Didactic Embedded Platform and Software Tools for Developing Real Time Operating System

Adam Kaliszan
Poznan University of Technology
Chair of Communication and Computer Networks
Email: adam.kaliszan@gmail.com
http://www.adam.kaliszan.yum.pl

Mariusz Głąbowski
Poznan University of Technology
Chair of Communication and Computer Networks
Email: mariusz.glabowski@et.put.poznan.pl

***Abstract*—This paper presents a new didactic platform that is capable of running an embedded Real Rime Operating System. The proposed platform consists of hardware, firmware and software tools. The project of the hardware part is distributed according to GPLv2 license. The firmware of the platform is based on FreeRtos distributed according to the modified GPL license, ported by the authors on the microcontrollers not originally supported, i.e., Atmega128 and Atmega168. All the software tools work on the Linux operating system and are free of charge; most of them have open source code. The main aim of the proposed platform is to familiarize students with the basics of embedded RTOS.**

***Keywords*-Embedded systems, Real Time Operating System, Multitasking, Interprocess communication**

I. Introduction

An important addendum of an embedded and operating systems theory course is practice. The practical part of the course is often conducted with the help of one of the existing operating systems, usually Linux or Windows. Linux has advantages, such as its versatility ranging from small embedded devices to powerful supercomputers. Owing to the Linux open source code, there are many written kernel modules [1] supporting new devices, what ensures such a great versatility of the system and it is applicable in many embedded systems. Microsoft offers different versions of its own operating system, ranging from Windows CE or Windows Mobile that are working on mobile phones, PDA devices and car navigation, to Windows Server. On account of Microsoft .NET framework, it is possible to write software in a very easy way. It should be noted that the software produced by Microsoft is not free. The fact that its code is closed complicates porting the operating system to new, not particularly common, hardware devices. Its application is limited to few basic CPU architectures.

Irrespective of a chosen operating system, the practical part is often limited to learning the basis of operating systems, i.e., learning Linux fundamental commands such as creating and removing files or directories, changing file attributes and launching programs. Such laboratory classes do not introduce the subject of embedded systems, as well as they have no connection to the operating system theory, since the laboratories do not cover topics like multitasking, interprocess communication and its synchronization or operations on file system. The mentioned difficulties are caused by the absence of a proper platform with a simplified programming interface that is capable of building (compiling) in a short amount of time. In the Linux case, the complication results mostly from a required compatibility with various standards, e.g., Linux is compatible with posix and sysV standards [2]. In order to provide the compatibility with each of these standards, separate interfaces have been introduced. Consequently, it takes a lot of time to get familiar with the whole programming interface and, finally, students getting prepared to their laboratories are generally focused on studying the documentation instead of understanding the sense of presented mechanisms of the operating systems. Additionally, the build time of the embedded Linux requires about one hour, while laboratory classes last usually 90 minutes. In view of the above-mentioned difficulties the authors felt encouraged to elaborate a new didactic platform, including hardware, firmware and software tools. In the presented system the handling of mechanisms such as files, multi-tasking, interprocess communication and process synchronization, have been simplified. The software code, worked out to meet the demand of the new platform, is open, distributed according to GPLv2 [3] license and it allows students to get familiar with particular mechanisms of the operating systems.

The remaining part of the paper is organized as follows. Section II presents the hardware of the proposed platform. In Section III the software architecture is described. In Section IV an exemplary exercise conducted with the help of the proposed platform is presented. Section V concludes the paper.

II. Hardware

The hardware part was designed with the help of a freeware version of Eagle [4] CAD software. The dimension of PCB board was limited to 10 by 8 centimeters (i.e., the maximum dimensions of PCB board allowed by freeware version of Eagle CAD software). The complete

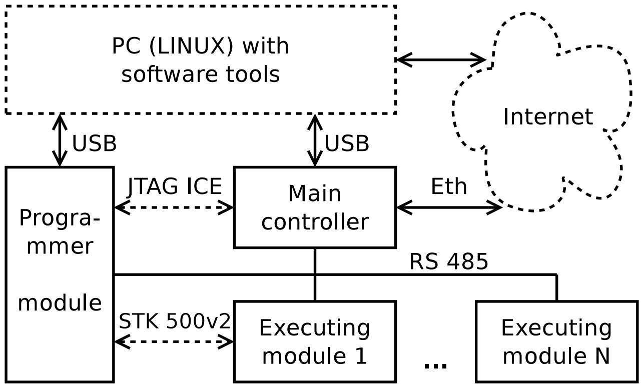


Figure 1. Modular schematic of the platform's hardware

project of the hardware is available at svn repository http://akme.yum.pl/eagle/ssw, where the login and the password is "student". In order to download the project, the following command must be executed in the shell prompt: *svn co http://akme.yum.pl/eagle/ssw*. The limited dimension of the board allows students to modify the project using freeware version of Eagle CAD. The hardware was designed in a user friendly manner: it uses a common interface and does not need any external power supply. The platform is connected to a PC by via USB, since RS 232 is not very common in modern personal computers. There is a place on the main controller for power converter. It allows the platform to work as a standalone device that does not require power supply from USB. The hardware project bases on AVR microcontrollers [5], [6]. This reduces instruction set computing CPU architecture is preferred by students because of its simplicity, freeware C compiler (avrgcc) and high performance in comparison with other 8-bit microcontroller architectures.

Figure 1 shows a schematic diagram of the platform. The system is distributed and consists of the main controller and executing modules. Both modules are being programmed, using universal programmer designed for the platform purposes. The solid line rectangles belong to the platform's hardware. The solid lines indicate communication interfaces or buses and the dotted lines indicate the programmer interfaces. The main controller is connected with the executing modules by RS 485 bus. The programmer module has also RS 485 interface in order to facilitate debugging or controlling the executing module if the main controller is disabled.

A. Main controller

The main controller is responsible for controlling the executing modules connected to the RS 485 bus, storing logs in its memory and communication with users by USB or Ethernet interface. The modular schematic of the main controller is presented in Fig. 2. The functional modules are presented as solid line rectangles and connectors or

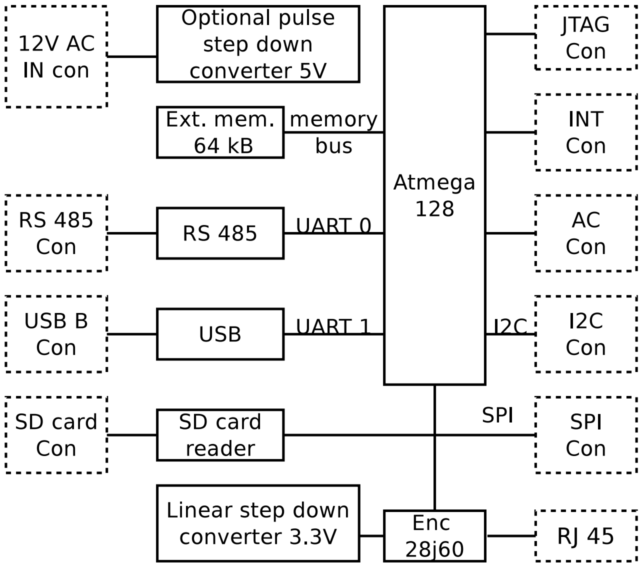


Figure 2. Ideological schematic of the main controller

jacks are presented as dotted line rectangles. The main controller consists of: microcontroller Atmega128, 64 kB of data external memory, USB interface (Ft232Rl chip), RS 485 interface (Max481 chip), Ethernet interface (Enc28j60 chip) and Secure Digital card reader. The microcontroller uses SPI bus to communicate with Ethernet driver and SD card reader. It is also possible to connect 8 additional devices to this bus through a SPI connector placed on the main controller. The controller also has the optional pulse step down converter, which can be useful if we want to use external power supply. This converter provides power supply to executing modules. In order to communicate with external devices, sensors and modules, the controller uses the following buses: SPI, I2C and RS 485. All buses have their own connectors. In order to reduce costs, a user communicates with the controller by console (VTY100 protocol) connected to USB port. The main controller has neither display nor keyboard. The CPU is programmed using JTAG interface that allows the user to debug the software. Additionally, there is connector (AC Con) with analogue inputs and connector (Int Con) with inputs generating interruptions.

B. Executing module

The executing module is responsible for switching on/off various devices, for example lights or roller shutters in an intelligent home. The executing module consists of: microcontroller Atmega168, RS 485 interface, 4 relays, 2 outputs for LED, two connectors with two inputs each, 5 jumpers for setting device address. The relays have independent power supply in order to avoid brownouts. The executing module can be programmed using SPI bus (STK 500v2 programmer) or RS 485 bus (bootloader with xModem protocol).

C. Programmer module

The Programmer module was designed to provide various functionality and reduce the costs. The programmer module uses USB interface and therefore it does not require additional power supply. Its main function is flashing firmware to the main controller or executing modules. Both devices have different programming interfaces (JTAG and SPI). The constructed programmer provides additional RS 485 and RS 232 TTL interfaces. The JTAG programmer bases on Atmega16 microcontroller and Atmel JTAG ICE firmware, therefore is compatible with AVR Studio. The archetype of SPI programmer is an open source project [7]. The hardware was slightly modified but the firmware remained unchanged. The SPI programmer uses STK 500v2 protocol and is compatible with AVR Studio.

III. Firmware

The firmware was written in C language. The complete source code is available at svn repository http://akme.yum.pl/FreeRtos/FreeRtos, where the login and the password is "student". The firmware part of the presented didactic platform consists of two basic parts: the firmware for the main controller and the firmware for the executing modules. Each device has a different microcontroller and has other functions, therefore it needs specialized firmware. There is embedded RTOS on both modules. The authors chose FreeRtos as RTOS because it is distributed under modified GPLv2 license [8]. FreeRtos uses two methods of providing multitasking: tasks and coroutines. Its kernel needs 4 kB of program memory, hence it is possible to use FreeRtos on microcontrollers with 8kB of program memory. Originally, FreeRtos was ported to the Atmega32. In the case of the proposed platform, it was necessary to make a port for Atmega168 and Atmega128 microcontrollers.

A. Main controller

Main controller is responsible for controlling the executing modules and communication with users. It stores logs and allows to schedule some operation, e.g., moving up the roller shutters. The main modules of the main controller firmware are the following: kernel, Command Line Interpreter, file system, Communication protocol, TCP/IP stack and xModem protocol.

1) Kernel: Multitasking in the main controller is provided with the help of tasks without preemption. Such an approach has numerous and significant advantages. Tasks are simple, have no restrictions on use and support full preemption (not used in our case). Moreover, they are fully prioritized [9]. The firmware was written without preemption, so re-entry to the task does not need to be carefully considered. The main disadvantage is that each task has its own stack. The Atmega128 has 128 kB of program memory and 4 kB of internal data memory extended by external chip to 64 kB and allows us to use FreeRtos with tasks. It is recommended to

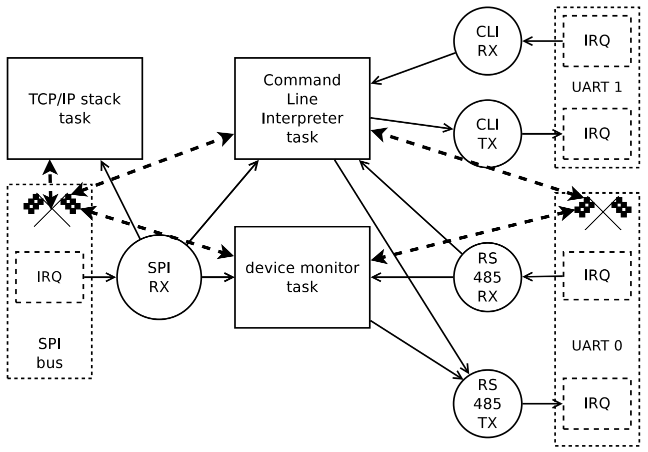


Figure 3. Architecture of main controller firmware

place stacks of the tasks in internal memory, hence there is 4 kB for stacks available. There are three tasks: Command Line Interpreter task, device monitor task and TCP/IP stack task. 4 kB is enough for three stacks. In order to save internal memory, buffers and other structures were moved to two times slower external memory. Constant strings and constant structs are stored in flash memory. In Figure 3 the firmware architecture of the main controller is presented. It bases on the mentioned three tasks.

The Command Interpreter task is responsible for communication with users through a console attached to USB port. This task uses serial port UART 1 for its exclusive use. This simplifies the implementation because it is not necessary to add the semaphore, which is responsible for blocking simultaneous access of many tasks to serial port UART 1. This task uses additionally the SPI bus and serial port UART 0. Other tasks are also using these resources, therefore they required semaphores to synchronize tasks. The semaphore blocks simultaneous access to one of the resources by more than one task. In Figure 3 the semaphores are marked by a racing checkered flag symbol. When the task is attempting to enter the critical section (e.g., read or write to serial port UART 0), it has to pass through the semaphore. If the semaphore is locked, the task is suspended as long as the semaphore is locked. Once the semaphore is unlocked, the task is released automatically and the semaphore is locked again by this task. The task unlocks the semaphore again after leaving the critical section. FreeRtos provides special API for handling semaphores. The task is suspended as long as the semaphore is locked, or until its optionally specified timeout.

FreeRTOS supports API for buffer handling in order to simplify the implementation of the main controller firmware.

There is a special function for writing to the buffer. If the buffer is full, the task is suspended as long as the buffer is full and optional specified timeout is not exceeded. The function informs (returns the result) if the operation was successful or not. Similarly there is a function for reading the buffer. If the buffer is empty, the task is suspended. The task is released when data is available in the buffer or timeout is exceeded. All the mentioned FreeRTOS API functions are not blocking. If the task is suspended, the microcontroller is executing other, not suspended, tasks. The developer has to care about avoiding deadlocks. Programming tasks is thus complementary to the operating systems theory within the range of topics related to deadlocks.

The task of the device monitor is to check the state of modules connected to RS 485 bus or SPI bus. This includes polling all devices connected to the RS 485 bus, reading analogue inputs values and communicating with devices connected to SPI bus (e.g., RTC clock). The task uses the resources such as SPI BUS or serial port UART 0. The task is synchronized with other tasks by semaphores.

The TCP/IP stack task is responsible for listening and establishing new TCP connections and handling them. The task uses SPI bus and is also synchronized. This tasks has a lower priority than two other tasks.

2) Command Line Interpreter: The main controller provides interactive communication with a user thanks to Command Line Interpreter. Initially, the CLI was taken from the AVRlib project [10]. Original CLI was not designed for multi task environment: only one instance of CLI was possible and, furthermore, it was working on global variables. The original CLI was not ready to cooperate with *stdio* C library. As a result, for the purpose of the proposed platform, most of the codes of the original CLI has been rewritten. Now, it is possible to use many independent instances of CLI. Each CLI has the history of 4 last commands and works on a new engine. The proposed CLI is compatible with *stdio* library and it is possible to use *fprintf* functions in order to make a print.

The new CLI API is friendly (it allows users to add new commands easily) and communication with the main controller is simple. The command *help* displays all available commands and its description.

3) File system: An important part of operating system theory is devoted to file systems. For the purpose of the didactic platform, a simple file system, the so-called Fat8, has been written. It can address up to 256 clusters. Each cluster, contrary to CP/M operating system, has 256 bytes instead of 128, what simplified the file system implementation. The whole implementation takes about 500 lines of code and is compatible with avr-libc [11] API. The file is visible as a stream. Writing to a file is possible using *fprintf* function.

4) Communication protocol: The main controller and the executing modules are connected to a common medium – RS 485 bus. The communication model looks as follows. The main controller (master) starts the transmission on the bus. Each frame sent by the master main controller has an address of a slave device (an executing module) – the receiver of the message. The slave device can answer to the message. The frame format is Type Length Value. The frame fields are the following: synchronization sequence, address, type of message, message length and message data. Two bytes with CRC sum end the frame.

5) TCP/IP stack: The TCP/IP stack implemented in the presented didactic platform is based on the stack proposed within *HTTP/TCP with the Atmega88 microcontroller (AVR web server)* [12] project. For the purpose of our project, the TCP/IP working on Atmega88 with 8 kB of program memory was adopted for multitasking system. The TCP/IP stack is supported in the presented didactic platform only partially. At the current stage, only the ICMP protocol and a simple WWW server is working. The next releases of the didactic platform will also include an implementation of servicing several TCP connections.

6) Xmodem protocol: This protocol allows to send or receive files. It cooperates with *stdio* library and input/output stream. This protocol is useful for bootloader handling. It allows to flash executing module by new firmware image. Implementation of TFTP protocol is much more complicated.

B. Executing module

The executing module controls 4 relays and reads four inputs. It is suitable for controlling two roller shutters or four light sources. Some controlling functions can be fulfilled automatically, e.g., after pressing the button the relay is switched on. The relay state may be changed after receiving special command from main controller.

1) Kernel: The executing module is not well equipped. Its microcontroller has 16 kB of program memory and 1 kB of data memory. In order to save data memory, the FreeRtos is using coroutines. The coroutines share common stack. The coroutines in FreeRTOS are automatically restored by the scheduler and a developer does not need to focus on them. Moreover, they are very portable across other architectures [9]. The disadvantage of the coroutines is a requirement of a special consideration. The lack of stack causes that data stored in local variables are destroyed after restoration of coroutine, what complicates the use of coroutines. The coroutine API functions can be called only inside the main coroutine function. In FreeRTOS, the cooperative operation is only allowed among coroutines, not between coroutines and tasks. For this reason there are only coroutines and no tasks in firmware of the executing module.

In Figure 4 the architecture of the executing module that controls two roller shutters is presented. For driving single roller shutter two relays are required; since one executing module can coordinate two roller shutters. The firmware consists of 4 coroutines, presented in Fig. 4 as solid line

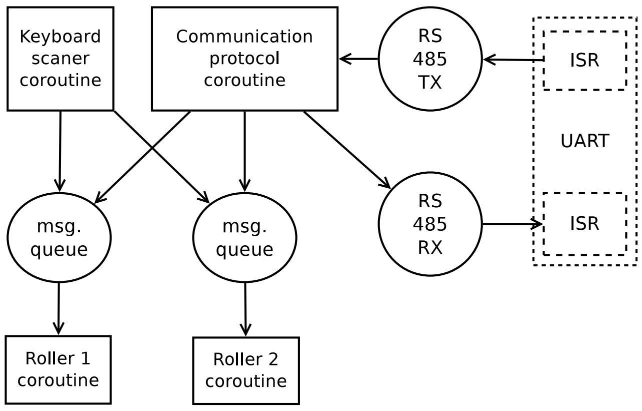
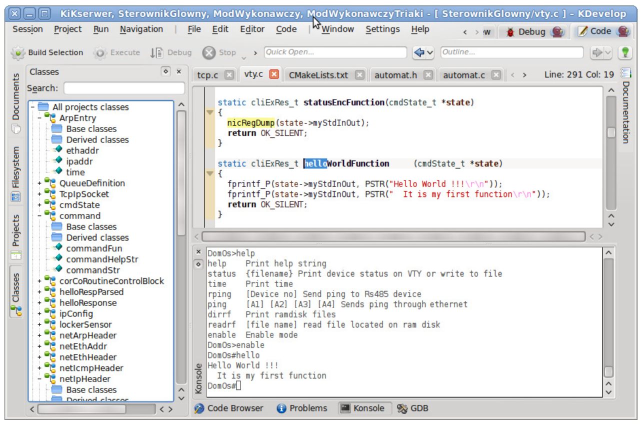


Figure 4. Architecture of executing module firmware


KiKserwer, SterownikGlowny, ModWykonawczy, ModWykonawczyTriaki - [SterownikGlowny/vty.c] - KDevelop
Session Project Run Navigation File Edit Editor Code Window Settings Help
Debug Code
Build Selection Execute Debug Stop
Quick Open... Outline...
Classes
Search:
tcp.c vty.c CMakeLists.txt automat.h automat.c
Line: 291 Col: 19
Code Browser Problems Konsole GDB

```
static cliExRes_t statusEncFunction(cmdState_t *state)
{
  nicRegDump(state->myStdInOut);
  return OK_SILENT;
}

static cliExRes_t helloWorldFunction     (cmdState_t *state)
{
  fprintf_P(state->myStdInOut, PSTR("Hello World !!!\r\n"));
  fprintf_P(state->myStdInOut, PSTR("  It is my first function\r\n"));
  return OK_SILENT;
}
```

```
DomOs>help
help    Print help string
status  {filename} Print device status on VTY or write to file
time    Print time
rping   [Device no] Send ping to Rs485 device
ping    [A1] [A2] [A3] [A4] Sends ping through ethernet
dirrf   Print ramdisk files
readrf  [file name] read file located on ram disk
enable  Enable mode
DomOs>enable
DomOs#hello
Hello World !!!
  It is my first function
DomOs#
```

Figure 5. IDE and Hello world function

rectangles. Two coroutines drive the rollers, additionally there is a coroutine that scans the keyboard connected to the executing module and another one responsible for communication within RS 485 bus. The coroutines communicate with each other by 2 buffers presented in Fig. 4 as circles. The coroutine responsible for communication with RS 485 bus can send appropriate commands to driving roller shutter coroutine with the help of the buffer. The same buffer can be used by the scanning keyboard coroutine to send a message. The messages sent by the buffer includes information about relay (its number), which should be switched on or off at a specified time.

2) *Communication protocol:* Executing modules work as slave devices. The communication is always started by a master device by sending a message with a slave device's address (destination address). All slave devices check destination address of the received messages. If the address is matching, the slave device answers and executes the command issued by the main controller. In most cases, messages with not matching address are ignored. There is only one exception to this rule, which is presented in the next section.

3) *Bootloader:* The bootloader is mainly used when STK 500v2 programmer is not available or when it is not connected. The main controller can flash firmware to the executing module. With the help of the xModem protocol the firmware image is first uploaded to the main controller and stored in a file. Next, the main controller sends restart command to the executing module and if the address is matched, the device restarts. Otherwise, the device disconnects from RS 485 bus for 60 seconds – this is enough to write firmware to the executing module. After restart of the executing module the bootloader code is executed. The bootloader waits 30 seconds for flash command. After receiving it, the executing module is trying to download firmware using the xModem protocol. The main controller sends firmware according to the xModem protocol.

4) *Keyboard scanner:* There is a coroutine described in Section III responsible for keyboard scanning. It can distinguish a key press from stick bumping on keyboard.

C. *Software tools*

The prepared toolset for the platform purposes works on Linux and consists of editor (Integrated Development Environment – IDE), compiler, repository and programmer software. In the Ubuntu distribution all of required programs are available in its repositories and can be installed using apt-get install command. Thanks to this advantage it is very easy to write the instruction for students, explaining how to prepare the system for work.

IV. LABORATORY EXERCISES

The presented platform allows users to prepare many exercises. For example, students can add new commands to main controller's CLI as it is shown in Fig. 5, as well as modify the interface of the application programming system File System API. It is also possible to add a new task to the main controller that is periodically sending messages to executing modules, for example to move roller shutters up or down.

An exemplary exercise for students can be focused on modification of executing module firmware. It is possible to use the execution module for controlling 4 light sources. The code is very similar to the code of executing module that is controlling roller shutters. The architecture of the firmware is presented in Fig. 6. Each light source, analogically to roller shutter firmware, is controlled by a separate coroutine. Therefore, it was necessary to add two more coroutines. Each coroutine that controls light shares the same code and has its own buffer with messages. The message format is one byte long. There is information in the message for how long the light has to be switched on. If the value is equal to zero, the light has to be switched off. The algorithm of the coroutine is shown in Fig. 7. Initially – during the initialization phase – the light is switched off, therefore

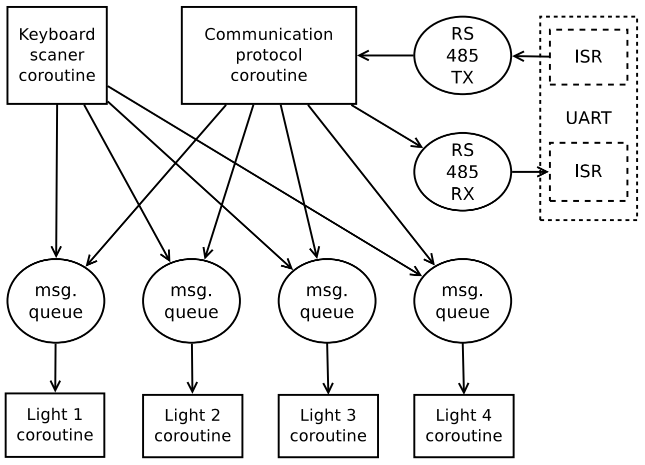


Figure 6. Architecture of executing module firmware controlling 4 light sources

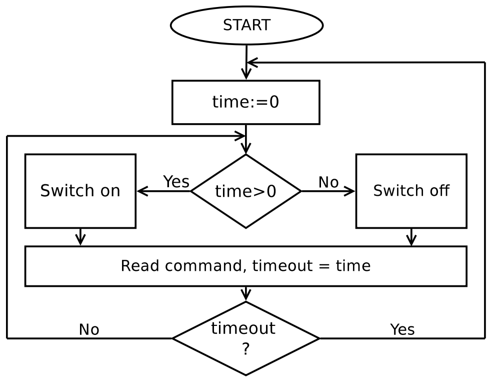


Figure 7. Algorithm for coroutine handling single light source

the variable *time* is equal to zero (analogically to message format). In the next step, the coroutine checks the value of *time* variable. If it is grater than zero, the light is switched on. Otherwise, the light is switched off. Next, the coroutine is waiting for a new message in the buffer, not longer than the time of switching on the light. If the timeout is exceeded and there is no message, the algorithm goes back to the initialization phase and the light will be switched off in the next step. If there is a new message, the light is switched on for a time specified in the message.

The solution based on using multitasking is easier and more readable. Each coroutine in this case is responsible for a different job.

V. Conclusion

The presented didactic system is a valuable addition to the theory of operating and embedded systems. It enables students to get familiarized with the aspects like multitasking, interprocess communication, and process synchronization. The platform has been designed in such a way as to facilitate its quick and easy implementation. For this effect AVR microcontrollers, which are increasingly popular among students taking interest in electronics, have been used. The presented solution is inexpensive and most of students can afford to build the presented platform and use it for didactic or practical purposes limited only by their imagination.

The paper focuses on technical details to assists other users in building a similar didactic platform that would support teaching of the theory of operating systems. Because of the limited size of the present article, only a selection of possible didactic exercises supported by the platform is presented, while further possibilities are only indicated.

Further refinement of the platform, with active participation of students, will include an implementation of the TCP/IP stack, telnet server (enabling CLI communication) and the operation of the FAT32 file system and SD cards.

The aptitude tests carried out among students indicate that the introduction of the presented didactic platform to the process of teaching the theory of operating systems have resulted in a considerable improvement in the exam performance (by 25%).

Acknowledgement

The authors would like to thank all the developers taking part in open source projects cited in the article.

References

[1] R. Love, *Linux Kernel Development*, 2nd ed. Novell Press, 2005.

[2] W. R. Stevens and S. A. Rago, *Advanced Programming in the UNIX Environment*. Addison-Wesley, 2005.

[3] F. S. Foundation, "Gnu general public licence v2," http://www.gnu.org/licenses/gpl-2.0.html, 1991.

[4] Cadsoft, "Eagle," http://www.cadsoft.de, 2010.

[5] Atmel, "Atmega128 datasheet," http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf, Aug. 2010.

[6] ——, "Atmega168 datasheet," http://www.atmel.com/dyn/resources/prod_documents/doc2545.pdf, Jul. 2010.

[7] G. Socher, "Avrusb500v2 – an open source atmel avr programmer, stk500 v2 compatible, with usb interface," http://tuxgraphics.org/electronics/200705/article07052.shtml.

[8] FreeRTOS, "Copyright notice," http://www.freertos.org/copyright.html.

[9] ——, "Freertos api reference," http://www.freertos.org.

[10] P. Stang, "Procyon avrlib api," http://www.mil.ufl.edu/~chrisarnold/components/microcontrollerBoard/AVR/avrlib, 2006.

[11] "Avr-libc api," http://avr-libc.nongnu.org/, 2010.

[12] G. Socher, "Http/tcp with an atmega88 microcontroller (avr web server)," http://www.tuxgraphics.org/electronics/200611/embedded-webserver.shtml, 2006.