using `NutTraceInit` immediately before the code you are interested in. The first parameter of this function determines the amount of items that are traced, the second parameter specifies whether tracing should be stopped when the trace buffer is full (`TRACE_MODE_ONESHOT`), or whether it should continuously overwrite the entries (`TRACE_MODE_CIRCULAR`), until tracing is stopped explicitly. The program shown above now automatically fills the trace buffer. You can either print it using the *trace* terminal command, or using the function `NutTracePrint`, which takes a single argument that determines how many trace entries shall be printed. If this argument is 0, the whole buffer is printed.

Exercise 52 *Trace the* `printf` *function. First use a string that is shorter than the length of the buffer (default is 64, may be changed using* `ioctl`*, see the avr-gcc manual for details), then a string that is longer. Enable the tracing of interrupts. Explain what you see.*

Chapter 6

Communication Using Bluetooth

6.1 Introduction

The Bluetooth technology is well suited to provide short-range wireless communications between electronic devices like e.g. mobile phones, laptops or PDAs. Without the need of a pre-established infrastructure, portable devices may create links and form Personal Area Networks (PANs).

This chapter addresses simple point-to-point communication between BTnodes. We will mainly concentrate on the interaction between the microcontroller and the Bluetooth radio and will – as far as possible – make use of pre-implemented data structures and functions of the BTNut system software. In doing so, the reader should gain some insight in the use of the thread/event-functionality of the Nut-OS and the low-level packet assembly routines provided by the BTnut API. To gain a certain confidence and understanding of Bluetooth communication, you can use the `bt-cmd` demo application.

We will have to familiarize the reader with certain details of the Bluetooth Specification [2]. In order to ease searching in the specification, all page numbers given in this tutorial refer to the page numbers of the PDF-document[1].

Section 6.2 presents the basic mechanisms that are used to access the Bluetooth radio capabilities. Therefore the interface between microcontroller and Bluetooth radio is explained. As an example, we take a closer look at the inquiry procedure used to discover other nearby Bluetooth devices. In Section 6.3 you will create wireless connections to other BTnodes and transmit short text messages.

6.2 Discovery of Bluetooth devices

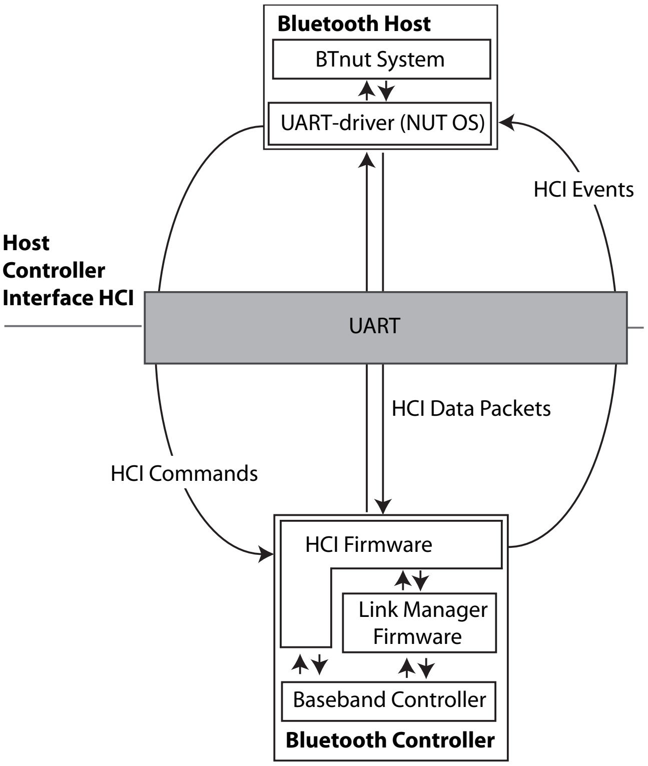
The Atmega128 microcontroller communicates with the Zeevo ZV4002 Bluetooth radio according to the principles defined in the *Host Controller Interface Functional Specification* [2].

In the following, we want to send an *Inquiry Command* to the Bluetooth controller. This command will cause the radio to enter inquiry mode and search for possible Bluetooth devices within communication range. The controller will count the total number of responding devices and collect a set of values for every single device. The value we are especially interested in is the *Bluetooth device address* of a discovered BTnode.

---

[1] We don't refer to the page numbers printed on the original document, since they are **not unique.**

Explanation *Host Controller Interface HCI*:
As depicted, the Host Controller Interface defines signaling and data exchange between the so-called *Bluetooth host* and the *Bluetooth controller*. The Bluetooth host can be seen as the microcontroller running the BTnut system software and driving the NutOS UART-driver. The Bluetooth controller is physically connected to the host system via the UART. The Bluetooth controller is located on the Bluetooth radio and comprises the HCI firmware, the link manager firmware and the baseband controller. *HCI commands* can be sent from the host to the controller to initiate radio communication and access configuration parameters. On the other hand, the controller uses *HCI events* to inform the host when something occurs. Finally, *HCI data packets* may be transmitted in both directions.



Exercise 53 *Each Bluetooth device is characterized by a unique Bluetooth device address. Find the device address (MAC) of your BTnode. How many bytes are needed to represent a Bluetooth device address?*

A HCI packet is defined as shown below.

```
struct bt_hci_pkt_cmd {
        u_char type;
        u_char payload[255];
};
```

The `type`-parameter is needed to distinguish between command, event and data packets. For our purpose, we set `type=0x01` to define a command packet. The `payload`-array reserves 255 bytes for the actual command packet as specified on page 509f of the Bluetooth specification [2]. It starts with a 2 byte OpCode which is divided into two fields, called Opcode Group Field (OGF) and OpCode Command Field (OCF). Note that the bit ordering of the packet definition follows the *Little Endian* format, i.e. the LSB is the first bit sent over the UART.

Exercise 54 *Open the Bluetooth specification on page 510 to figure out how HCI Command Packets are constructed in general. You will find a detailed description of the Inquiry Command on pages 531 and 532.*

Figure out, what the single entries of the following `bt_hci_pkt_cmd` *mean:*

```
    struct bt_hci_pkt_cmd pkt;
     pkt.type=0x01;
     pkt.payload[0]=0x01;
     pkt.payload[1]=0x04;
     pkt.payload[2]=0x05;
     pkt.payload[3]=0x33;
     pkt.payload[4]=0x8b;
     pkt.payload[5]=0x9e;
     pkt.payload[6]=0x05;
     pkt.payload[7]=0x05;
```

In particular, how long will this inquiry last and what is the maximum number of Bluetooth devices that can be found like this? **HINT:** *The general inquiry access code (GIAC) is 0x9E8B33 (see page 213 [2]).*

A function `inquiry` that sends an inquiry and displays the addresses of the found Bluetooth devices should look as follows: (Don't be confused if you are not familiar with all data types and functions – they will be explained later!)

```
struct btstack* stack;

void inquiry (u_char* arg){

//define a HCI command packet
struct bt_hci_pkt_cmd pkt;
// assemble the single bytes of the struct pkt (see previous exercise!)
// INSERT YOUR CODE HERE
// INSERT YOUR CODE HERE

// define a "command_response" structure
struct bt_hci_cmd_response wcmd;

//array for the storage of the answers of max. 10 devices
struct bt_hci_inquiry_result inquiry_result[10];

//initialize the cmd_response-structure
wcmd.ogfocf= ((0x01<<8)|(0x01<<2));
wcmd.cmd_handle= 0xFFFF;
wcmd.response=0;
wcmd.ptr= &inquiry_result;
wcmd.block=0;

//register the wcmd in the WaitQueue of the btstack
_bt_hci_setWaitQueue(stack,&wcmd);

//send the command packet ...
_bt_hci_send_pkt(stack,(u_char*)&pkt);

printf("Starting inquiry .....\n");
//wait for the inquiry to complete
// INSERT YOUR CODE HERE
// INSERT YOUR CODE HERE

printf("Inquiry done! \n");

// print inquiry_result[] to the terminal
// INSERT YOUR CODE HERE
// INSERT YOUR CODE HERE
```

```
}
```

First of all, we need a pointer to a variable of `struct btstack`-type for our function to work properly. This variable stores data for numerous devices, buffers and internal states. We need this structure for the definition of the UART-transport. Furthermore, the `btstack` structure stores a list of "signatures" of all uncompleted commands – or more precisely – a list of pointers to `bt_hci_com_response`-structures.

Explanation *`struct bt_hci_cmd_response`*:

```
struct bt_hci_cmd_response {
        u_short ogfocf;
        u_short cmd_handle;
        long response;
        void *ptr;
        HANDLE block;
};
```

The `ogfocf` is used to store the complete OpCode of the pending command. Setting the `cmd_handle` to 0xFFFF indicates that this command is not referring to an open baseband connection. When events return as a response to our Inquiry Command, the number of found devices will be stored in the component `long response`. The addresses of the found devices (together with several other values) will be stored at the location where the `void *ptr` is pointing. To indicate that our results are available, the `HANDLE block` will be #SIGNALED.

Explanation *`struct inquiry_result`*:

```
struct bt_hci_inquiry_result {
        bt_addr_t bdaddr;
        u_char page_scan_rep_mode :4;
        u_char page_scan_period_mode :4;
        u_long cod;
        u_short clock_offset;
        short rssi;
};
```

This struct stores all the collected data of one single discovered Bluetooth device. We are only interested in the `bt_addr_t`-component. As you already found out, the `bt_addr_t`-type is equivalent to a `u_char[6]`.

So we only send the Inquiry with the `_bt_hci_send_pkt`-function, pass an address to a `_bt_hci_setWaitQueue`-function and the result will be "automatically" stored in our prepared variables? Who is receiving and handling all the incoming events from the Bluetooth radio?

Answer: All the work is done by a THREAD called *"BTStack"*. This THREAD ...

- invokes a blocking `_bt_hci_get_pkt()`-function.
- searches for a matching `struct bt_hci_cmd_response` if an event arrives.
- dumps the payload of the event correctly.
- invokes a `EventPostAsync()` for the respective `HANDLE`.
- performs a final `NutThreadYield()`.

You should create the "BTStack"-THREAD in your main program by calling

```
stack = bt_hci_init(&BT_UART);
```

This function call simultaneously initializes the UART to the Bluetooth radio. Additonally, you should include the following header-files to ensure availability of all the functions and data types used so far:

```
#include <hardware/btn-hardware.h>
#include <terminal/btn-terminal.h>
#include <stdio.h>              // freopen
#include <dev/usartavr.h>       // NutRegisterDevice, APP_UART, UART_SETSPEED
#include <bt/bt_hci_dispatch.h> // for the setWaitQueue command
#include <sys/event.h>          // for NutEventWait
#include <bt/bt_hci_cmds.h>
```

Exercise 55 *Complete the* `inquiry` *function and register the command* `inquiry` *as a terminal command. Don't forget to initialize your hardware with* `btn_hardware_init()` *and* `btn_hardware_bt_on()`*. After having successfully implemented your Inquiry command, find out which BTnodes you have discovered!*

Exercise 56 *Use the OS-Tracer from Chapter 5 to trace your Inquiry command and see how the BTStack fetches the single events. You can start the tracer with* trace oneshot *and stop it with* trace stop*. Read on page 532 in [2] about which event packets may arrive at the Bluetooth host and identify those events in the trace plot.* **Hint:** *For this you will need to temporarily disable the LED thread.*

6.3 Creating Connections and Sending Data Packets

One of the parameters we send to the Bluetooth Controller with our Inquiry command was the *general inquiry access code* (GIAC). The Bluetooth Controller rearranges the Inquiry command packet in such a way, that the packet sent over the air begins with this GIAC. Actually, all transmissions over the physical channel *have* to begin with such an access code.

With the reception of a Bluetooth address we gained some knowledge that we can exploit to access the channel once more and create a connection to another device: We have to pass this address as a parameter to a "Create-Connection-command" which in turn causes the Bluetooth Controller to initiate the *Page procedure.* During this procedure, the Link Manager on the Bluetooth Controller tries to establish a link level connection to another device. Therefore, messages beginning with a *device access code* DAC are generated. The DAC is derived from the paged device's Bluetooth address.

Explanation *Terminal command* uartdebug:
The terminal command `bt uartdebug 1` displays all HCI traffic on the UART. Bytes starting with a "w" are sent to the Bluetooth Controller, those starting with a "r" are received from the Controller. Events and Commands can be interpreted as follows:

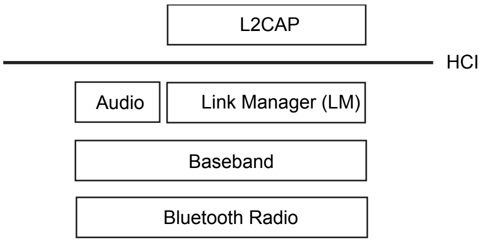
```
bytes                 1  |     2      |       3       |       4       |    5
--------------------------------------------------------------------------------
HCI command packet:   1  |          Opcode            |parameterlength| parameter
HCI event   packet:   4  | event code |parameterlength| parameter
```

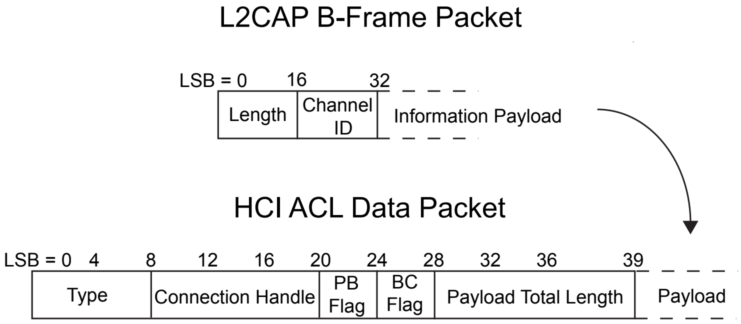
Exercise 57 *Compile and upload the* bt-cmd *application. Type* bt uartdebug 1*. Start an* bt inquiry *and create a new connection to an arbitrary BTnode using the command* bt con*. Identify the impinging events that are caused by the* bt con*-command! Analyze the received* ConnectionCompleteEvent *to figure out if we established a synchronous (SCO) or asynchronous (ACL) connection.* **Hint:** *HCI events are listed starting on page 695 [2] according to their Event Code.*

Now you should be connected with another BTnode i.e. both radios should be synchronized in terms of slot timing, frequency hopping sequence and access code to the physical channel. You can check your connections with the `contable`-command. As you see, a *connection handle* has been assigned to your connection. Those handles are used to identify connections between Bluetooth devices.

Once a connection is established, we want to send simple text messages to another BTnode.

Explanation *Logical Link Control and Adaptation Protocol L2CAP*:
The Logical Link Control and Adaptation Protocol (L2CAP) resides directly above the Host Controller Interface (HCI). At the L2CAP layer, communication is based on so-called *channels*. This abstraction allows multiplexing and de-multiplexing of multiple channels over a shared link. Furthermore, L2CAP carries out segmentation and reassembly of application data for higher protocol layers. The figure shows a L2CAP basic information frame (B-frame) packet, starting with 2 bytes for the *length* of the information payload. Here, the length indicates the size of the payload in bytes. Bytes number 3 and 4 represent the *channel ID*. The rest of the packet is reserved for the actual payload. Clearly, the size of the payload is limited. But for the short messages we want to send in this tutorial we won't get into conflict with those payload limits.





Also an HCI asynchronous connection-oriented (ACL) data packet is illustrated. To distinguish between HCI commands, events and data packets, the *type* parameter has to be set. The *packet boundary* PB flag is used to indicate the first packet (PB=2) or a continuing fragment packet (PB=1) of a higher layer message. By setting the *broadcast flag* BC=0 a point-to-point message is defined. Finally, the payload length (again in bytes) concludes the header of the HCI ACL Packet. The body of the HCI ACL Packet consists of the L2CAP B-Frame Packet in our example. More information about L2CAP can be found in [2] on pages 963ff.

In the following, we want to write a `transmit` function which sends a HCI ACL Data packet of the form

```
u_char hci_acl_pkt[total_size];
  hci_acl_pkt[1]          =  ... ;
  hci_acl_pkt[2]          =  ... ;
             ...
  hci_acl_pkt[total_size] =  ... ;
```

We will send this packet using the function *bt_hci_send_acl_pkt* with the following signature:

```
bt_hci_send_acl_pkt(struct btstack *stack, u_short con_handle, u_char pb_flag,
       u_char bc_flag, u_short payload_total_length, struct bt_hci_pkt_acl *pkt);
```

As you can see, this function automatically sets the entries of the HCI ACL Packet header. However, it is still necessary to allocate memory (`total_size`) for all the entries of the `hci_acl_pkt`-packet, although the first entries don't have to be specified.

Exercise 58 *Copy the* bt-cmd-*application and add a new function called* transmit. *Register this function as a terminal command that takes a connection handle, a channel ID and a* string-*message as arguments. Define a* hci_acl_pkt *packet that allocates enough memory for a complete HCI ACL packet with an information payload of 20 characters and transmit it.* **Hint:** *You don't have to know any details about the* bt_hci_pkt_acl *-struct. Just cast your* hci_acl_pkt *packet accordingly!*

In order to receive short messages, the following `receive`-function has to be defined:

```
struct bt_hci_pkt_acl* receive(void *arg, struct bt_hci_pkt_acl *pkt, bt_hci_con_handle_t con_handle,
 u_char pb_flag, u_char bc_flag, u_short len, u_long t_arrive)
{
        u_char* l2cap_hdr = pkt->payload;
        u_char* l2cap_data;
        u_short chan_id;

        chan_id = l2cap_hdr[2] | (l2cap_hdr[3] << 8);

        l2cap_data = &l2cap_hdr[4];

        printf("message received on channel %d: %s\n", chan_id, l2cap_data);
        return pkt;
}
```

If you now define the packet

```
u_char acl_pkt[120];
```

a register the `receive`-function as a callback

```
bt_hci_register_acl_cb(stack, receive, (struct bt_hci_pkt_acl*)acl_pkt, NULL);
```

messages sent to your BTNode will be displayed automatically on the terminal.

Exercise 59 *Test your* `transmit`*-function by sending a short message to a SUPERVISOR-node that uses a preloaded application. Use channel 65 for sending this message. If you have implemented everything corretly so far, you will receive an acknowledgment from the SUPERVISOR-node immediately.*

Optional Exercise 60 *Check if some of your neighbors have already finished exercise 59. Try to communicate with another group doing this tutorial. Optionally, try to combine commands from* `bt-cmd` *such as* `name`, `rname`, `role`, `roleset` *with* `transmit` *to get status information from other nodes.*

Appendix A

Software Versions Used

The Embedded Systems lecture held in spring 2006 at ETH Zurich used the following software versions:

`AVR Studio 4 v412SP1 build462`

`Silabs CP2101 USB to UART Bridge 20050102`

`WinAVR 20060125`

`doxygen 1.4.6`

`Java 2 SDK 1.5.0_06`

`RXTX-2.0-7pre1`

`javax_comm-2_0_3-solsparc`

`eclipse 3.1.2`

`org.eclipse.cdt.sdk-3.0.2`

`easyshell-1.2.0`

`ZOC Terminal 5.0.6`

`Emacs 21.2`

`Matlab 7.1r14`

Appendix B

Solutions

Chapter 2 – First Steps in BTnode Programming

Solution 1 *The external memory is connected to Port A (address and data bus) and Port C (address bus) as well as to three pins of Port G that are assigned WR*, RD* and ALE functions. A trandparent D-type latch is used to multiplex Port A in order to save IO space at the cost of longer access times. The BTnode rev3 further uses two pins of Port B (PB7 and PB6) to bank switch 4 banks of 60 kbytes external memory (the processor can only make use of one bank at a time).*

The LED/power latch is attached to Port C. This allows to multiplex the address bus and the latch used for the LEDs and power/radio configuration onto Port C. It is controlled via pin PB5 (LATCH_SELECT).

Pin	**Function**
PC0	Blue LED
PC1	Red LED
PC2	Yellow LED
PC3	Green LED
PC4	ON_VCC_IO
PC5	ON_VCC_CC
PC6	ON_VCC_BT
PC7	RESET_BT

The first problem from this hardware setup is that the software has to keep track of the states of the latch outputs since it is impossible to inquire the current latch state at the port outputs in software. The second problem is that the routine for driving the latch should not be interruptible yet it has to be short so that it does not interfere too much with other software components timing requirements.

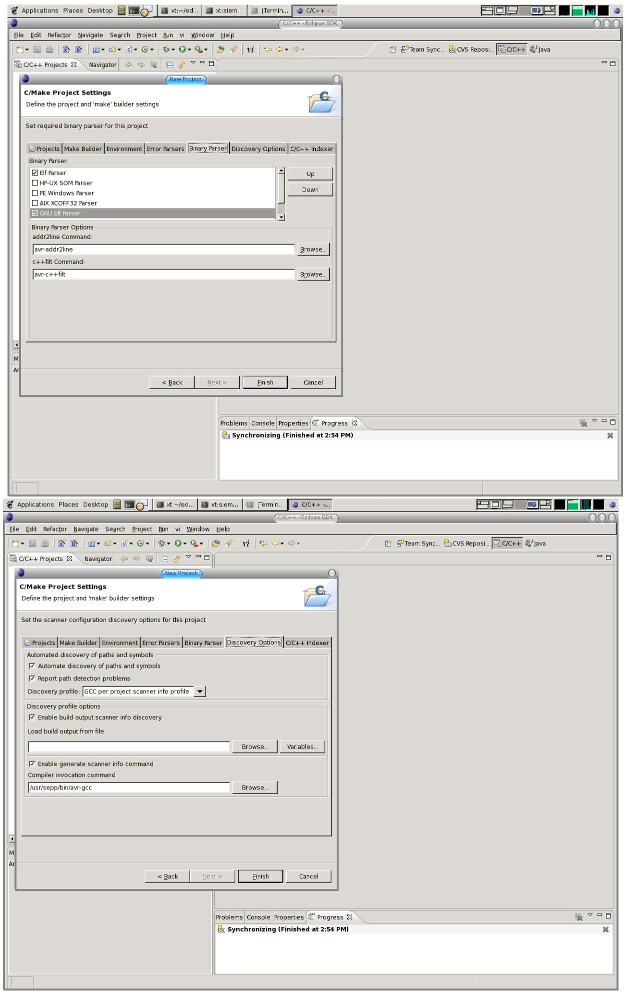
Solution 2 *Project setup in Eclipse follows the menu functions and is straightforward.*

Solution 3 *The indexing function is a very powerful tool within Eclipse. It is much faster than global search and can discriminate definitions, declarations and references.*

Solution 4 *The BTnut system software supports scheduling of different patterns at the LEDs, a simple, yet powerful userinterface on embedded systems.*

Solution 5 *The* `Content Assist` *function is yet another powerful feature when using Eclipse. It allows to quickly navigate different library functions and gives quick hints to their correct usage/syntax. Use it to add an additional LED pattern to a simple BTnut application:*

```
// hardware init
btn_hardware_init();
```



C/Make Project Settings
Define the project and 'make' builder settings

Set required binary parser for this project

Binary Parser:
☑ Elf Parser
☐ HP-UX SOM Parser
☐ PE Windows Parser
☐ AIX XCOFF32 Parser
☑ GNU Elf Parser

Binary Parser Options
addr2line Command: avr-addr2line
c++filt Command: avr-c++filt

Synchronizing (Finished at 2:54 PM)

C/Make Project Settings
Define the project and 'make' builder settings

Set the scanner configuration discovery options for this project

Automated discovery of paths and symbols
☑ Automate discovery of paths and symbols
☑ Report path detection problems
Discovery profile: GCC per project scanner info profile

Discovery profile options
☑ Enable build output scanner info discovery
Load build output from file
☑ Enable generate scanner info command
Compiler invocation command: /usr/sepp/bin/avr-gcc

Synchronizing (Finished at 2:54 PM)

Figure B.1: Use the Eclipse menues to navigate the the new projects wizard.

```
btn_led_init(1);
btn_led_add_pattern(BTN_LED_PATTERN_HALF, 0, 10, BTN_LED_INFINITE);
```

Solution 6 *Not yet.*

Solution 7 *The Atmel AVR is a family of simple yet very versatile microcontrollers based on an 8-bit RISC core running single cycle instructions. AVR instructions are tuned to decrease the size of the program whether the code is written in C or Assembly.*

The AVR Libc implements simple fixed point arithmetic functions using commands from the AVR instruction set such as: `lds`*,* `xor` *or* `fmul`*.*

Due to it's simple core architecture mathematical operations requiring complicated processor hardware such as multiply and divide are omitted. However there are specialized libraries available that implement sets of mathematical functions (fixed point anf floating point) or even cryptography on the AVR architecture.

Some examples with online resources are:

- *mikroPascal for AVR*
- *mikroBasic for AVR*

- *Procyon AVRlib*

Solution 8 *Not yet.*

Solution 9 *Not yet.*

Solution 10 *You should see the following output:*

```
C:\Documents and Settings\es2005>avr-as --version
GNU assembler version 2.15 (avr) using BFD version 2.15 + coff-avr-patch (20030831)

C:\Documents and Settings\es2005>avr-gcc -v
Reading specs from C:/WinAVR/bin/../lib/gcc/avr/3.4.3/specs
Configured with: ../gcc-3.4.3/configure --prefix=m:/WinAVR --build=mingw32
--host=mingw32 --target=avr --enable-langu
Thread model: single
gcc version 3.4.3

C:\Documents and Settings\es2005>avr-ld -v
GNU ld version 2.15 + coff-avr-patch (20030831)

C:\Documents and Settings\es2005>uisp --version
uisp version 20050207
(C) 1997-1999 Uros Platise, 2000-2003 Marek Michalkiewicz
uisp is free software, covered by the GNU General Public License.
You are welcome to change it and/or distribute copies of it under
the conditions of the GNU General Public License.

C:\Documents and Settings\es2005>avrdude -v
avrdude: Version 4.4.0cvs
         Copyright (c) 2000-2004 Brian Dean, http://www.bdmicro.com/
         System wide configuration file is "C:\WinAVR\bin\avrdude.conf"
avrdude: no programmer has been specified on the command line or the config file
         Specify a programmer using the -c option and try again
```

Solution 11 *You should see the following output:*

```
avrdude -help
Usage: avrdude [options]
Options:
  -p <partno>                Required. Specify AVR device.
  -b <baudrate>              Override RS-232 baud rate.
  -B <bitclock>              Specify JTAG/STK500v2 bit clock period (us).
  -C <config-file>           Specify location of configuration file.
  -c <programmer>            Specify programmer type.
  -D                         Disable auto erase for flash memory
  -P <port>                  Specify connection port.
  -F                         Override invalid signature check.
  -e                         Perform a chip erase.
  -U <memtype>:r|w|v:<filename>[:format]
                             Memory operation specification.
                             Multiple -U options are allowed, each request
                             is performed in the order specified.
  -n                         Do not write anything to the device.
  -V                         Do not verify.
  -u                         Disable safemode, default when running from a script.
  -s                         Silent safemode operation, will not ask you if
                             fuses should be changed back.
  -t                         Enter terminal mode.
  -E <exitspec>[,<exitspec>] List programmer exit specifications.
  -y                         Count # erase cycles in EEPROM.
  -Y <number>                Initialize erase cycle # in EEPROM.
  -v                         Verbose output. -v -v for more.
  -q                         Quell progress output. -q -q for less.
  -?                         Display this usage.

avrdude project: <URL:http://savannah.nongnu.org/projects/avrdude>
```

Solution 12 *You should see the following output:*

```
avrdude: AVR device initialized and ready to accept instructions
Reading | ################################################## | 100% 0.02s
avrdude: Device signature = 0x1e9702
avrdude: safemode: Fuses OK
avrdude done.  Thank you.
```

Solution 13 *You should see the following output:*

```
avrdude: AVR device initialized and ready to accept instructions
Reading | ################################################## | 100% 0.02s
avrdude: Device signature = 0x1e9702
avrdude: erasing chip
avrdude: safemode: Fuses OK
avrdude done.  Thank you.
```

You should see the following output:

```
avrdude: AVR device initialized and ready to accept instructions
Reading | ################################################## | 100% 0.02s
avrdude: Device signature = 0x1e9702
avrdude: reading input file "bt-cmd.btnode3.hex"
avrdude: writing flash (66182 bytes):
Writing | ################################################## | 100% 14.47s
avrdude: 66182 bytes of flash written
avrdude: safemode: Fuses OK
avrdude done.  Thank you.
```

Solution 14 *Not yet.*

Solution 15 *You should see the following output:*

```
avrdude: AVR device initialized and ready to accept instructions
Reading | ################################################## | 100% 0.02s
avrdude: Device signature = 0x1e9702
avrdude: NOTE: FLASH memory has been specified, an erase cycle will be performed
         To disable this feature, specify the -D option.
avrdude: erasing chip
avrdude: reading input file "uart-echo.btnode3.hex"
avrdude: writing flash (13410 bytes):
Writing | ################################################## | 100% 2.95s
avrdude: 13410 bytes of flash written
avrdude: verifying flash memory against uart-echo.btnode3.hex:
avrdude: load data flash data from input file uart-echo.btnode3.hex:
avrdude: input file uart-echo.btnode3.hex contains 13410 bytes
avrdude: reading on-chip flash data:
Reading | ################################################## | 100% 1.48s
avrdude: verifying ...
avrdude: 13410 bytes of flash verified
avrdude: safemode: Fuses OK
avrdude done.  Thank you.
```

Solution 16 *You should see the following output:*

```
make btnode3
echo "#define PROGRAM_VERSION "\""20060405-1501"\" > program_version.tmp
mv -f program_version.tmp program_version.h
avr-gcc -c -mmcu=atmega128 -Os -Wall -Werror -Wstrict-prototypes -Wa,-ahlms=bt-cmd.btnode3.lst -D__HARVARD_ARCH__
-D__BTNODE3__ -DUSE_USART0 -DUART0_READMULTIBYTE -DUART0_NO_SW_FLOWCONTROL -DUSE_USART1 -DUART1_READMULTIBYTE
-DUART1_NO_SW_FLOWCONTROL -I../..//btnode/include -I../..//../nut/include bt-cmd.c -o bt-cmd.btnode3.o
avr-gcc bt-cmd.btnode3.o  ../..//lib/btnode3/nutinit.o -Wl,--start-group -L../..//lib/btnode3 -mmcu=atmega128
-Wl,--defsym=main=0,-Map=bt-cmd.btnode3.map,--cref -L../..//lib/btnode3 -lnutos -lnutdev -lnutarch -lnutcrt
-lbt -lhardware -lcm -leepromdb -lsuart -lled -lcc -ldebug -lmhop -lsync -lutils -lterminal -lsupport -lcdist
-Wl,--end-group -o bt-cmd.btnode3.elf
avr-size bt-cmd.btnode3.elf
   text    data     bss     dec     hex filename
  60416    4062    2814   67292   106dc bt-cmd.btnode3.elf
avr-objcopy -O ihex bt-cmd.btnode3.elf bt-cmd.btnode3.hex
rm bt-cmd.btnode3.elf
```

You should see the following output:

```
make btnode3 upload
make: Nothing to be done for `btnode3'.
make burn.btnode3
make[1]: Entering directory `/home/beutel/eclipse/btnut/app/bt-cmd'
avrdude -pm128 -cavrispv2 -Pusb -s     -U flash:w:bt-cmd.btnode3.hex:i

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.01s
```

```
avrdude: Device signature = 0x1e9702
avrdude: NOTE: FLASH memory has been specified, an erase cycle will be performed
         To disable this feature, specify the -D option.
avrdude: erasing chip
avrdude: reading input file "bt-cmd.btnode3.hex"
avrdude: writing flash (64478 bytes):

Writing | ################################################## | 100% 5.19s

avrdude: 64478 bytes of flash written
avrdude: verifying flash memory against bt-cmd.btnode3.hex:
avrdude: load data flash data from input file bt-cmd.btnode3.hex:
avrdude: input file bt-cmd.btnode3.hex contains 64478 bytes
avrdude: reading on-chip flash data:

Reading | ################################################## | 100% 4.17s

avrdude: verifying ...
avrdude: 64478 bytes of flash verified

avrdude: safemode: Fuses OK

avrdude done.  Thank you.

make[1]: Leaving directory '/home/beutel/eclipse/btnut/app/bt-cmd'
```

Solution 17 *Not yet.*

Solution 18 *Not yet.*

Solution 19 *Not yet.*

Solution 20 *Not yet.*

Chapter 3 – Device Level Programming

Solution 21 *The blue LED is connected to pin 0 of port C. Since port C is the upper byte of the address bus, the value* `0x0100` *on the address bus switches on the blue LED. Since the function* `write_led` *shifts the argument value by 8 bits, you have to use* `write_led(0x01)` *to switch on the blue LED.*

Solution 22 – Sample Code

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

void shortpause(u_short duration)
{
    u_short i;
    for (i=0;i<duration;i++) {
    }
}

void pause(u_short duration)
{
    u_short i;
    for (i=0;i<duration;i++) {
        shortpause(0xffff);
    }
}

void write_led(u_char value)
{
    volatile u_char * pointer;
    u_char dummy;

    pointer  = (u_char *) ( ((u_short)value) << 8);
    dummy    = *pointer;

    sbi(PORTB, 5);
```

```
    asm volatile  ("nop" ::);
    cbi(PORTB, 5);
}

int main(void)
{
    int toggle = 0;

    sbi(DDRB, 5);
    while (1) {
        if (toggle) {
            toggle = 0;
            write_led(0x01);
        }
        else {
            toggle = 1;
            write_led(0x00);
        }
        pause(12);
    }
    return 0;
}
```

Solution 23 *I have implemented the pause with the help of a* `shortpause()` *function (see solution to exercise 20). Calling* `pause(12)` *resulted in a pause of approximately 1 second. Calling* `pause(12)` *runs the loop in* `shortpause` $12 \cdot$ `0xffff` $= 768'000$ *times. Now we look at the list file:*

```
  ...
  13                      shortpause:
  14                      /* prologue: frame size=0 */
  15                      /* prologue end (size=0) */
  16 0000 20E0                    ldi r18,lo8(0)
  17 0002 30E0                    ldi r19,hi8(0)
  18 0004 2817                    cp r18,r24
  19 0006 3907                    cpc r19,r25
  20 0008 28F4                    brsh .L8
  21                      .L6:
  22 000a 2F5F                    subi r18,lo8(-(1))
  23 000c 3F4F                    sbci r19,hi8(-(1))
  24 000e 2817                    cp r18,r24
  25 0010 3907                    cpc r19,r25
  26 0012 D8F3                    brlo .L6
  27                      .L8:
  28 0014 0895                    ret
  29                      /* epilogue: frame size=0 */
  30                      /* epilogue: noreturn */
  31                      /* epilogue end (size=0) */
  32                      /* function shortpause size 11 (11) */
  33                             .size        shortpause, .-shortpause
  ...
```

The loop is coded in the lines 21 to 26, line 21 is not an instruction, thus the loop is 5 instructions long. So in 1 second, approximately $768'000 \cdot 5 = 3'930'000$ *instructions are executed. Assuming that every instruction takes one cycle results in a clock frequency of 3.9 MHz.*

Looking at the instruction set summary in the atmega manual tells us that the `brlo` *takes two cycles, thus we get to a clock frequency of* $768'000 \cdot 6 \ Hz = 4.7MHz$.

The clock frequency is actually 7.3 MHz. The error of course results from the extremely inaccurate measurement of 'one second'.

Solution 24 – Sample Code

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

// COMPUTE ADC values corresponding to 1 and 2 Volts:
// If battery voltage is 1V, the BAT_SENSE signal is 0.5V.
// The reference voltage is 3.3V and corresponds to the ADC value 1024.
// Therefore 0.5V corresponds to an ADC value of 1024/3.3*0.5=155.
// A battery voltage is 2V, ADC value is 310.
#define BAT_1_VOLT 155
#define BAT_2_VOLT 310
```

```
void shortpause(u_short duration)
{
    u_short i;
    for (i=0;i<duration;i++) {
    }
}

void pause(u_short duration)
{
    u_short i;
    for (i=0;i<duration;i++) {
        shortpause(0xffff);
    }
}

void write_led(u_char value)
{
    volatile u_char * pointer;
    u_char dummy;

    pointer  = (u_char *) ( ((u_short)value) << 8);
    dummy    = *pointer;

    sbi(PORTB, 5);
    asm volatile  ("nop" ::);
    cbi(PORTB, 5);
}

int get_battery_voltage(void) {

    int result;

    ADMUX |= 1<<MUX0;
    ADMUX |= 1<<MUX1;
    ADCSRA |= 1<<ADPS0;
    ADCSRA |= 1<<ADPS1;
    ADCSRA |= 1<<ADPS2;
    ADCSRA |= 1<<ADEN;
    ADCSRA |= 1<<ADSC;
    while (ADCSRA & (1<<ADSC)) ;

    result = ADCL;
    result |= ADCH << 8;

    return result;
}

int main(void)
{
    int battery_voltage = 0;

    DDRB |= 1<<DDB5;
    while (1) {

        battery_voltage = get_battery_voltage();

        if (battery_voltage < BAT_1_VOLT) {
           write_led(0x02);
        }
        else {
            if (battery_voltage < BAT_2_VOLT) {
               write_led(0x04);
            }
            else {
               write_led(0x08);
            }
        }
        pause(12);
        write_led(0x01);
        pause(12);
    }
    return 0;
}
```

Solution 25 – Sample Code

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

```
// COMPUTE ADC values corresponding to 1 and 2 Volts:
// If battery voltage is 1V, the BAT_SENSE signal is 0.5V.
// The reference voltage is 3.3V and corresponds to the ADC value 1024.
// Therefore 0.5V corresponds to an ADC value of 1024/3.3*0.5=155.
// A battery voltage is 2V, ADC value is 310.
#define BAT_1_VOLT 155
#define BAT_2_VOLT 310

void shortpause(u_short duration)
{
    u_short i;
    for (i=0;i<duration;i++) {
    }
}

void pause(u_short duration)
{
    u_short i;
    for (i=0;i<duration;i++) {
        shortpause(0xffff);
    }
}

void write_led(u_char value)
{
    volatile u_char * pointer;
    u_char dummy;

    pointer  = (u_char *) ( ((u_short)value) << 8);
    dummy    = *pointer;

    sbi(PORTB, 5);
    asm volatile  ("nop" ::);
    cbi(PORTB, 5);
}

int get_battery_volt(void) {

    int result;

    ADMUX |= 1<<MUX0;
    ADMUX |= 1<<MUX1;
    ADCSRA |= 1<<ADPS0;
    ADCSRA |= 1<<ADPS1;
    ADCSRA |= 1<<ADPS2;
    ADCSRA |= 1<<ADEN;
    ADCSRA |= 1<<ADSC;
    while (ADCSRA & (1<<ADSC)) ;

    result = ADCL;
    result |= ADCH << 8;

    return result;
}

static void timer3IRQ(void *arg)
{
    int battery_voltage = get_battery_volt();

    if (battery_voltage < BAT_1_VOLT) {
       write_led(0x02);
    }
    else {
        if (battery_voltage < BAT_2_VOLT) {
           write_led(0x04);
        }
        else {
           write_led(0x08);
        }
    }
    // Reset the counter to non-zero value, see expl. in main routine.
    TCNT3H = 0x21;
    TCNT3L = 0x64;
}

int main(void)
{
    int toggle = 0;

    DDRB |= 1<<DDB5;
```

```
    NutRegisterIrqHandler(&sig_OVERFLOW3, timer3IRQ, 0);
    // 16 bit timer without prescaler (clock frequency 7.3MHz)
    // -> overflows once every 0xffff*(1/7.3E6)s=9ms
    // For an overflow every 2s, prescaler should be
    // 2s/9ms = 223. The closest value is 256 (see table on page 135).
    // This gives an overflow every 2.3s. This could be adjusted by
    // setting the counter value to 0.3/2.3*0xfffff = 0x2164 after
    // every overflow, thus at the end of the timer interrupt routine
    TCCR3B |= 1<<CS32;
    ETIMSK |= 1<<TOIE3;


    while (1) {
        if (toggle) {
            toggle = 0;
            write_led(0x01);
        }
        else {
            toggle = 1;
            write_led(0x00);
        }
        pause(10);
    }

    return 0;
}
```

Solution 26 – Sample Code

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

// COMPUTE ADC values corresponding to 1 and 2 Volts:
// If battery voltage is 1V, the BAT_SENSE signal is 0.5V.
// The reference voltage is 3.3V and corresponds to the ADC value 1024.
// Therefore 0.5V corresponds to an ADC value of 1024/3.3*0.5=155.
// A battery voltage is 2V, ADC value is 310.
#define BAT_1_VOLT 155
#define BAT_2_VOLT 310

void shortpause(u_short duration)
{
    u_short i;
    for (i=0;i<duration;i++) {
    }
}

void pause(u_short duration)
{
    u_short i;
    for (i=0;i<duration;i++) {
        shortpause(0xffff);
    }
}

void write_led(u_char value)
{
    volatile u_char * pointer;
    u_char dummy;

    pointer  = (u_char *) ( ((u_short)value) << 8);
    dummy    = *pointer;

    sbi(PORTB, 5);
    asm volatile  ("nop" ::);
    cbi(PORTB, 5);
}

int get_battery_volt(void) {

    int result;

    ADMUX |= 1<<MUX0;
    ADMUX |= 1<<MUX1;
    ADCSRA |= 1<<ADPS0;
    ADCSRA |= 1<<ADPS1;
    ADCSRA |= 1<<ADPS2;
    ADCSRA |= 1<<ADEN;
    ADCSRA |= 1<<ADSC;
```

```
    while (ADCSRA & (1<<ADSC)) ;

    result = ADCL;
    result |= ADCH << 8;

    return result;
}

static void timer3IRQ(void *arg)
{
    int battery_voltage = get_battery_volt();

    if (battery_voltage < BAT_1_VOLT) {
       write_led(0x02);
    }
    else {
        if (battery_voltage < BAT_2_VOLT) {
           write_led(0x04);
        }
        else {
           write_led(0x08);
        }
    }
}

int main(void)
{
    int toggle = 0;

    DDRB |= 1<<DDB5;

    NutRegisterIrqHandler(&sig_OUTPUT_COMPARE3A, timer3IRQ, 0);
    // 16 bit timer without prescaler (clock frequency 7.3MHz)
    // -> overflows once every 0xffff*(1/7.3E6)s=9ms
    // For an overflow every 2s, prescaler should be
    // 2s/9ms = 223. The closest value is 256 (see table on page 135).
    // This gives an overflow every 2.3s.
    TCCR3B |= 1<<CS32;
    // To get an interrupt every 2s, the interrupt should be triggered
    // when the counter reaches 2/2.3*0xffff=0xde9a.
    OCR3AH = 0xde;
    OCR3AL = 0x9a;
    // Enable this interrupt
    ETIMSK |= 1<<OCIE3A;

    // THE ADVANTAGE of the CTC over the solution for ex. 23 is that the interval
    // can be adjusted more precisely. In the previous mode you loose the time
    // that elapses between the moment the interrupt is triggered and the moment
    // the timer registers are reset.

    while (1) {
        if (toggle) {
            toggle = 0;
            write_led(0x01);
        }
        else {
            toggle = 1;
            write_led(0x00);
        }
        pause(10);
    }

    return 0;
}
```

Solution 27 *In my case, i measured an execution time of 0.25 ms without* `printf()` *and 1.2 ms with* `printf()`*.*

Solution 28 *Not yet.*

Solution 29 – Sample Code

```
#include <hardware/btn-hardware.h>
#include <dev/irqreg.h>
#include <stdio.h>                 // freopen
```

```
#include <io.h>                     // _ioctl
#include <dev/usartavr.h>           // NutRegisterDevice, APP_UART, UART_SETSPEED
#include <sys/timer.h>

// COMPUTE ADC values corresponding to 1 and 2 Volts:
// If battery voltage is 1V, the BAT_SENSE signal is 0.5V.
// The reference voltage is 3.3V and corresponds to the ADC value 1024.
// Therefore 0.5V corresponds to an ADC value of 1024/3.3*0.5=155.
// A battery voltage is 2V, ADC value is 310.
#define BAT_1_VOLT 155
#define BAT_2_VOLT 310

int adc_done = 0;
int battery_voltage = 0;

void shortpause(u_short duration)
{
    u_short i;
    for (i=0;i<duration;i++) {
    }
}

void pause(u_short duration)
{
    u_short i;
    for (i=0;i<duration;i++) {
        shortpause(0xffff);
    }
}

void write_led(u_char value)
{
    volatile u_char * pointer;
    u_char dummy;

    pointer  = (u_char *) ( ((u_short)value) << 8);
    dummy    = *pointer;

    sbi(PORTB, 5);
    asm volatile  ("nop" ::);
    cbi(PORTB, 5);
}

int get_battery_volt(void)
{
    int result;

    ADMUX  |= 1<<MUX0;
    ADMUX  |= 1<<MUX1;
    ADCSRA |= 1<<ADPS0;
    ADCSRA |= 1<<ADPS1;
    ADCSRA |= 1<<ADPS2;
    ADCSRA |= 1<<ADEN;
    ADCSRA |= 1<<ADSC;
    while (ADCSRA & (1<<ADSC)) ;

    result = ADCL;
    result |= ADCH << 8;

    return result;
}

static void timer3IRQ(void *arg)
{
    battery_voltage = get_battery_volt();

    if (battery_voltage < BAT_1_VOLT) {
       write_led(0x02);
    }
    else {
        if (battery_voltage < BAT_2_VOLT) {
           write_led(0x04);
        }
        else {
           write_led(0x08);
        }
    }
    // Reset the counter to non-zero value, see expl. in main routine.
    TCNT3H = 0x21;
    TCNT3L = 0x64;
    adc_done = 1;
```

```
}

int init_stdout(void)
{
    u_long baud = 57600;

    sbi( PORTD, 2);
    NutRegisterDevice(&APP_UART, 0, 0);
    freopen(APP_UART.dev_name, "r+", stdout);
    _ioctl(_fileno(stdout), UART_SETSPEED, &baud);
    return 1;
}

int main(void)
{
    init_stdout();
    printf("Hello world!\n");
    DDRB |= 1<<DDB5;

    NutRegisterIrqHandler(&sig_OVERFLOW3, timer3IRQ, 0);
    // 16 bit timer without prescaler (clock frequency 7.3MHz)
    // -> overflows once every 0xffff*(1/7.3E6)s=9ms
    // For an overflow every 2s, prescaler should be
    // 2s/9ms = 223. The closest value is 256 (see table on page 135).
    // This gives an overflow every 2.3s. This could be adjusted by
    // setting the counter value to 0.3/2.3*0xfffff = 0x2164 after
    // every overflow, thus at the end of the timer interrupt routine
    TCCR3B |= 1<<CS32;
    ETIMSK |= 1<<TOIE3;

    while (1) {
        while (adc_done == 0) {
            pause(1);
        }
        printf("Battery voltage is %d units\n",battery_voltage);
        adc_done = 0;
    }

    return 0;
}
```

Solution 30 *There are a few different cases to consider here:*

`battery_voltage_millivolt = (3300*battery_voltage_raw)/512;`

This does not work. in my case `battery_voltage_raw` *is 380 units, thus* $3300 \cdot 380 = 1254000$ *is far larger than what can be put in an int (16 bit) with a maximal value of* $+32767$*. Signed int is no better, the maximum is* $+65535$*. In contrast a signed long overflows at* $+2147483647$*, which is sufficient for this case.*

`battery_voltage_millivolt = 3300*(battery_voltage_raw/512);`

This also does not work, because $380/512 = 0$ *(its integers!).*

Here is the final solution:

```
int battery_voltage_raw, battery_voltage_millivolt;
battery_voltage_millivolt = (3300*(long)battery_voltage_raw)/512;
```

Solution 31 – Sample Code

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

// COMPUTE ADC values corresponding to 1 and 2 Volts:
// If battery voltage is 1V, the BAT_SENSE signal is 0.5V.
// The reference voltage is 3.3V and corresponds to the ADC value 1024.
// Therefore 0.5V corresponds to an ADC value of 1024/3.3*0.5=155.
// A battery voltage is 2V, ADC value is 310.
#define BAT_1_VOLT 155
#define BAT_2_VOLT 310
```

```
int adc_done = 0;
int battery_voltage = 0;

u_char temp_sreg;

void shortpause(u_short duration)
{
    u_short i;
    for (i=0;i<duration;i++) {
    }
}

void pause(u_short duration)
{
    u_short i;
    for (i=0;i<duration;i++) {
        shortpause(0xffff);
    }
}

void write_led(u_char value)
{
    volatile u_char * pointer;
    u_char dummy;

    pointer  = (u_char *) ( ((u_short)value) << 8);
    dummy    = *pointer;

    sbi(PORTB, 5);
    asm volatile  ("nop" ::);
    cbi(PORTB, 5);
}

int get_battery_volt(void) {

    int result;

    ADMUX |= 1<<MUX0;
    ADMUX |= 1<<MUX1;

    ADCSRA |= 1<<ADPS0;
    ADCSRA |= 1<<ADPS1;
    ADCSRA |= 1<<ADPS2;

    ADCSRA |= 1<<ADEN;
    ADCSRA |= 1<<ADSC;
    while (ADCSRA & (1<<ADSC)) ;

    result = ADCL;
    result |= ADCH << 8;

    return result;
}

static void timer3IRQ(void *arg)
{
    battery_voltage = get_battery_volt();

    if (battery_voltage < BAT_1_VOLT) {
       write_led(0x02);
    }
    else {
        if (battery_voltage < BAT_2_VOLT) {
           write_led(0x04);
        }
        else {
           write_led(0x08);
        }
    }
    // Reset the counter to non-zero value, see expl. in main routine.
    TCNT3H = 0x21;
    TCNT3L = 0x64;
    adc_done = 1;
}

int init_stdout(void)
{
    u_long baud = 57600;

    sbi( PORTD, 2);
    NutRegisterDevice(&APP_UART, 0, 0);
```

```
    freopen(APP_UART.dev_name, "r+", stdout);
    _ioctl(_fileno(stdout), UART_SETSPEED, &baud);
    return 1;
}

void EnterCritical(void)
{
    temp_sreg = SREG;
    cli();
}

void ExitCritical(void)
{
    SREG = temp_sreg;
    // an explicit sei(); is not necessary, since the I flag is
    // already set if it had been before the previous EnterCritical();
}

int main(void)
{
    int battery_voltage_volt;
    init_stdout();
    printf("Hello world!\n");
    DDRB |= 1<<DDB5;

    NutRegisterIrqHandler(&sig_OVERFLOW3, timer3IRQ, 0);
    // 16 bit timer without prescaler (clock frequency 7.3MHz)
    // -> overflows once every 0xffff*(1/7.3E6)s=9ms
    // For an overflow every 2s, prescaler should be
    // 2s/9ms = 223. The closest value is 256 (see table on page 135).
    // This gives an overflow every 2.3s. This could be adjusted by
    // setting the counter value to 0.3/2.3*0xfffff = 0x2164 after
    // every overflow, thus at the end of the timer interrupt routine
    TCCR3B |= 1<<CS32;
    ETIMSK |= 1<<TOIE3;

    EnterCritical();
    while (1) {
        while (adc_done == 0) { //adc_done is shared
            ExitCritical();
            // pause is not necessary
            EnterCritical();
        }
        battery_voltage_volt = (3300*(long)battery_voltage)/512; //battery_voltage is shared
        printf("Battery voltage is %d millivolts\n",battery_voltage_volt); //battery_voltage is shared
        adc_done = 0; //adc_done is shared
    }
    ExitCritical();

    return 0;
}
```

Solution 32 *In line 7, the value for the LEDs is put on the address bus. It is assumed that it remains there until the latch is disabled in line 11. This is not the case if between lines 7 and 11 an interrupt occurs. Thus an* `EnterCritical()` *should be inserted before line 7 and an* `ExitCritical()` *after line 11.*

```
01 void write_led(u_char value)
02 {
03    volatile u_char * pointer;
04    u_char dummy;
05
06    pointer  = (u_char *) ( ((u_short)value) << 8);
07    dummy    = *pointer;
08
09    sbi(PORTB, 5);
10    asm volatile  ("nop" ::);
11    cbi(PORTB, 5);
12 }
```

Chapter 4 – Programming with Threads

Solution 33 – Sample Code

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

THREAD(my_thread, arg)
{
    for (;;) {
        btn_led_clear(LED0);
        btn_led_set(LED1);
        NutThreadYield(); // second yield to add (then both LEDs are on)
    }
}

int main(void)
{
    // hardware init
    btn_hardware_init();
    btn_led_init(0);


    NutThreadCreate("my_thread",my_thread,0,192);
    for (;;) {
        btn_led_clear(LED1);
        btn_led_set(LED0);
        NutThreadYield(); // first yield to add (then only red LED is on)
    }
    return 0;
}
```

Solution 34 *The output is:*

```
my_thread is alive
main is alive
main is alive
my_thread is alive
my_thread is alive
main is alive
main is alive
my_thread is alive
my_thread is alive
...
```

Sample Code

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

int init_stdout(void)
{
    u_long baud = 57600;

    sbi( PORTD, 2);
    NutRegisterDevice(&APP_UART, 0, 0);
    freopen(APP_UART.dev_name, "r+", stdout);
    _ioctl(_fileno(stdout), UART_SETSPEED, &baud);
    return 1;
}

THREAD(my_thread, arg)
{
    for (;;) {
        printf("my_thread is alive\n");
        NutSleep(1000);
    }
}

int main(void)
{
    // hardware init
    btn_hardware_init();
    btn_led_init(0);

    init_stdout();
```

```
    NutThreadCreate("my_thread",my_thread,0,192);
    for (;;) {
        printf("main is alive\n");
        NutSleep(1000);
    }
    return 0;
}
```

Solution 35 *The output is:*

```
my_thread is alive
main is alive
my_thread is alive
main is alive
my_thread is alive
main is alive
my_thread is alive
main is alive
my_thread is alive
main is alive
...
```

You would expect for both exercises the same output, that of exercise 33. The reason for the deviating behavior is that the mechanism that wakes up threads after a sleep is buggy:

1. *NutSleep puts the threads in the sleep queue, AFTER all threads of higher or equal priority.*
2. *When threads are woken up, the threads are put in the run queue, BEFORE all threads of lower or equal priority.*

Thus if two threads have the same priority and are woken up at the same time, their order gets reversed. The probelm does not occur when the threads have different priorities or are woken up at different times.

Both threads are woken up at the same time, because `NutSleep` *has a granularity of approximately 64 milliseconds. We have seen in the previous chapter that a printf (with such a short string) takes about 1 millisecond, thus both threads go to sleep and are woken up in the same 64 milliseconds time slot.*

Sample Code

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

int init_stdout(void)
{
    u_long baud = 57600;

    sbi( PORTD, 2);
    NutRegisterDevice(&APP_UART, 0, 0);
    freopen(APP_UART.dev_name, "r+", stdout);
    _ioctl(_fileno(stdout), UART_SETSPEED, &baud);
    return 1;
}

THREAD(my_thread, arg)
{
    NutThreadSetPriority(20);
    for (;;) {
        printf("my_thread is alive\n");
        NutSleep(1000);
    }
}

int main(void)
{
    // hardware init
    btn_hardware_init();
    btn_led_init(0);
```

```
    init_stdout();

    NutThreadCreate("my_thread",my_thread,0,192);
    for (;;) {
        printf("main is alive\n");
        NutSleep(1000);
    }
    return 0;
}
```

Solution 36 *The ouptut received is:*

```
MAIN IS ALIVE
MAIN IS ALis alive
my_thread IVE
MAIN IS ALIVE
MAIN IS ALIVE
MAIN IS is alive
my_thread ALIVE
MAIN IS ALIVis alive
my_thread E
MAIN IS ALIVE
MAIN IS ALIVE
MAIN IS ALis alive
my_thread IVE
MAIN IS ALIVE
MAIN IS ALIVE
MAIN IS is alive
...
```

Thus both threads run and print to the terminal in an uncoordinated fashion.

Why do both threads run, even though there is no `NutThreadYield()` *or* `NutSleep()` *in the code?*

The reason is that `printf()` *copies the string in a buffer, which is later put on the UART by an interrupt service routine (UART TX empty interrupt). If the buffer is filled quickly, it becomes full. When the buffer is full,* `printf()` *implicitely does a* `NutThreadYield`. *Therefore you should be very careful about protecting data that is shared by multiple threads when using* `printf()`.

Sample Code

```
#include <stdio.h>
#include <dev/usartavr.h>
#include <sys/thread.h>
#include <sys/timer.h>
#include <sys/tracer.h>
#include <hardware/btn-hardware.h>
#include <sys/osdebug.h>

#include <terminal/btn-terminal.h>

int init_stdout(void)
{
    u_long baud = 57600;

    sbi( PORTD, 2);
    NutRegisterDevice(&APP_UART, 0, 0);
    freopen(APP_UART.dev_name, "r+", stdout);
    _ioctl(_fileno(stdout), UART_SETSPEED, &baud);
    return 1;
}

THREAD(my_thread, arg)
{
    int sleeptime = *(int*)arg;
    for (;;) {
        printf("%s is alive, will sleep for %d seconds\n",runningThread->td_name,sleeptime);
        NutSleep((u_long)1000*sleeptime);
    }
}

void create(u_char * arg)
{
    static int sleeptime = 1;
    char name[20];
    int val;
    val = sscanf(arg,"%s",name);
```

```
    if (val==1) {
        printf("Create a thread with name %s and sleeptime %d\n",name,sleeptime);
        if (0 == NutThreadCreate(name,my_thread,&sleeptime,292)) {
            printf("FAILED!\n");
        }
        else {
            printf("SUCCESFUL!\n");
            sleeptime++;
        }
    }
}

int main(void)
{
    // hardware init
    btn_hardware_init();

    init_stdout();

    btn_terminal_init(stdout, "[bt-cmd@btnode]£");
    btn_terminal_register_cmd("create",create);
    btn_terminal_run(BTN_TERMINAL_NOFORK, 0);
    return 0;
}
```

Solution 37 – Sample Code

```
#include <stdio.h>
#include <dev/usartavr.h>
#include <sys/thread.h>
#include <sys/timer.h>
#include <sys/tracer.h>
#include <hardware/btn-hardware.h>
#include <sys/osdebug.h>

#include <terminal/btn-terminal.h>

int init_stdout(void)
{
    u_long baud = 57600;

    sbi( PORTD, 2);
    NutRegisterDevice(&APP_UART, 0, 0);
    freopen(APP_UART.dev_name, "r+", stdout);
    _ioctl(_fileno(stdout), UART_SETSPEED, &baud);
    return 1;
}

THREAD(my_thread, arg)
{
    for (;;) {
        printf("%s is alive\n",runningThread->td_name);
        NutSleep(1000);
    }
}

void create(u_char * arg)
{
    char name[20];
    int val;
    // strange behavior here: typing "create name" does not work, but "create name " does!?
    val = sscanf(arg,"%s",name);
    if (val==1) {
        printf("Create a thread with name %s\n",name);
        if (0 == NutThreadCreate(name,my_thread,0,292)) {
            printf("FAILED!\n");
        }
        else {
            printf("SUCCESFUL!\n");
        }
    }
}

int main(void)
{
    // hardware init
    btn_hardware_init();

    init_stdout();
```

```
    btn_terminal_init(stdout, "[bt-cmd@btnode]$");
    btn_terminal_register_cmd("create",create);
    btn_terminal_run(BTN_TERMINAL_NOFORK, 0);
    return 0;
}
```

Solution 38 – Sample Code

```
#include <stdio.h>
#include <dev/usartavr.h>
#include <sys/thread.h>
#include <sys/timer.h>
#include <sys/tracer.h>
#include <hardware/btn-hardware.h>
#include <sys/osdebug.h>

#include <terminal/btn-terminal.h>

int init_stdout(void)
{
    u_long baud = 57600;

    sbi( PORTD, 2);
    NutRegisterDevice(&APP_UART, 0, 0);
    freopen(APP_UART.dev_name, "r+", stdout);
    _ioctl(_fileno(stdout), UART_SETSPEED, &baud);
    return 1;
}

THREAD(my_thread, arg)
{
    int sleeptime = *(int*)arg;
    for (;;) {
        printf("%s is alive, will sleep for %d seconds\n",runningThread->td_name,sleeptime);
        NutSleep((u_long)1000*sleeptime);
    }
}

void create(u_char * arg)
{
    static int sleeptime = 1;
    char name[20];
    int val;
    val = sscanf(arg,"%s",name);
    if (val==1) {
        printf("Create a thread with name %s and sleeptime %d\n",name,sleeptime);
        if (0 == NutThreadCreate(name,my_thread,&sleeptime,292)) {
            printf("FAILED!\n");
        }
        else {
            printf("SUCCESFUL!\n");
            sleeptime++;
        }
    }
}

int main(void)
{
    // hardware init
    btn_hardware_init();

    init_stdout();

    btn_terminal_init(stdout, "[bt-cmd@btnode]$");
    btn_terminal_register_cmd("create",create);
    btn_terminal_run(BTN_TERMINAL_NOFORK, 0);
    return 0;
}
```

Solution 39 – Sample Code

```
#include <stdio.h>
#include <dev/usartavr.h>
#include <sys/thread.h>
#include <sys/timer.h>
```

```
#include <sys/tracer.h>
#include <hardware/btn-hardware.h>
#include <sys/osdebug.h>
#include <terminal/nut-cmds.h>

#include <terminal/btn-terminal.h>

int init_stdout(void)
{
    u_long baud = 57600;

    sbi( PORTD, 2);
    NutRegisterDevice(&APP_UART, 0, 0);
    freopen(APP_UART.dev_name, "r+", stdout);
    _ioctl(_fileno(stdout), UART_SETSPEED, &baud);
    return 1;
}

THREAD(my_thread, arg)
{
// IF THE variable dummy_str is initialized AND USED, more stack is used
//    char  dummy_str[20];
    int sleeptime = *(int*)arg;
//    sprintf(dummy_str,"hello world\n");
    for (;;) {
        printf("%s is alive, will sleep for %d seconds\n",runningThread->td_name,sleeptime);
        NutSleep(1000*sleeptime);
    }
}

void create(u_char * arg)
{
    static int sleeptime = 1;
    int stacksize;
    char name[20];
    int val;
    val = sscanf(arg,"%s%d",name,&stacksize);
    if (val==2) {
        printf("Create a thread with name %s and sleeptime %d\n",name,sleeptime);
        if (0 == NutThreadCreate(name,my_thread,&sleeptime,stacksize)) {
            printf("FAILED!\n");
        }
        else {
            printf("SUCCESFUL!\n");
            sleeptime++;
        }
    }
}

int main(void)
{
    // hardware init
    btn_hardware_init();

    init_stdout();

    btn_terminal_init(stdout, "[bt-cmd@btnode]$");
    btn_terminal_register_cmd("create",create);
    nut_cmds_register_cmds();
    btn_terminal_run(BTN_TERMINAL_NOFORK, 0);
    return 0;
}
```

Solution 40 – Sample Code

```
#include <stdio.h>
#include <dev/usartavr.h>
#include <sys/thread.h>
#include <sys/timer.h>
#include <sys/tracer.h>
#include <sys/event.h>
#include <hardware/btn-hardware.h>
#include <led/btn-led.h>

HANDLE event;

int init_stdout(void)
{
    u_long baud = 57600;
```

```
    sbi( PORTD, 2);
    NutRegisterDevice(&APP_UART, 0, 0);
    freopen(APP_UART.dev_name, "r+", stdout);
    _ioctl(_fileno(stdout), UART_SETSPEED, &baud);
    return 1;
}

THREAD(my_thread, arg)
{
    int count = 0;
    for (;;) {
        NutEventWait(&event, NUT_WAIT_INFINITE);
        count++;
        printf("myThread has received event no. %d\n",count);
    }
}

int main(void)
{
    init_stdout();
    NutEventPost(&event);
    printf("main posts an event\n");
    NutEventPost(&event);
    printf("main posts an event\n");
    NutThreadCreate("myThread",my_thread,0,192);
    printf("main enters endless loop\n");
    for (;;)
        NutThreadYield();
    return 0;
}
```

Solution 41 – Sample Code

```
#include <stdio.h>
#include <dev/usartavr.h>
#include <sys/thread.h>
#include <sys/timer.h>
#include <sys/tracer.h>
#include <sys/event.h>
#include <hardware/btn-hardware.h>
#include <led/btn-led.h>
#include <string.h>

HANDLE event;

void shortpause(u_short duration)
{
    u_short i;
    for (i=0;i<duration;i++) {
    }
}

void pause(u_short duration)
{
    u_short i;
    for (i=0;i<duration;i++) {
        shortpause(0xffff);
    }
}

int init_stdout(void)
{
    u_long baud = 57600;

    sbi( PORTD, 2);
    NutRegisterDevice(&APP_UART, 0, 0);
    freopen(APP_UART.dev_name, "r+", stdout);
    _ioctl(_fileno(stdout), UART_SETSPEED, &baud);
    return 1;
}

THREAD(my_thread, arg)
{
    int count = 0;
    if (strncmp("A",runningThread->td_name,1)==0) {
//        NutThreadSetPriority(10);
    }
    for (;;) {
```

```
        NutEventWait(&event, NUT_WAIT_INFINITE);
        count++;
        printf("Thread %s has received event no. %d\n",runningThread->td_name,count);
        pause(12);
    }
}

int main(void)
{
    int count = 7;
    init_stdout();
    NutThreadCreate("A",my_thread,0,192);
    NutThreadCreate("B",my_thread,0,192);
    while (count > 0) {
        NutEventPost(&event);
//        printf("main posts an event\n");
        count--;
    }
    printf("main enters endless loop\n");
    pause(12);
    for (;;)
        NutThreadYield();
    return 0;
}
```

Solution 42 *The ouptut received is:*

```
main posts an event
main posts an event
main enters endless loop
myThread has received event no. 1
```

`myThread` *receives only 1 event, even though main had posted an event twice before starting* `myThread`*. Thus we see that the event-queue remembers that an event was posted when no one waits for it, but it does not remember how many events have been posted. The implementation of the event-queues is so that the event-queue enters the 'signaled' state when an event is posted and nobody waits for it, but there is no counter of such events.*

Sample Code

```
#include <stdio.h>
#include <dev/usartavr.h>
#include <sys/thread.h>
#include <sys/timer.h>
#include <sys/tracer.h>
#include <sys/event.h>
#include <hardware/btn-hardware.h>
#include <led/btn-led.h>

HANDLE event;

int init_stdout(void)
{
    u_long baud = 57600;

    sbi( PORTD, 2);
    NutRegisterDevice(&APP_UART, 0, 0);
    freopen(APP_UART.dev_name, "r+", stdout);
    _ioctl(_fileno(stdout), UART_SETSPEED, &baud);
    return 1;
}

THREAD(my_thread, arg)
{
    int count = 0;
    for (;;) {
        NutEventWait(&event, NUT_WAIT_INFINITE);
        count++;
        printf("myThread has received event no. %d\n",count);
    }
}

int main(void)
{
    init_stdout();
    NutEventPost(&event);
```

```
    printf("main posts an event\n");
    NutEventPost(&event);
    printf("main posts an event\n");
    NutThreadCreate("myThread",my_thread,0,192);
    printf("main enters endless loop\n");
    for (;;)
        NutThreadYield();
    return 0;
}
```

Solution 43 *The ouptut received is:*

```
hread A has received event no. 1
Thread B has received event no. 1
Thread A has received event no. 2
Thread B has received event no. 2
Thread A has received event no. 3
Thread B has received event no. 3
main enters endless loop
Thread A has received event no. 4
```

Thus we see that every event is received ONLY by one thread, even though two threads are waiting. Since both threads have the same priority, they are served in turns.

When line 80 (`NutThreadSetPriority`*) is uncommented, then the output is:*

```
Thread A has received event no. 1
Thread A has received event no. 2
Thread A has received event no. 3
Thread A has received event no. 4
Thread A has received event no. 5
Thread A has received event no. 6
Thread A has received event no. 7
main enters endless loop
```

Thus we see that thread priorities DO play a role.

Note that the calls to the pause function are a quick hack that leads to a readable terminal output (remember that `printf` *implicitely does a* `NutThreadYield`*, see exercise 34).*

Sample Code

```
#include <stdio.h>
#include <dev/usartavr.h>
#include <sys/thread.h>
#include <sys/timer.h>
#include <sys/tracer.h>
#include <sys/event.h>
#include <hardware/btn-hardware.h>
#include <led/btn-led.h>
#include <string.h>

HANDLE event;

void shortpause(u_short duration)
{
    u_short i;
    for (i=0;i<duration;i++) {
    }
}

void pause(u_short duration)
{
    u_short i;
    for (i=0;i<duration;i++) {
        shortpause(0xffff);
    }
}

int init_stdout(void)
{
    u_long baud = 57600;

    sbi( PORTD, 2);
    NutRegisterDevice(&APP_UART, 0, 0);
    freopen(APP_UART.dev_name, "r+", stdout);
```

```
    _ioctl(_fileno(stdout), UART_SETSPEED, &baud);
    return 1;
}

THREAD(my_thread, arg)
{
    int count = 0;
    if (strncmp("A",runningThread->td_name,1)==0) {
//        NutThreadSetPriority(10);
    }
    for (;;) {
        NutEventWait(&event, NUT_WAIT_INFINITE);
        count++;
        printf("Thread %s has received event no. %d\n",runningThread->td_name,count);
        pause(12);
    }
}

int main(void)
{
    int count = 7;
    init_stdout();
    NutThreadCreate("A",my_thread,0,192);
    NutThreadCreate("B",my_thread,0,192);
    while (count > 0) {
        NutEventPost(&event);
//        printf("main posts an event\n");
        count--;
    }
    printf("main enters endless loop\n");
    pause(12);
    for (;;)
        NutThreadYield();
    return 0;
}
```

Chapter 5 – Embedded Debugging

Solution 44 *Not yet.*

Solution 45 *Not yet.*

Solution 46 *Not yet.*

Solution 47 *Not yet.*

Solution 48 *Not yet.*

Solution 49 – Sample Code

```
#include <io.h>
#include <stdio.h>
#include <dev/usartavr.h>
#include <sys/thread.h>
#include <sys/timer.h>
#include <sys/tracer.h>
#include <hardware/btn-hardware.h>
#include <led/btn-led.h>
#include <terminal/btn-terminal.h>

int init_stdout(void)
{
    u_long baud = 57600;
    sbi( PORTD, 2);
    NutRegisterDevice(&APP_UART, 0, 0);
    freopen(APP_UART.dev_name, "r+", stdout);
    _ioctl(_fileno(stdout), UART_SETSPEED, &baud);
    return 1;
}
```

```
THREAD(ledthread, arg)
{
    for (;;) {
        btn_led_clear(LED1);
        btn_led_set(LED0);
            NutSleep(500);
            btn_led_clear(LED0);
        btn_led_set(LED1);
        NutSleep(500);
    }
}

int main(void)
{
    // hardware init
    btn_hardware_init();
    btn_led_init(0);
    init_stdout();
    NutThreadCreate("LedThr",ledthread,0,192);
    btn_terminal_init(stdout, "[bt-cmd@btnode]$");
    btn_terminal_register_cmd("trace",NutTraceTerminal);
    btn_terminal_run(BTN_TERMINAL_NOFORK, 0);

    return 0;
}
```

Solution 50 *Not yet.*

Solution 51 *Not yet.*

Solution 52 *Not yet.*

Chapter 6 – Communication using Bluetooth

Solution 53 *The Bluetooth device address can be found on a label attached to the BTnode (e.g. 00:04:3F:00:00:4B) and consists of 6 bytes.*

Solution 54 `struct bt_hci_pkt_cmd pkt;   //`

```
     pkt.type=0x01;              //HCI command packet
     pkt.payload[0]=0x01;        //first byte of OpCode
     pkt.payload[1]=0x04;        //second byte of OpCode
     pkt.payload[2]=0x05;        //total length of parameters
     pkt.payload[3]=0x33;        //GIAC
     pkt.payload[4]=0x8b;        //GIAC
     pkt.payload[5]=0x9e;        //GIAC
     pkt.payload[6]=0x05;        //inquiry length 5*1.28s = 6.4 seconds
     pkt.payload[7]=0x05;        //maximum number of devices
```

Solution 55 – Sample Code

```
// send an inquiry command ...
// and use the Btstack thread to receice the answers
#include <sys/tracer.h>
#include <hardware/btn-hardware.h>
#include <terminal/btn-terminal.h>
#include <stdio.h> // freopen
#include <dev/usartavr.h> // NutRegisterDevice, APP_UART, UART_SETSPEED
#include <led/btn-led.h> // needed! (together with btn_led_init(0);), maybe a bug ???
#include <bt/bt_hci_dispatch.h> // for the setWaitQueue command
#include <sys/event.h>  // for NutEventWait and NUT_WAIT_INFINITE
#include <bt/bt_hci_cmds.h> // for sending hci commands
```

```
#define HCI_COMMAND_DATA_PACKET        0x01
#define HCI_OGF_LINK_CONTROL        0x01
#define HCI_OCF_LC_INQUIRY                0x01
#define BT_HCI_HANDLE_INVALID 0xFFFF

struct btstack* stack;

void init_stdout(void) {
u_long baud = 57600;
btn_hardware_init();
NutRegisterDevice(&APP_UART, 0, 0);
freopen(APP_UART.dev_name, "r+", stdout);
_ioctl(_fileno(stdout), UART_SETSPEED, &baud);
}

//print a single bluetooth device address to the terminal
void print_bt_addr(bt_addr_t addr) {
      printf("%.2x:%.2x:%.2x:%.2x:%.2x:%.2x", addr[5],addr[4],addr[3],addr[2],addr[1],addr[0]);
}

//print the number of found devices and their addresses
void print_inq_result(struct bt_hci_cmd_response wcmd) {
    int i;
    printf("Devices: %li\n", wcmd.response);
    if (wcmd.response<0) {
      wcmd.response=0;
    } else {
      printf("Device         bt_addr \n");
    }
    for (i=0; i<wcmd.response; i++) {
        printf("[%d]: ", i);
        print_bt_addr(   (((struct bt_hci_inquiry_result*)(wcmd.ptr)) + i)->bdaddr   );
        printf(" \n");
    }
}

void inquiry (char* arg){

//**************** packet construction******************************

struct bt_hci_pkt_cmd pkt;

pkt.type=HCI_COMMAND_DATA_PACKET;
pkt.payload[0]=HCI_OCF_LC_INQUIRY;
pkt.payload[1]=HCI_OGF_LINK_CONTROL<<2;
//cmd length = 5 bytes
pkt.payload[2]=0x05;
//General Inquiry Access Code (GIAC)
pkt.payload[3]=0x33;
pkt.payload[4]=0x8b;
pkt.payload[5]=0x9e;
// waiting time for the inquiry to complete
// ----> 5 * 1.28 s = 6.4 s
pkt.payload[6]=0x05;
// maximum number of responding devices
pkt.payload[7]=0x05;

//*************** prepare and register cmd_response-structure********

struct bt_hci_cmd_response wcmd;

//array for the storage of the answers of max. 10 devices
struct bt_hci_inquiry_result inquiry_result[10];

//initialize the cmd_response-structure
wcmd.ogfocf= ((HCI_OCF_LC_INQUIRY<<8)|(HCI_OGF_LINK_CONTROL<<2));
wcmd.cmd_handle= BT_HCI_HANDLE_INVALID;
wcmd.response=0;
wcmd.ptr= &inquiry_result;
wcmd.block=0;

//register the wcmd in the WaitQueue of the running stack
_bt_hci_setWaitQueue(stack,&wcmd);

//*************** send packet, wait and readout the results********

_bt_hci_send_pkt(stack,(u_char*)&pkt);
printf("Starting inquiry .....\n");
NutEventWait(&(wcmd.block),NUT_WAIT_INFINITE);
printf("Inquiry done! \n");
```

```
print_inq_result(wcmd);
}

int main(void) {
btn_hardware_init();
btn_led_init(0);
init_stdout();

// bluetooth module on (takes a while)
btn_hardware_bt_on();

// Start the stack and let the initialization begin
stack = bt_hci_init(&BT_UART);

btn_terminal_init(stdout, "[es200X]$");
btn_terminal_register_cmd("inquiry", inquiry);
btn_terminal_register_cmd("trace",NutTraceTerminal);
btn_terminal_run(BTN_TERMINAL_NOFORK, 0);
return 0;
}
```

Solution 56 *You can clearly identify a CommandStatusEvent, several InquiryResultEvents as well as a final InquiryCompleteEvent.*

Solution 57 *Not yet.*

Solution 58 – Sample Code

```
//       sending a string message within an acl-packet
/*
 * Copyright (C) 2000-2006 by ETH Zurich
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the copyright holders nor the names of
 *    contributors may be used to endorse or promote products derived
 *    from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY ETH ZURICH AND CONTRIBUTORS
 * ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL ETH ZURICH
 * OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 * BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS
 * OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED
 * AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
 * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF
 * THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * For additional information see http://www.btnode.ethz.ch/
 *
 * $Id: ch-6-ex-58.c,v 1.3 2006/05/19 15:02:08 cmoser79 Exp $
 *
 */

/*!
 * $Log: ch-6-ex-58.c,v $
 * Revision 1.3  2006/05/19 15:02:08  cmoser79
 * final changes to chapter 6 (Clemens)
 *
 * Revision 1.43  2006/04/06 08:52:10  kevmarti
 * removed 'log_cmds_register_cmds()' function (registering is now done in 'log_cmds_init()')
 *
 * Revision 1.42  2006/04/05 12:56:21  kevmarti
 * adjusted call to 'log_init()'
 *
```

```
 * Revision 1.41  2006/04/05 12:47:37  kevmarti
 * Added call to log_cmds_init()
 *
 * Revision 1.40  2006/04/05 10:44:22  kevmarti
 * terminal cmds for logging moved from 'debug/logging.c' to 'terminal/log-cmds.c'
 *
 * Revision 1.39  2006/04/05 10:05:48  beutel
 * *** empty log message ***
 *
 * Revision 1.38  2006/04/05 05:29:36  dyerm
 * fixed reading of bt version and features for the fancy header
 *
 * Revision 1.37  2006/03/29 01:15:00  olereinhardt
 *
 * Changed signedness of strings in order to compile with avr-gcc 4.0.2
 *
 * Revision 1.36  2006/03/24 14:44:50  dyerm
 * removed obsolete bt_acl_com
 *
 * Revision 1.35  2006/03/23 17:13:57  beutel
 * added version, features and name to bt-cmd
 *
 * Revision 1.34  2006/03/23 17:12:39  beutel
 * added version, features and name to bt-cmd
 *
 * Revision 1.33  2006/03/23 07:22:24  dyerm
 * Merged changes from multihop_merge branch. See individual changes on
 * multihop_merge branch. See Changelog for summary of changes.
 *
 */

/**
 * \example bt-cmd/bt-cmd.c
 *
 * \date 2004/06/18
 *
 * \author Martin Hinz <btnode@hinz.ch>
 * \author Jan Beutel <j.beutel@ieee.org>
 *
 * Example application to show the use of the bt stack and the simple but
 * powerful terminal interface.
 */

#include <stdio.h>
#include <dev/usartavr.h>
#include <sys/heap.h>
#include <sys/timer.h>

#include <hardware/btn-hardware.h>

#include <bt/bt_hci_cmds.h>

#include <terminal/btn-terminal.h>
#include <terminal/btn-cmds.h>
#include <terminal/bt-cmds.h>
#include <terminal/nut-cmds.h>
#include <terminal/log-cmds.h>

#include <led/btn-led.h>
#include <debug/logging.h>

#include "program_version.h"
#define CVS_VERSION "$Id: ch-6-ex-58.c,v 1.3 2006/05/19 15:02:08 cmoser79 Exp $"

struct btstack* stack;
extern u_char _bt_hci_debug_uart;

//sending a text message on a certain channel with a certain connection handle
void transmit (char* arg){

        int handle, channel,i;
        u_char message[20];
        //set empty message
        for (i=0;i<=19;i++)
                message[i]=' ';

        //dont separate single words in "message" with spaces,
        //BUT: leave space after end of "message",
        //(otherwise bluetooth module may be re-booted)
        sscanf(arg, "%u%u%s", &handle, &channel, message);
```

```
        //define a packet:  5 bytes for hci-packet-header,
        //4 bytes for L2CAP-packet header and  20 bytes payload
        u_char hci_acl_pkt[29];

        //the following bytes are set by the bt_hci_send_acl_pkt-function:
        //hci_acl_pkt[0] type
        //hci_acl_pkt[1] con handle
        //hci_acl_pkt[2] con handle + flags
        //hci_acl_pkt[3] flags + data length
        //hci_acl_pkt[4] data length
        hci_acl_pkt[5]=(u_char)(20 & 0xFF);
        hci_acl_pkt[6]=(u_char)((20>>8) & 0xFF);
        hci_acl_pkt[7]=(u_char)(channel & 0xFF);
        hci_acl_pkt[8]=(u_char)((channel>>8) & 0xFF);

        //attach the string message
        for(i=0;i<=18;i++)
                hci_acl_pkt[9+i]=message[i];

        //... and send the packet
        bt_hci_send_acl_pkt(stack,handle,2,0,24,(struct bt_hci_pkt_acl*)(hci_acl_pkt));

        printf("Message (%s) sent with handle %d, channel %d \n",message,handle,channel);
}


struct bt_hci_pkt_acl* receive(void *arg, struct bt_hci_pkt_acl *pkt, bt_hci_con_handle_t con_handle, u_char pb_flag, u_char bc_flag, u_short
{
        u_char* l2cap_hdr = pkt->payload;
        u_char* l2cap_data;
        u_short chan_id;

        chan_id = l2cap_hdr[2] | (l2cap_hdr[3] << 8);

        l2cap_data = &l2cap_hdr[4];

        printf("message received on channel %d: %s\n", chan_id, l2cap_data);
        return pkt;
}


/**
 * main function that initializes the hardware, led, terminal, bluetooth
 * and acl communication stack and registers some predefined commands.
 * Use tab-tab to see the registered commands once the program is running.
 */
int main(void)
{
          u_char acl_pkt[120];

    // serial baud rate
    u_long baud = 57600;
    u_long cpu_crystal;
    u_long nut_tick_freq;

    // hardware init
    btn_hardware_init();
    btn_led_init(1);

    // init app uart
    NutRegisterDevice(&APP_UART, 0, 0);
    freopen(APP_UART.dev_name, "r+", stdout);
    _ioctl(_fileno(stdout), UART_SETSPEED, &baud);

    // logging
    log_init();

    // hello world!
    printf("\n# ------------------------------------------------------");
    printf("\n# Welcome to BTnut (c) 2006 ETH Zurich\n");
    printf("# bt-cmd program version: %s\n", PROGRAM_VERSION);
    printf("# %s\n", CVS_VERSION);
    cpu_crystal = NutGetCpuClock();
    nut_tick_freq = NutGetTickClock();
    printf("# running @ %u.%04u MHz, NutFreq=%ul Hz\n",
           (int) (cpu_crystal / 1000000UL), (int) ((cpu_crystal - (cpu_crystal / 1000000UL) * 1000000UL) / 100),
        (int) nut_tick_freq);
    printf("# -----------------------------------------------------");
    printf("\nbooting Bluetooth module...\n");

    // bluetooth module on (takes a while)
```

```
    btn_hardware_bt_on();

    // verbose debug of all hci information
    //_bt_hci_debug_uart = 1;

    // Start the stack and let the initialization begin
    stack = bt_hci_init(&BT_UART);

    bt_hci_write_default_link_policy_settings(stack, BT_HCI_SYNC,
                                          BT_HCI_LINK_POLICY_ROLE_SWITCH |
                                          BT_HCI_LINK_POLICY_HOLD_MODE |
                                          BT_HCI_LINK_POLICY_SNIFF_MODE |
                                          BT_HCI_LINK_POLICY_PARK_STATE );
    bt_addr_t addr;
    struct bt_hci_local_version_result version;
    u_char features[8];
    u_char _bt_cmds_name[30];

    bt_hci_read_bt_addr(stack, BT_HCI_SYNC, addr);
    printf("Bluetooth MAC address: %.2x:%.2x:%.2x:%.2x:%.2x:%.2x\n", addr[5], addr[4], addr[3], addr[2], addr[1], addr[0]);
    bt_hci_read_local_version_information(stack, BT_HCI_SYNC, &version);
    printf("HCI version: %X %.4X %X %.4X %.4X\n", version.hciversion,
           version.hcirevision, version.lmpversion, version.manufacturername, version.lmpsubversion);
    bt_hci_read_local_supported_features(stack, BT_HCI_SYNC, features);
    printf("LMP features: %.2X %.2X %.2X %.2X %.2X %.2X %.2X %.2X\n",
           features[0], features[1], features[2], features[3], features[4], features[5], features[6], features[7]);
    bt_hci_read_local_name(stack, BT_HCI_SYNC, _bt_cmds_name, sizeof(_bt_cmds_name));
    printf("Local name: '%s'\n", _bt_cmds_name);

    // give hint
    printf("hit tab twice for a list of commands\n\r");

    // terminal init
    char prompt[20];
    sprintf(prompt, "[bt-cmd@%.2x:%.2x]$", addr[1], addr[0]);
    btn_terminal_init(stdout, prompt);
    bt_cmds_init(stack);
    bt_cmds_register_cmds();
    btn_cmds_register_cmds();
    nut_cmds_register_cmds();
    log_cmds_init(stdout);

    bt_hci_register_acl_cb(stack, receive, (struct bt_hci_pkt_acl*)acl_pkt, NULL);

    btn_terminal_register_cmd("transmit",transmit);
    // terminal mode
    btn_terminal_run(BTN_TERMINAL_NOFORK, 0);

    return 0;
}
```

Solution 59 *Not yet.*

Solution 60 *Not yet.*

Bibliography

[1] Atmel. *Atmel ATmega128L - 8-Bit AVR Microcontroller with 128k in-System programmable Flash*, November 2004.

[2] Bluetooth Special Interest Group. *Specification of the Bluetooth System v.1.2*, November 2003.

[3] J.L. Hennessy and D.A. Patterson. *Computer organization and design: The hardware/software interface.* Morgan Kaufmann Publishers, San Francisco, CA, 2nd edition, 1997.