A Didactic Platform for Practical Study of Real Time Embedded Operating Systems

Adam Kaliszan
Chair of Communication and Computer Networks
Poznan University of Technology
ul. Polanka 3, 60-965 Poznań, Poland
Email: adam.kaliszan@gmail.com
http://www.adam.kaliszan.yum.pl

Mariusz Głąbowski
Chair of Communication and Computer Networks
Poznan University of Technology
ul. Polanka 3, 60-965 Poznań, Poland
Email: mariusz.glabowski@put.poznan.pl
http://glabowski.eu

***Abstract*—The article proposes a new didactic platform for practical study of embedded Real Rime Operating Systems (RTOSs). Three fundamental parts that are included in the platform are discussed in detail: the hardware part, the firmware part and the software tools. In the description of the hardware part the following parts are addressed: main controller, input/output module, executing module and programmer module. 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. The firmware part of the platform proposed and described in the article implements: the command line interpreter, file system, the protocol for communication between main controller and executing modules, TCP/IP stack and xModem protocol, among others. All the software tools work on the Linux operating system and are free of charge; most of them have open source code. Particular attention is given to a presentation of laboratory exercises that have been worked out in the process. These exercises are designed to facilitate the learning process in the study of embedded operating systems with the application of the proposed didactic platform. The proposed platform is not expensive and is easy to assemble. Most students can afford to build or modify it on their own.**

***Keywords*-Embedded systems; Real Time Operating System; Multitasking; Interprocess communication; Intelligent home.**

I. INTRODUCTION

Practice is an important addendum to any embedded operating systems theory course [1]. 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 certain advantages, such as its versatility, ranging from small embedded devices to powerful supercomputers. Thanks to Linux open source code, there are many written kernel modules [2] supporting new devices, which ensures such a great versatility of the system and makes it applicable in many embedded systems. Microsoft, in turn, 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 [3]. On account of Microsoft .NET framework, it is possible to write software in a very easy way. However, it should be noted that the software produced by Microsoft is not free. Additionally, the fact that its code is closed complicates porting the operating system to new, not particularly common, hardware devices. Hence, its application is limited to a few basic CPU architectures.

Irrespective of a chosen operating system, the practical part of an operating systems course 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 applications. Such laboratory classes do not introduce the subject of embedded systems, nor do they have any connection to the operating system theory, since most laboratories do not cover topics such as multitasking, interprocess communication and its synchronization or operations on file systems. Furthermore, as it is often the case, proposed laboratory exercises in operating systems have little relevance to practical implementations.

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 [4]. 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 essence of presented mechanisms of the operating systems. Additionally, the build time of the embedded Linux requires about one hour, while laboratory classes usually last 90 minutes (at Polish technical universities).

In view of the above-mentioned difficulties the authors felt encouraged to develop a new didactic platform, including hardware, firmware and software tools. In the proposed platform, the handling of mechanisms such as files, multitasking, interprocess communication and process synchronization, have been simplified. The platform was initially presented at AICT 2011 [1]. Due to page limitations, the conference paper includes only the most important assump-

tions and a general description of the operation of the proposed platform. This article extends [1], presenting below a detailed description of all component elements of the platform. We also propose an extensive set of new laboratory exercises that make it possible for students to carry on with practical exercises in embedded real time operating systems unaided and on their own. In particular, we draw the reader's attention to the fact that the proposed platform can be used in controlling the intelligent home (smart home, eHouse) [5][6]. The firmware for the platform can be modified while being implemented in laboratory exercises according to one's needs and wishes, which secures easy and fast expansion of its functionality.

The remainder of the article is organized as follows. Section II presents state of the art. Section III presents the hardware of the proposed platform. In Section IV, the software architecture is described, including the programming software (software development kit) used in the process. In Section V, exemplary exercises conducted with the help of the proposed platform are presented. Section VI concludes the article.

II. Related work

One of the first operating systems developed for educational purposes was Mach system [7][8]. A group of systems represented by the Mach system was developed in academic circles in the years 1985–1994. The solutions delivered have been further adopted to numerous commercial operating systems, such as NeXTSTEP or Mac OS X [9].

The rapid development of software, especially of operating systems, started actually when the idea of GNU open source appeared in 1985 [10][11]. The operating systems, developed in accordance with the GNU idea, such as Linux [12] or FreeBSD [13], came into general use and became so attractive that they have been competing with the commercial solutions since then. The open source (GNU) systems combine the advantages of both the systems developed for educational needs and the systems used commercially.

Unfortunately, from the standpoint of teaching, these systems have become more and more advanced, thus preventing their use in the classes and teaching materials on the basics of operating systems. Simultaneously, students interested in practical issues and applications were not motivated enough to learn typical, education-oriented systems. The operating systems (with an open source) of real-time, dedicated to support the embedded systems [1], were indicated as much easier and optimal type of operating systems, possible to apply into the teaching process. The chapter presents further an analysis on potential use of existing real-time operating systems, programming environments and libraries in the teaching process. The analysis is limited to the systems comprising a support for the embedded systems. The following criteria were taken into account during the overview of existing solutions: an open source written in C, a support for the AVR architecture (because of rich microcontroller equipment, a simple and functional set of instructions, a free set of tools including C language compiler and a common presence in the projects for beginner constructors), a liberal license granting system, a support for the controller handling the Ethernet interface, an ability to be embedded on any microcontroller. Particular attention was paid to the latter criterion. Once met, it helps to replace a program written in a single thread, where a complicated loop implementing many tasks is made, with a multi-threaded program, where each step is implemented with a separate thread. Such approach increases the readability of the program (in each loop only one step is performed) and facilitates the division of work.

The first operating system to be considered in the article is Ethernat Nut/OS. The project of Nut/OS operating system is free of charge, open, BSD-licensed. It is a real-time operating system with a stack of TCP/IP network protocols, which supports the AVR, ARM7, ARM9 microcontrollers [14]. There are many projects of evaluation boards for this system. It should be noted, however, that these are complex devices that support, e.g., the embedding of Linux system. There is no option to embed the Nut/OS system on the simplest microcontrollers. The system features are: sustainable use of resources, support for multi-threading, dynamic memory management, but above all an implementation of TCP/IPv4 stack. The code is written in C. The software tools are prepared for Linux, Windows and MacOS. The system requires 32 kB memory – in case of the AVR microcontroller, an external bus is then required in order to add an external memory. The memory bus is used also to work with different types of peripherals, such as Ethernet controller. Particular emphasis in the project was placed on the optimization of supporting the peripherals, with the use of memory bus. This approach caused the appearance of difficulties with the servicing of devices applying the serial bus.

Arduino [15] can also be considered as an interesting didactic platform. It was designed to build, develop software for and be applied in simple devices. It includes both hardware elements and software tools. The hardware elements were made using the AVR microcontroller (without an external memory bus), and the processors of AtMega88 family (AtMega168, AtMega328), which differ in program memory capacity. There were numerous modules containing peripheral devices, i.a., Ethernet ENC28J60 controller and SD card reader, developed for the Arduino platform. The programming language is Arduino Programming Language [16] based on Wiring [17]. This language has the same syntax as C and contains a series of macros and functions for hardware abstraction. The applied abstraction helps with hiding some configuration details, e.g., the registers controlling entry/exit ports. Unfortunately, this platform is not a good element facilitating the teaching of operational systems

basics, because a major part of its functionality, typical for operating systems, is omitted therein, such as multitasking and mechanisms of inter-process communication.

One of the most interesting current real-time operating systems, from the standpoint of the teaching of the basics of operating systems and embedded systems, is FreeRTOS [18] system. It was written in C and is licensed in accordance with a modified GNU GPL [19] license. It supports many microprocessor families (27 processors architectures). This system has been designed for minimal requirements. Its kernel takes 4 kB of program memory and it needs 0.5 kB of data memory to be embedded. With such small requirements the system may be applied to almost any microcontroller. FreeRTOS system supports pre-emption and multi-threading; additionally, the co-routines have been implemented therein. One of the advantages, deciding its selection for the purposes of the teaching platform discussed in the article, are good documentation and a large group of users and developers, which provides a good, long-term support for the platform. The free version is devoid of libraries designed to handle FAT 32 file system. Universal libraries working on all platforms to support network interfaces are not added either. Simultaneously, getting familiar with the FreeRTOS system enables the learners to use also the commercial versions of this system. An interesting extension of the FreeRTOS system is, e.g., OpenRTOS. OpenRTOS is FreeRTOS, provided under a commercial license that makes no reference to the GPL and includes fully featured professional grade USB, file system and TCP/IP components [18].

III. Hardware

Figure 1 shows a schematic diagram of the didactic platform described in the article, whereas Figure 2 shows a photo of the platform. The system is distributed and consists

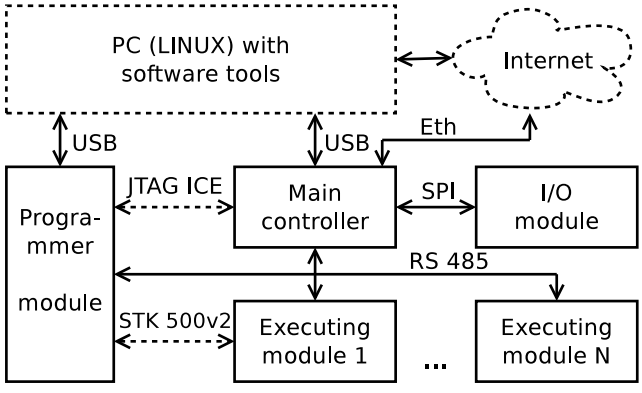

Figure 1. Modular schematic of the platform's hardware

of the main controller, optional input/output (I/O) module and executing modules. The main controller and the executing module are programmed using a universal programmer designed and custom-developed for the platform's 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 an RS 485 bus. Additionally, the input/output module (I/O module) is connected to the main module by the SPI bus. The programmer module also has RS 485 interface in order to facilitate debugging or controlling the executing module if the main controller is disabled.

The hardware part was designed with the help of a freeware version of Eagle [20] CAD software. The dimensions of the PCB board were limited to 10 by 8 centimeters (i.e., the maximum dimensions of PCB board allowed by freeware version of Eagle CAD software). The complete project of the hardware is available at svn repository http://rtosOnAvr.yum.pl/hardware/ssw [21], 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 dimensions of the board allow 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 via USB, since RS 232 is not very common in modern personal computers. There is a place on the main controller for a power converter. It allows the platform to work as a stand-alone device that does not require power supply from the USB port. The hardware project is based on AVR microcontrollers [22][23]. This reduced 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.

A. Main controller

The main controller is responsible for controlling the executing modules connected to the RS 485 bus and the I/O module, storing logs in its memory, and communicating with users via a USB or Ethernet interface. The modular schematic of the main controller is presented in Figure 3. The functional modules are presented as solid line rectangles and connectors or jacks are presented as dotted line rectangles. The main controller consists of: microcontroller Atmega128, 64 kB of data external memory, USB interface (Ft232Rl chip [24]), RS 485 interface (Max481 chip [25]), Ethernet interface (Enc28j60 chip [26]) and Secure Digital card reader. 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. A diagram of PCB board of the main controller that includes its most important elements and connectors is shown in Figure 4.

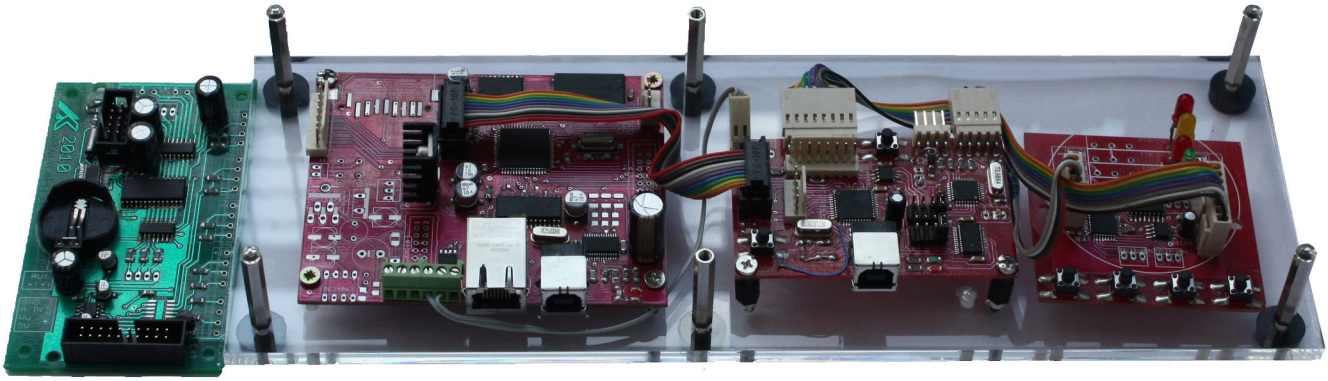

Figure 2. Didactic embedded platform: I/O module, main controller, programmer execution module.

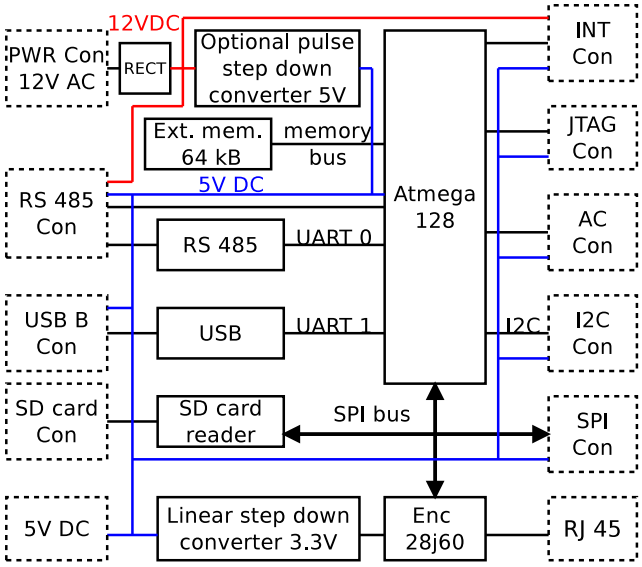

Figure 3. Ideological schematic of the main controller

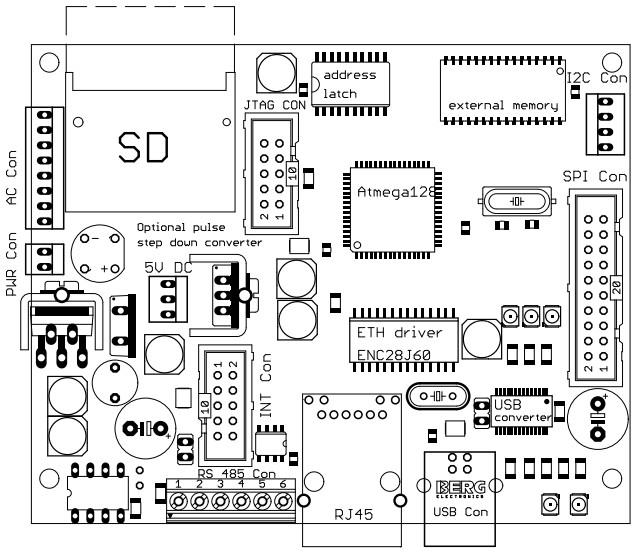

Figure 4. Main controller PCB with connectors

The microcontroller uses the SPI bus to communicate with the Ethernet controller and the SD card reader. It is also possible to connect 8 additional devices to this bus through an SPI connector placed on the main controller. The SPI connector can also be used in hooking up the input/output (I/O) module, as it is presented in Figure 1. Only memory is connected to the memory bus. The bus has no connector led out of its housing, i.e., no additional system can be connected to it.

The controller also has the optional 5 V pulse step down converter and a rectifying bridge. These elements of the controller can be useful if we want to use an external power supply because both the processor and other systems of the platform are powered by a 5 V voltage source. This power supply can be supplied either by a pulse converter or an USB port. In Figure 4, the external 12 V power supply line that leads from the rectifying bridge is denoted with the colour red and the 5 V power supply with the colour blue. The main controller provides power supply to all modules that are attached to it and, hence, each connector (or jack), to which any module can be hooked up, has its connection for power supply of its own.

As it was mentioned earlier, when a 5 V pulse converter is not available, the system can be powered from an USB port. With the application of this type of power supply, however, the A/D converter in the input/output module does not operate properly. Its analogue part is powered by a 5 V voltage source from it own linear converter that, at its output, requires at least 8 V. When it is powered by 5 V voltage (from an USB port), it will give output voltage lower than 5 V, and thus the A/D converter will be operating improperly. For didactic needs, it is possible to change the characteristics of the Input/Output module and make the analogue part of the converter powered by the 5 V voltage from the USB port. The execution module requires the power supply of 12 V to switch its own relays. Instead of 12 V, it is possible to supply 5 V (from the USB port) and introduce relays that operate under 5 V. Despite certain

inconveniences, the system powered from the USB port is fully operational and functional in didactic situations. Lack of external power supply makes it easier to hook up and work with the set during lab classes (it is sufficient to supply power to the device from an USB port of the computer).

In order to reduce costs, the user communicates with the controller via console (VTY100 protocol). Access to the console is available both via the USB interface and the Ethernet interface. The main controller has neither display nor keyboard. The CPU is programmed using the JTAG interface that allows the user to debug the software. Additionally, there is a connector (*AD Con*) with analogue inputs and a connector (*Int Con*) with inputs generating interrupts. The connector *RS 485* provides proper access to earth ground, and the voltage 5 V and 12 V that provides power supply to the executing modules. Additionally, an input to the microcontroller has been introduced that can generate interrupts. This allows for a modification of the protocol operating on the RS485 bus so that the devices connected to the bus could impose service demands. This, in turn, makes it possible to eliminate the necessity for continuous checking of executing modules. The main module has a limited number of lines that can operate as input/output. The number of inputs/outputs can be, alternatively, made higher as a result of the application of an I/O (input/output) module that is connected to the SPI connector.

B. I/O module

The schematic diagram for the I/O module proposed in the article is presented in Figure 5, while its printed circuit board (PCB), along with a description of its most important components and connectors, is shown in Figure 6. The input/output module is composed of a port expander, an 8-input A/D converter and a real-time clock (RTC). Individual elements of the I/O module are presented in Figure 5 as solid line rectangles. Connectors are presented as dotted line rectangles.

The inputs of the I/O module are led out in such a way as to make them capable of being used during laboratory classes, those considered in the article, and for solutions of the type "smart home" (intelligent house). The module is connected to the main controller by the connectors *SPI Con* and *Int Con*. Connector *SPI Con* addresses and communicates with individual elements of the I/O module, whereas connector *Int Con* provides 12 V power supply and receives interrupts from the I/O module.

In the I/O module, the port expander is implemented with a MPC23S17 chip [27], which is connected to the SPI bus and the address line 7. The address line determines whether the system can use the SPI bus. This system has two 8-bit ports. Each line of the port can operate as input or output. In the I/O module seven lines from each of the ports operate as output, while the last one is not used. Each port of the expander is connected to a separate line

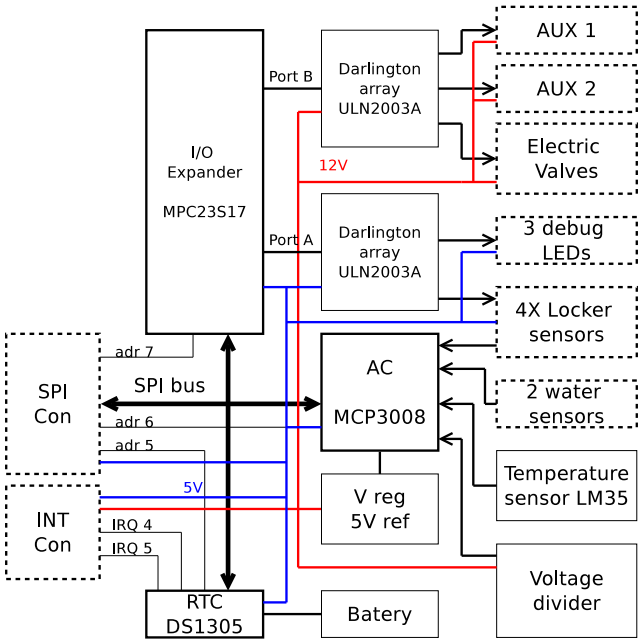

Figure 5. Ideological schematic (schematic diagram) of the I/O module

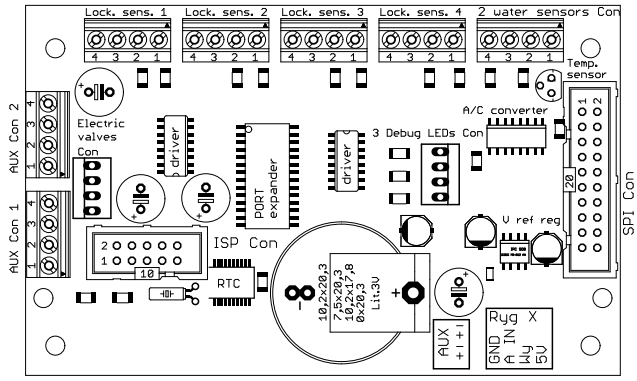

Figure 6. I/O module's PCB with connectors

of the (high-voltage) high-current controller implemented with the use of the high-voltage high-current Darlington transistor array (ULN2003A chip [28]). This system enables controlling devices whose consumed power exceeds 10 mA (current running though a single output of the expander cannot exceed 10 mA). The ports of the expander control different voltage. Port A controls devices that are powered by 5 V voltage, whereas port B controls devices that are powered by 12 V voltage. Despite the high current that the expander can control – after the application of transistors – one should not forget about the limited power of the power-supply unit and the limited current voltage that can run through the rectifying bridge in the main controller. In the case of an excessive load, the voltage on the power-supply line 12 V may drop.

For the convenience of the the didactic platform con-

sidered in the article, in the I/O module, some outputs are assigned to dedicated applications. Four outputs that control 5 V devices are designed to operate (flash) the diodes in lock bolt (valve) sensors and each of them is led out with a separate connector *Lock sensor*. The remaining three lines are led out with the application of the connector *3 debug LEDs*. Three outputs from port B, that control devices powered by 12 V voltage, are dedicated to control electro-valves and are led out through the connector *Electrics Valves*. The remaining four lines are led out with the application of two connectors: *AUX1* and *AUX2*. They can be used to control additional devices.

The A/D converter (MCP3008 chip [29]) located in the I/O module is connected to the SPI bus and to the address line 6 that allows it to occupy the SPI bus. The analogue part of the converter is powered from the output of the linear converter. In order to work properly, this converter requires an external 12 V power supply. In the case of a construction of a system that is to be powered from/by a USB port (without external power supply), the analogue part of the converter has to be connected directly to 5 V voltage (from the USB port). In the present didactic platform the converter is designed to check the state of valves (lock bolts) and makes it possible to verify whether the flat/house has been flooded. Checking the state of the valves (lock bolts) is based on measurement taking of the decrease in voltage (voltage drop) on the photo-transistor, after its LED diode is illuminated. If the doors are locked, then between the transistor and the diode there is a valve (lock bolt) that blocks admission of light to the transistor. Following this, the transistor does not transmit current and a voltage drop ensues. The flooding control is based, in turn, on a measurement of the voltage drop on the "flooding" sensor. If the sensor is dry, then it does not transmit current and a drop in voltage follows. In the case of flooding, the sensor transmits current and the voltage drop in it decreases. The A/D converter is supplemented with an analogue temperature sensor LM35 [30] and an output from the voltage divider (potential divider) of 12 V power supply. Thanks to the measurement takings in the voltage obtained from the divider, it is possible to determine whether the 12 V power supply line is overloaded or not.

The last system used in the I/O module is a real-time clock. This system is addressed through the address line 5. The real-time clock is placed in the I/O module instead of within the main controller due to the lack of space on the board of the main controller (no available place results from the limitations imposed by the free version of the eagle program). The applied DS1305 chip [31] is capable of generating interrupts. These interrupts are activated along with the alarm activation. Hour and date can be set in the RTC system. The system should remain operational even if power supply is not available, thus it is necessary to connect it to an additional battery cell that keeps its memory

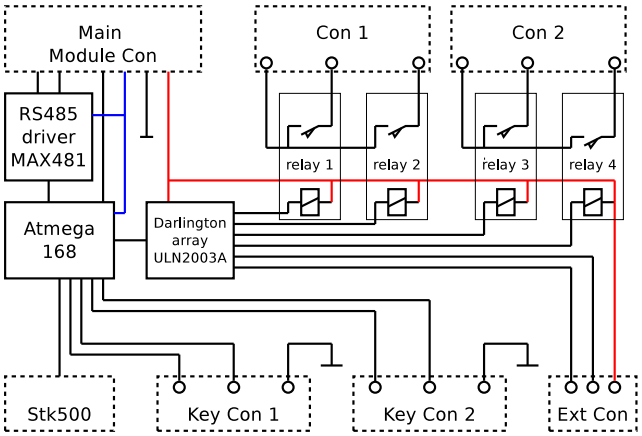

Figure 7. Ideological schematic of the executive module

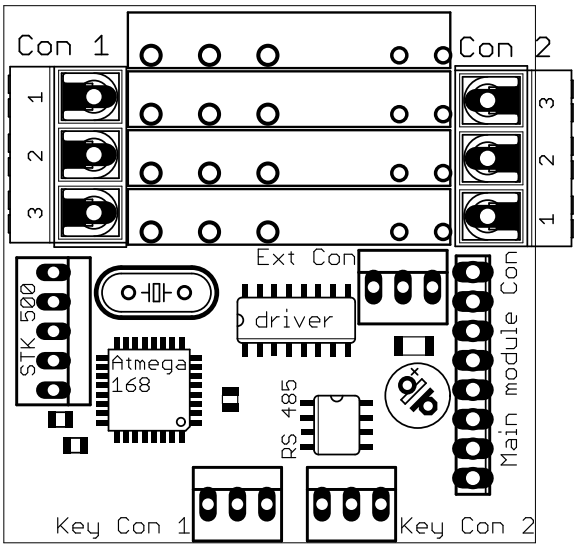

Figure 8. Execution module's PCB with connectors

running and provide power to the clock system. In order to make use of alarms generated by the system, it has to be connected with the connector *INT Con* to the inputs of the microcontroller that generates IRQ 4 and IRQ 5 interrupts.

C. Executing module

The executing module is responsible for switching on/off various devices, e.g., lights or roller shutters in an intelligent home. Figure 7 shows a schematic diagram of the executing module, whereas Figure 8 presents the arrangement (deployment) of its most important elements and connectors on the PCB board.

The executing module consists of: microcontroller Atmega168, RS 485 interface MAX481 [25], Darlington array ULN2003A and four relays. The module is equipped with a number of connectors that are shown in Figure 7 as dotted

line rectangles. Connector *Main Module Con* is designed to connect the executing module to a common bus that serves the remaining modules and the main controller. The connector provides power supply and the connection to the RS 485 bus. Additionally, it has an additional line led out that can be used for different purposes in the future, e.g., to reset executing modules or to send interrupts to the main module. The module has two connectors (*Con 1* and *Con 2*) with relay joints. Two receivers (e.g., sources of light and their power supply) can be connected to each of these connectors. Additionally, the module is supplemented with the connector *Ext Con*, to which additional diodes or other low-power receivers can be connected.

The Atmega 168 microcontroller is programmed via SPI (connector *Spi Prog*). This means that there is no possibility to withhold from an execution of the program and to view the state of registers of the microcontroller. What may be useful then is to connect additional diodes to connector *Ext Con*. These will be instrumental in determining the current state of the device.

The executing module operates four buttons that control the devices. The outputs envisaged to accommodate the buttons are led out on the connectors *Key Con1* and *Key Con 2*. The executing module works as a slave device on the RS 485 bus. Its address can be set up with the help of five jumpers. Two of them are arranged at the stage of the preparation of the PCB board (by welding in resistors 0 Ω), whereas the remaining three jumpers are led out on the connector of the STK500 programmer.

The relays have independent power supply in order to avoid brownouts. Its voltage depends on the relays used (in the case of external 5 V power supply it is recommended to use special relays). Voltage is also supplied to connector *Ext Con* that can control low-power receivers. These receivers are powered with the same voltage as the relays. The executing module is realized with relays purposely, since this arrangement simplifies the preparation of the driver for didactic purposes. Working of the relay is audible and any device, including those that operate in low voltage, can be connected to it. In place of a relay, a LED diode can be appropriately connected. The LED diode provides information about the state of the output. The set presented in Figure 2 has been prepared in this way. In place of supplying power to the relay coil, a LED diode has been welded in and connected serially with a 330 Ω resistor.

Additionally, the repository [21] includes projects of other executing modules that have been realized using triacs with zero crossing circuit. An application of such a module exclusively for didactic purposes, however, would be impractical as this solution would require connecting devices that are powered by 230 V voltage and any work with the module would require special safety precautions to avoid electric shocks. The application of the executing module with triacs enables students to apply learned skills in practice

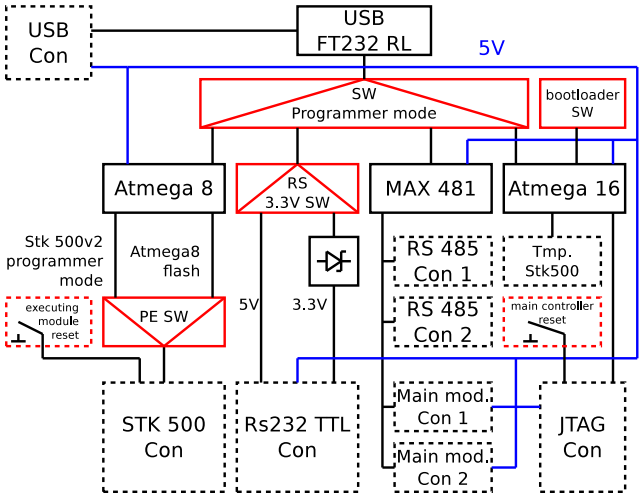

Figure 9. Ideological schematic of the programmer

because they are in a position to apply the presented set to control devices in an e-House. For this particular purpose, the module has been designed in such a way as to be easily accommodated in a standard recessed box (flush-mounted box). Its inputs have been additionally secured by adding zener diodes, while triacs have been secured by adding an appropriate RC circuit. Power supply to the external devices is secured with a varistor in such a way as to avoid any damage to triacs in case of power (voltage) surge in the mains. One should not forget, however, that there are certain differences in a module that is supported by relays and the one supported by triacs (the latter lacks appropriate place for the connector *Ext Con*), which necessitates a slight modification to the software.

The executing module can be programmed using the SPI bus (STK 500v2 programmer) or RS 485 bus (bootloader with xModem protocol). In the case of improper operation of the device (deadlock), unlocking the bootloader mode can be impractical. To make programming with the help of a bootloader possible in such a situation, it is sufficient to connect an additional line on the connector *Main Module Con* to the input reset of the microcontroller.

D. Programmer module

The programmer module has been designed to provide extensive functionality with a simultaneous reduction of costs. The programmer module uses the 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 (main controller and executing module) 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 it is compatible with AVR

Studio. The archetype of the SPI programmer is an open source project [32] and it bases on Atmega8 microcontroller. The hardware has been slightly modified but the firmware has remained unchanged. The SPI programmer uses STK 500v2 protocol and is compatible with AVR Studio.

Figure 9 shows a schematic diagram of the programmer. Individual modules of the programmer are presented as solid line rectangles, whereas the connectors as dotted line rectangles. Additionally, the control switches are shown as rectangles marked with solid red line, while astable switches as rectangles marked with dotted red line.

The mode of operation of the programmer is selected (set) by two control switches. Each of them has to be set (positioned) in the same position for a given mode of operation. These control switches perform the function of a multiplexer. Connector *STK500 Con* is a connector of the STK 500v2 programmer. In addition, it can be used to embed (install) the firmware into the target Atmega8 processor so that the latter could operate as a programmer. For this purpose, the control switch *PE Sw* should be set into (Program Enable SW).

Additionally, the programmer system employs a control switch that allows the voltage of the transmitter of the RS232 port to be reduced to 3.3 V on the output of the connector *RS232 TTL Con*. This option is very useful when the programmer module is used to connect itself to the router console, e.g., Edimax 6104KP. The work on the console and the modifications to the firmware for this device is not, however, the subject for the present article and thus will be omitted.

Leads of the serial port can be used to establish connection with other devices that have a serial port led out with the TTL voltage level or 3.3 V voltage level. The RS485 communication bus is led out on four connectors. This allows for a number of sets to be connected serially so that, ultimately, a network of distributed devices can be built up. While implementing such a solution one has to remember, however, that the system thus executed has only one main controller working or, alternatively, a change of the protocol has to be implemented (e.g., apply a protocol of the type Token Ring, in which main controllers will be passing on tokens to one another that allow them to work on the bus in the master mode and checking the availability of the remaining modules or main controllers operating at the time in the slave mode). Two connectors (*RS 485 Con 1* and *RS 485 Con 2*) have only the RS 485 bus led out. The remaining connectors (*Main controller 1* and *Main controller 2*) have additionally 5 V voltage led out. By selecting the RS485 operating mode, we can monitor and communicate with the executing modules through the RS 485 bus. The solution that has been applied to the didactic platform makes it possible to control the executing modules directly from the computer. The only conditioning element is then to switch off the option of transmission on the bus by the main controller,

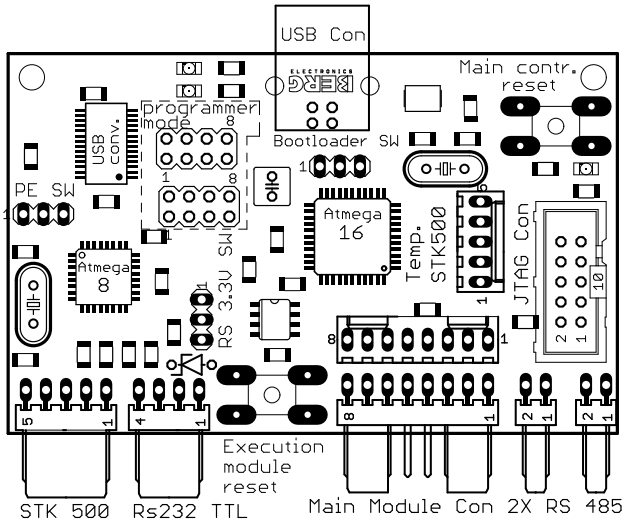

Figure 10. Programmer's PCB with connectors

or an application of some other protocol (e.g., a protocol of the type Token Ring).

The main module can be programmed by a JTAG programmer. It is connected to the connector *JTAG Con*. The JTAG programmer is activated by the connector *Temp. STK500*. The bootloader is installed through the connector. The control switch *Bootloader SW* [33] is used for the activation of the bootloader. With the bootloader, it is possible to download the newest version of the firmware of the JTAG programmer from the Internet. When this is the case, installation of the AvrStudio studio is then required.

The executing module and the main controller can be restarted with the buttons *executing module reset* and *main controller reset* that are placed in the programmer.

IV. Firmware

The firmware was written in C language. The complete source code is available at svn repository http://rtosOnAvr.yum.pl/software/FreeRtos [34], 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 different functions, therefore it needs specialized firmware. There is an embedded RTOS on both modules. The authors have chosen FreeRTOS as the RTOS because it is distributed under a modified GPLv2 license [19]. 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 8 kB of program memory. Originally, FreeRTOS was ported to the Atmega32. In the case of the proposed platform, it has been necessary to make a port for Atmega168 and Atmega128 microcontrollers.

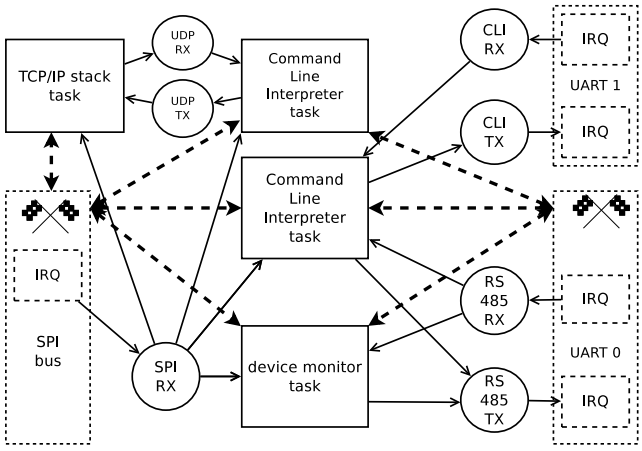

Figure 11. Architecture of main controller firmware

A. Main controller

The main controller is responsible for controlling the I/O module, executing modules and communication with users. It stores logs and allows the scheduling of 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 case of labs excercises). Moreover, they are fully prioritized [35]. The firmware has been 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 place stacks of the tasks in internal memory, hence there are 4 kB available for stacks. There are four tasks: two Command Line Interpreter tasks, a device monitor task and a TCP/IP stack task. 4 kB is enough for four stacks. In order to save internal memory, buffers and other structures have been moved to two times slower external memory. Constant strings and constant structs are stored in flash (program) memory. In Figure 11 the firmware architecture of the main controller is presented. It bases on the mentioned four tasks.

The system supports two simultaneous console sessions. Each session is serviced by separate Command Line Interpreter task. The first task (at the top of Figure 11) is responsible for the communication with the user according to the TCP/IP protocol stack. It reads out the sequence of characters (signs) given by the user from the UDP RX buffer and transmits the reply to the UDP TX buffer. The second task that services the command interpreter operates in a similar way. This task receives data from the UART1 serial port through the CLI RX buffer and transmits data using the CLI TX buffer. This task uses serial port UART 1 for its exclusive use. This simplifies the implementation since the introduction of synchronization is not necessary. In addition, the CLI tasks make use of co-shared resources such as the SPI bus and the UART0 serial port. Since only one task can use the co-shared resources at a given time, it is thus necessary to introduce certain synchronization that enables exclusive access to be implemented. Synchronization can be effected with the help of the mechanisms made available by the FreeRTOS system, such as, e.g., semaphores.

The semaphore blocks simultaneous access to one of the resources by more than one task. In Figure 11 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 a special API for handling semaphores. The task is suspended as long as the semaphore is locked, or until its optionally specified timeout.

FreeRTOS supports an 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 non-blocking functions. 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 the RS 485 bus or the SPI bus. This includes polling all devices connected to the RS 485 bus, reading analogue inputs values and communicating with devices connected to the 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 connections and handling them. Currently work is being carried out on a full implementation of the TCP protocol. Remote access to the console is effected through the UDP protocol. The task uses the SPI bus and is also synchronized. This tasks has a lower priority than the

two other tasks.

The proposed didactic platform does not provide support for preemption. Excluding the preemption allows to error notification – the errors would stay unperceived if the preemption was used. The students deal with the preemption and race condition at high-level programming language courses.

2) Command Line Interpreter: The main controller provides interactive communication with a user via a Command Line Interpreter. Initially, the CLI was taken from the AVRlib project [36]. The original CLI was not designed for a multitask environment: only one instance of the CLI was available 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 code of the original CLI has been rewritten. Now, it is possible to use many independent instances of CLI. Each CLI has the history of its last four commands and works on a new engine. The proposed CLI is compatible with the `stdio` library and it is possible to use `fprintf` functions in order to make a print.

The new CLI API is user-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. In the next section, the method for adding new commands to the interpreter will be discussed.

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 FAT 8, has been written. It can address up to 256 clusters. Each cluster, contrary to the CP/M operating system, has 256 bytes instead of 128, which has simplified the file system implementation. The whole implementation takes about 500 lines of code and is compatible with the avr-libc [37] API. The file is visible as a stream. Writing to a file is possible using the `fprintf` function.

4) Communication protocol: The main controller and the executing modules are connected to a common medium – the 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)* [38] 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 UDP socket are implemented. The next releases of the didactic platform will also include an implementation of IPv6, servicing several TCP connections and WWW server.

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

7) The operation of specific devices that are connected to the SPI bus: The I/O module does not include systems that need software downloading to operate. The systems of the I/O module require, however, appropriate control. Software the controls them is already built into the main controller as firmware.

The following libraries have been created for the specific needs of the platform: A/D converter (MCP3008 system), port expander (MPC23S17 system) and the real-time systems (DS1305). All these systems communicate with the microcontroller through the SPI bus. Each system is connected to a different address line. Addresses of individual systems can be set up in the configuration file of the project (design).

In order to simplify the communication process, a library to handle the SPI bus has also been prepared. This library introduces the possibility of checking the state of the semaphore before an attempt is made to occupy this communication bus by a given process. After entering the critical section, before communication commences, the operating mode for the SPI bus is configured to adjust its configurations to, individual to a given device, processing speed of sending/receiving data. The library responsible for servicing the SPI communication bus also provides two commands for concurrent writing and reading to/from the SPI bus (*spiSendSpinBlock* and *SpiSend*). The first is a blocking operation. The process performs busy waiting until termination of data sending on the SPI bus, whereas the other version of the command is a non-blocking operation. During data sending on the SPI bus the process is suspended (excluding instances of busy waiting), and, within the time offered, the operation system can perform another task. After termination of the sending operation, the task can be resumed. In the case of the work with systems that can communicate with great speed through the SPI communication bus, blocking operations should be applied because switching of a context occupies more time than the operation of sending one byte. If the device works at a low speed and the duration of sending data on the SPI bus takes more than switching of the context twice, then non-blocking operations should be used. After termination of the use of the communication bus, the process must release it.

The libraries responsible for the service of particular

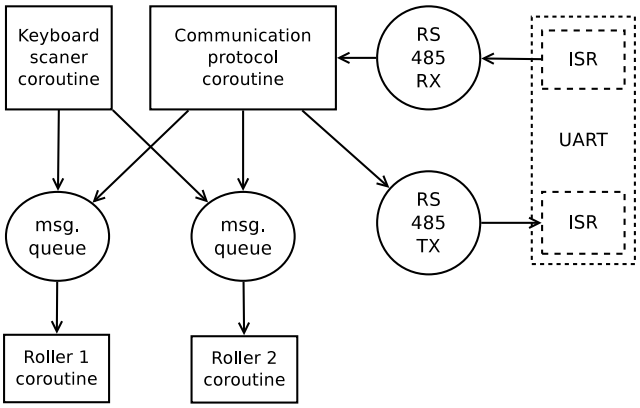

Figure 12. Architecture of executing module firmware

systems, e.g., the RTC clock, make use of the API for the service of the SPI communication bus. This simplifies the implementation of services rendered to other remaining devices. More details related to the control of the devices of the module are included in the following section.

B. Executing module

The executing module controls four relays and reads four inputs. It is suitable for controlling, e.g., 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 has no complex configuration. Its microcontroller has only 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 a 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 [35]. The disadvantage of the application of coroutines requires special consideration. The lack of stack causes data stored in local variables to be destroyed after the restoration of a coroutine, which 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 the firmware of the executing module.

Figure 12 shows the architecture of the executing module that controls two roller shutters. For driving a single roller shutter two relays are required as one executing module can coordinate two roller shutters. The firmware consists of four coroutines, presented in Figure 12 as solid line 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 the RS 485 bus. The coroutines communicate with each other by two buffers presented in Figure 12 as circles. The coroutine responsible for communication with the RS 485 bus can send appropriate commands to the 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 on relays (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 the destination address of the received messages. If the slave device's address matches the message's destination address, the slave device answers and executes the command issued by the main controller. In most cases, messages with not matching addresses are ignored. There is only one exception to this rule, which is presented in the next section.

The coroutine that services the communication protocol communicates with the RS485 bus via the buffers: RS485 RX and TX. These buffers are also used by interrupt handlers such as "Receive Complete" and "Data Register Empty". If a serial port receives a new sign (name), then the Receive Complete interrupt is initiated. In the implementation of the software for the executing module, the service for this interrupt involves placing this sign in the RS485 RX buffer. This is a buffer that is realized in a programmable way with the capacity of 16 bytes. Apart from program buffers, AVR microcontrollers are equipped with sending and receiving buffers for serial ports with the capacity of two bytes. If a sending buffer is available (at least 1 byte is free), then "Data Register Empty" interrupt appears. Handling of this interrupt involves checking the state of the RS 485 TX buffer. If certain data are in the buffer, then they are retrieved and stored in the sending buffer of the device. When they are missing, "Data Register Empty" interrupt handling is activated. This interrupt must be activated after new data are stored in the RS 485 TX buffer. The next section will include a description of the API of the FreeRTOS system designed to handle the buffer by coroutines and the functions handling interrupts.

The initial design for the didactic platform envisages a possibility of an expansion to the communication protocol for the communication bus has an additional line, with which devices of the slave type can generate interrupts. In addition, slave devices can send and read the information on the state of the bus concurrently. This makes them capable of detecting conflicts when a number of devices sends data along a common medium – the RS 485 bus.

3) Bootloader: The bootloader is mainly used when a 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 a restart command to the executing module and if the address is matched, the device restarts. Otherwise, the device disconnects from the 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 the 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 IV responsible for keyboard scanning. It can distinguish a key press from stick bouncing on keyboard.

C. Software tools

The prepared toolset for the platform purposes works on Linux and consists of an editor (Integrated Development Environment – IDE), compiler, repository and programmer software.

Software programs worked out for the purpose of the didactic platform are configured in such a way as to be handled by a freeware Kdevelop 4 editor. For this purpose, the file cmake.txt had to have been written separately and appropriately for each of the projects. It provides an instruction, based on which the Makefile file (in the case of the Linux system) will be generated. In addition, the file cmake.txt informs the editor about the names of files that are included in the project.

Figure 13 presents a screen shot of the editor. The screen shows a project called CLI. Files of this project have been grouped thematically based on the information within a cmake.txt file. Kdevelop, since its version four, stores information on the project in the cmake.txt file (earlier versions used GNU autotools [39], and information about project files was stored in the makefile.am file). Using the program cmake [40], the Makefile file is generated that provides instructions to the program make [41] concerning the method for a compilation of individual files and the way they should be compiled to create the image of the system (hex file) that would be ready to be installed into the microcontroller.

Some sample Makefile files are attached to the projects included in the repository [34]. They can be accessed and used directly without the cmake tool. To do this, it is sufficient to activate the `make` program to compile the project and the `make program` to transfer the image to the microcontroller. This solution is, however, rather inconvenient, especially when errors occur. When this is the case, it is necessary to access the information on errors and then open a given file and find the indicated code line to remedy or eliminate the error.

The addition of the `cmake.txt` file considerably improves code writing. The project can be compiled by pushing the button *Build Section* in in the Kdevelop editor that is framed in red in Figure 13. In the case of an error occurrence, the editor displays an appropriate message and a single click takes us to the erroneous code fragment.

In addition, the Kdevelop environment collects information on all data structures defined in the project and facilitates browsing and managing them. These structures are made accessible after a special bookmark is activated, which is shown in Figure 14.

In the Ubuntu distribution, all of the required programs are available in its repositories and can be installed using the `apt-get install` command. Thanks to this advantage it is very easy to write instructions for students, how to prepare the system to be up and running.

V. Laboratory exercises

The presented platform allows users to prepare an extensive number of exercises, both in operating systems and in embedded systems. The laboratory exercises prepared for the platform can successfully replace exercises that are usually carried out with the help of the Linux system. The two sample laboratory classes presented further on in the section include basic issues related to the theory of operating systems, i.e., multitasking, synchronization of processes, inter-processor communication and the interpreter of commands. At the same time, the proposed platform can also be used as a didactic support to classes in network embedded systems. Students can both design a protocol of their own that would be operative on the RS485 bus, and can modify network protocols and (in the future) a www server.

For the purpose of the teaching process during classes, templates, provided in the repository available at http://rtosOnAvr.yum.pl/software/FreeRTOS [34] in the directory Lab, have been worked out. The project templates are provided in the `templateProjects` directory. The library functions described in the previous section, have been placed in the `freeRtos/Lib`. It is recommended that the contents of this directory should not be modified during laboratory classes. Each of the projects included in the platform has a *makefile* added. The file allows the user to quickly and easily construct a project: all that is needed is to simply type the command: `make` and `make program` to upload the constructed firmware image into the microcontroller. Additionally, the projects include files `cmake.txt` that enable integration of a project with the KDevelop editor.

In the remainder of this section, two sample laboratory exercises that employ the proposed platform are presented, i.e., the exercises "CLI Interface" and "Coroutines FreeRTOS API".

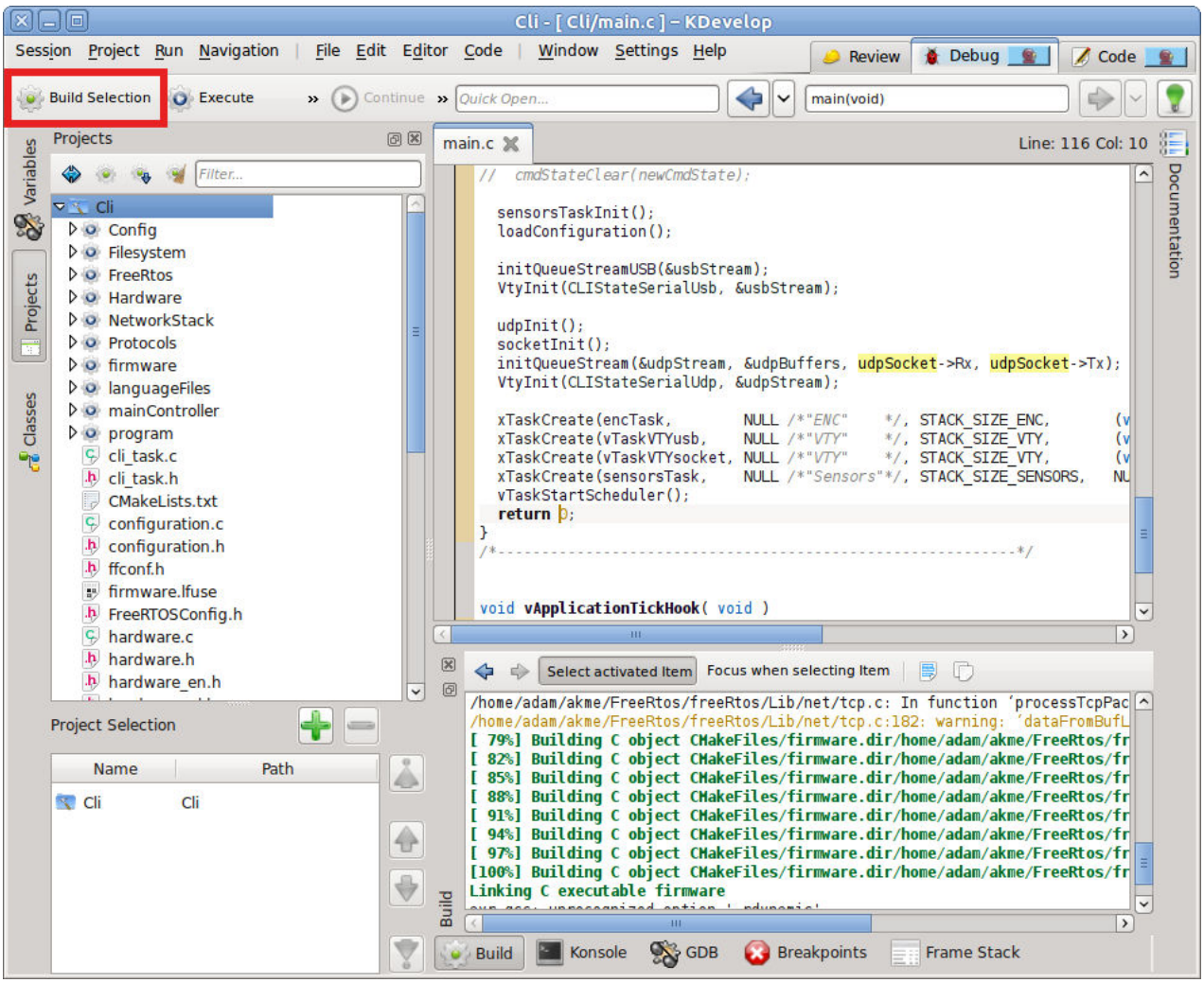

Figure 13. Kdevelop as IDE

A. CLI Interface

During the first laboratory classes students learn in more detail about the option of adding new commands. The approach adopted in the laboratory classes is similar to the approach adopted in programming teaching: during the classes a simple command will be created that, after prompting, will cause the "Hello World !!!" message to be displayed on the screen. The base project template, to which a new command is to be added, is in the directory `Labs/cli`.

The addition of a new command does not require any extensive knowledge of the code structure for the whole of the software for the main controller. In order to achieve the main goal of the exercise it is sufficient to perform the following operations: writing a function that will be executed after the appropriate command and defining the name of the command and complementing it with its description. Each command is written in the `command` structure.

```
struct command
{
  prog_char *commandStr;
  prog_char *commandHelpStr;
  CmdlineFuncPtrType commandFun;
};
```

The structure includes all elements that are necessary in the process of adding a new command. The expression type `prog_char *` defines the index for a string stored in the flash memory of the program. Such strings are handled in a different way than strings (`char *`) stored in data memory. The type `CmdlineFuncPtrType` is an index for the function that executes a command. Each such function accepts the index for the installation of the command interpreter as argument and returns the result that provides information whether the command has been properly executed or not. The declaration of the index for the function is presented below.

```
typedef cliExRes_t
  (*CmdlineFuncPtrType)(cmdState_t *state);
```

The result of the function is defined in the enumerated (enum) type cliRes_t.

```
enum cliExecuteResult
{
  OK_SILENT =0,
  OK_INFORM,
  SYNTAX_ERROR,
  ERROR_SILENT,
  ERROR_INFORM,
  ERROR_OPERATION_NOT_ALLOWED
};

typedef enum cliExecuteResult
cliExRes_t;
```

Type cliExecuteResult includes six feasible results. When the command is properly formed, the value OK_SILENT or OK_INFORM is returned. The latter value is used to inform overtly about the proper execution of the command. Additional values of the enumerated type make it possible to, e.g., provide information on the lack of required parameters or on parameters that have been given inaccurately, in the case of commands that require additional parameters to be furnished. When this is the case, the execution function will return the value SYNTAX_ERROR. With the instance of an error occurrence during the execution of a command, the command interpreter can inform overtly with the message (ERROR_INFORM) or, alternatively, it can leave out the information on the error (ERROR_SILENT). If the execution of a given command is not possible, then the last value of the enum type under consideration will be returned, i.e., the value ERROR_OPERATION_NOT_ALLOWED. In the case of the considered command that prompts the "Hello World" welcome message, information on a properly executed command may not necessarily appear on the screen, therefore the value OK_SILENT will be returned by the function as shown in Figure 14.

In the case of the proposed didactic platform, messages are written similar to the way they are written in the C language, i.e., with the help of the function fprintf_P. The sequence _P means that the text chain is stored in program memory and not in data memory. The index for the output stream is in the structure that stores information on the instance of the command interpreter cmdState_t. This structure will be discussed in more detail in the later part of this section. The command interpreter has been designed in such a way as to make it capable of handling many languages. Hence, all commands and their descriptions are written in separate files, e.g., vty_en.h for the English language, or vty_pl.h for the Polish language. At the stage of adding a command, it is recommended to add in each of these files an appropriate chain so that, after a change in the language, the project could be immediately compiled. Variables that define text chains are labelled according to the following convention: variables that include the name of the command will start with cmd_, whereas variables that include the name of the command along with a description of the command will start with cmd_help_. Thus, for the sample "hello" command under consideration:

```
prog_char cmd_hello[] = "hello";
prog_char cmd_help_hello[]
  = "Writes␣hello";
```

The screen shot from the Kdevelop program that includes the function executed after the "hello" command has been enabled is presented in Figure 14. The name of the function, in line with the adopted convention, ends with a suffix Function. Note that the text is written onto the screen with the help of the function fprintf. This function adopts as the first argument the index for the output stream. At this point, the application of the PSTR macro as the second argument needs certain explanation here. This macro imposes an inclusion of stings in the memory of the program instead of, as it is adopted conventionally, in data memory.

The last element related to the addition of a command is to place the structure with the added command in an appropriate command table. In the example presented in Figure 14, the command has been placed in the menu at the privileged level. Therefore, the table cmdListEnable included in the vty.c file should be completed with yet another type command_t.

```
command_t __ATTR_PROGMEM__
  cmdListEnable[] =
{
  {cmd_help, cmd_help_help, helpFunction},
  ...
  {NULL, NULL, NULL}
};
```

The table cmdListEnable is placed in the memory of the program (attribute __ATTR_PROGMEM__). This mean that an increase in the number of commands will not lead to a decrease in the available data memory. A reduction in the available data memory could eventually lead to a situation where the system simply hangs. In addition, a special command status has been added to the system. The command returns information on available memory. If the obtained value has a negative number, then it is necessary to decrease one of the stacks of tasks being serviced. Otherwise, the stack or the buffer cache will overlap a section of the memory cache that is occupied by global variables, which, in consequence, will block the operation of the system or will render the operation of the system unstable.

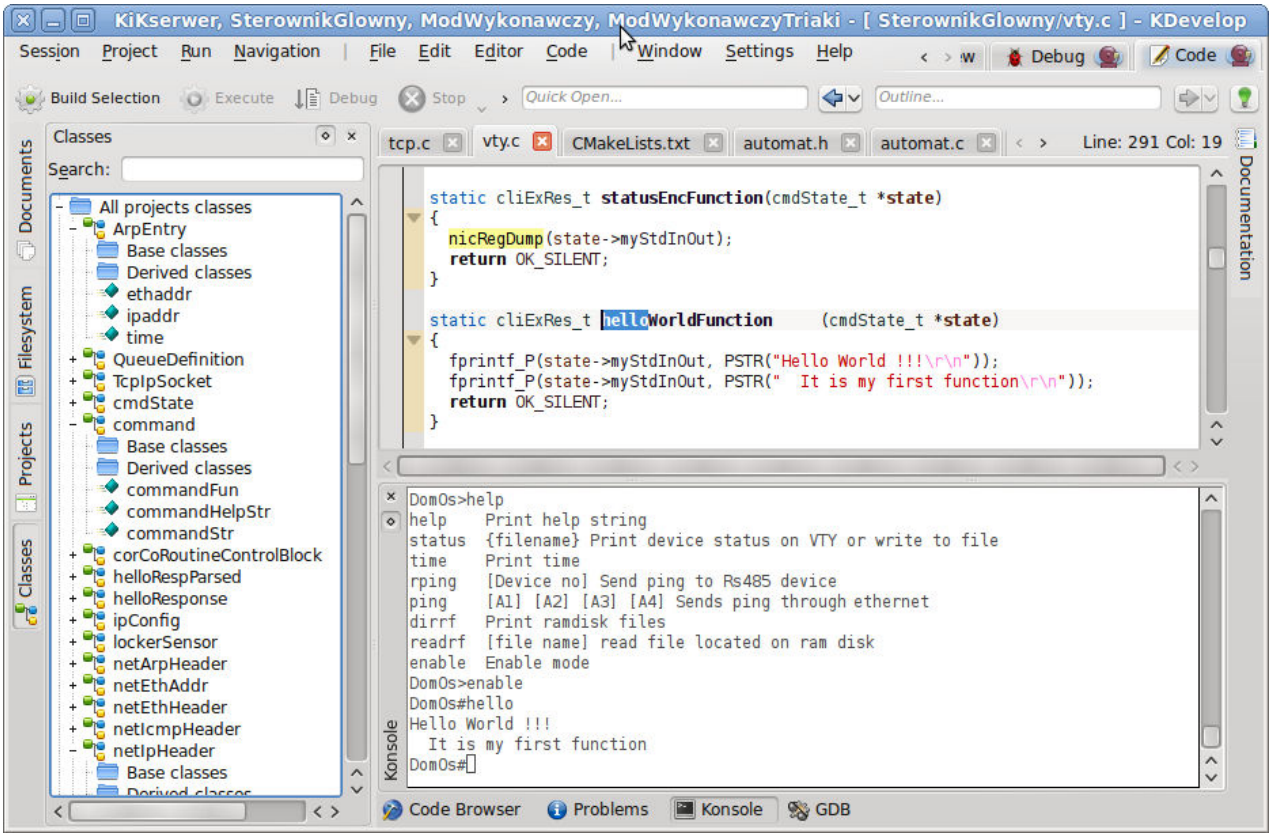

Figure 14. IDE and Hello world function

Further active participation in the set of the laboratory classes requires students to know the file system developed in the project. There are the following files in the project directory: `main.c(h)`, `cli_tasks.c(h)`, `netstack_task.c(h)`, `sensors_task.c(h)`, `hardware.c(h)`, `serial.c(h)`, `vty.c(h)`, `configuration.c(h)`. A device is initiated in the file "`main`", then tasks are created. Functions targeted by appropriate tasks are in the files: `cli_tasks.c(h)`, `netstack_task.c(h)` and `sensors_task.c(h)`. The functions of these tasks make use of the module libraries of appropriate modules. The knowledge of their implementation is not necessary to carry on with the exercises during the classes. Basic knowledge of the API is sufficient. The "`hardware`" file includes appropriate functions that handle the devices included in the evaluation set. The `configure.c` file includes functions that handle writing and reading of the configuration, e.g., the IP address and the mask. The `serial.c` file is responsible for handling serial ports. These ports send and receive data through buffers that are also used by tasks operating in the system. In the `vty.c` file, functions to be performed after an appropriate command has been written to the interpreter are defined.

The next proposed laboratory task related to the usage of the command interpreter is to add a command that controls the output of the MCP 23S17 expander connected to the microcontroller through the SPI bus. The controlling functions for the expander are in the `Lib` directory and students do not have to know its precise implementation. The implementation itself has been realized in such a way as to show some interesting aspects of the C language that are not necessarily discussed during lectures.

The library functions used during the lab classes have been prepared in such a way as to be employed in an all-purpose manner, i.e., the address of a device connected to the SPI bus has not been specified. This device (the port expander in the discussed case) is prompted with the help of the functions included in the `hardware.c` file. The file includes functions that are appropriately adapted to a specific set, where each of the devices involved is always connected to the same address line. These settings are written in the `hardware.h` file. This saves time for students as they do not have to learn the construction of the main controller in detail. Figure 3 shows that each of the modules connected to the SPI bus has a different address and, thus, its address line is connected to a separate output in a port of the microcontroller. The functions included in

the hardware.c file automatically control the available ports of the microcontroller and ensure that no more than one device is addressed at a time. Some devices are addressed by high state, others by low state. Using appropriate macros, the user does not have to know all these implementation details, which allows at the same time to maintain high efficiency in controlling the modules. A sample way of addressing the MPC23S17 port expander is presented in the following code.

```
void enableSpiMPC23S17(void)
{
#if MCP23S17_SPI_CS_EN_MASK_OR != 0
  MCP23S17_SPI_CS_PORT |=
    MCP23S17_SPI_CS_EN_MASK_OR;
#endif
#if MCP23S17_SPI_CS_EN_MASK_AND != 0xFF
  MPC23S17_SPI_CS_PORT &=
    MPC23S17_SPI_CS_EN_MASK_AND;
#endif
}
```

In a similar way, releasing the address for the same device can be implemented in the following way.

```
void disableSpiMPC23S17(void)
{
#if MCP23S17_SPI_CS_EN_MASK_OR != 0
  MCP23S17_SPI_CS_PORT &=
    (~MCP23S17_SPI_CS_EN_MASK_OR);
#endif
#if MCP23S17_SPI_CS_EN_MASK_AND != 0xFF
  MPC23S17_SPI_CS_PORT |=
    (~MPC23S17_SPI_CS_EN_MASK_AND);
#endif
}
```

The following constants have been defined in the hardware.h file: MCP23S17_SPI_CS_PORT, MCP23S17_SPI_CS_EN_MASK_OR, MCP23S17_SPI_CS_EN_MASK_AND. The first constant defines the port of the microcontroller, to which the address line that gives the expander access to the bus is connected. The constant MCP23S17_SPI_CS_EN_MASK_OR determines, which outputs of the port are to be in logical 1 state to address the device, whereas the negative constant MCP23S17_SPI_CS_EN_MASK_AND determines, which bits are to be in logical 0 state to address the device. If a system operating on the SPI bus is addressed by logical 0 state, then the constant ..._SPI_CS_EN_MASK_OR has a zero value, therefore the logical sum bit operation does not change its value. In order to avoid superfluous operations, the conditional compilation directive has been applied. Similarly, if a device is addressed in high state, then the constant ..._SPI_CS_MASK_AND has the value 0xFF and the product bit operation gives no results. To omit such an operation, conditional compilation has also been applied. When changing the way of connecting a given system to the SPI bus, it is sufficient to modify the hardware.h file. Note that some of the functions have been implemented twice in the project (e.g., function enableSpiMPC23S17): in the library files and in the directory of the project itself. Such an approach is possible when the WEAK attribute is applied. When the definition of the function reappears (without WEAK attribute), then it replaces the earlier function with the WEAK attribute. The application of the WEAK attribute is a more efficient alternative as compared to indexes to functions or virtual functions available in the C++ language.

After getting to know the expander's API, a new requirement emerges – the reading of the line number that has to be set to either high or low state. The CLI mechanism provided allows the user to read additional arguments furnished along with the command. Using the attribute argc in type cmdState_t, it is possible to read the index of the last argument. The argument with the index 0 is the name of the command, while arguments with the consecutive indexes are the parameters, with which the command is executed. In SPI, the functions cmdlineGetArgStr, cmdlineGetArgInt and cmdlineGetArgHex are given for CLI. These functions return respectively: sign chain, integer number determined on the basis of the conversion of the sign chain written in decimal format into a numerical value, and integer number determined on the basis of the sign chain written in hexadecimal format/number. Therefore, in order to write a function that sets a given line in port A in the port expander into high state, the number of that line has to be read first. Then, using the logical sum resulting from the state of port A and the bit left-shift of the number 1 by the value of the line number, sets a new state of port A of the expander. The fragment of the code below shows the function that sets the state for port A.

```
static cliExRes_t
  setPortExtAFunction(cmdState_t *state)
{
  if (state->argc < 2)
    return SYNTAX_ERROR;
  uint8_t newState =
    cmdlineGetArgInt(1, state);
  MPC23s17SetDirA(0x00, 0);
  MPC23s17SetPortA(newState, 0);
  return OK_SILENT;
}
```

The static parameter before the type returned by the function denotes that the function is available only in the file, in which it has been declared. This facilitates maintaining order in the code. The aim of the task is to add an additional function that sets a given output line into high or low state according to its number within the port. To execute this task,

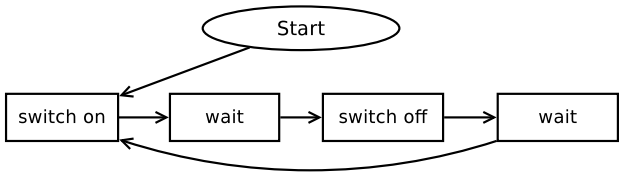

Figure 15. Flash light algorithm

the code of the function presented above can be used.

B. Coroutines FreeRTOS API

Subsequent laboratory classes introduce the issues of cooperative multitasking. Cooperative multitasking is executed with the help of coroutines, which, when used, reduce requirements in system resources. The application of coroutines is followed, however, by certain limitations. The understanding of the idea of coroutines and the specificity of their usage will be facilitated by the laboratory exercise presented below. The exercise involves the introduction of a modification to the software for the executing module so that it will be capable of controlling four light sources. The template for the project is in the directory: `Lab/Coroutines`. The first task is to control the light in such a way as to make the light flash. The algorithm for controlling a single light source is presented in Figure 15.

The algorithm is to execute the following: switching the light on, waiting for requested time, switching the light off and waiting for requested time again. Flashing of three light sources is easy to execute with the help of timers (Atmega168 microcontroller has three timers). The situation is more complicated when the number of light sources to be controlled is greater than the available counters and when each light flashes with a different frequency, independent from other light sources. In such a situation, there are two available options: a suitable program that executes the task can be written, although is not very clear or readable; or, in the most preferable solution, to make use of multitasking offered by operating systems. The realization of the algorithm that controls four light sources and uses multitasking is presented in Figure 16. Each light source is controlled in a separate thread. In the executing modules multitasking is carried out with the help of coroutines, hence each coroutine controls a separate light source, whereas the control algorithm remains the same for all sources. This means that each coroutine can perform the same function. The API of the FreeRTOS system defines for each coroutine the index to the function that is later to be performed by the coroutine. The index to the function of a coroutine has the following form: `void` vACoRoutine(`xCoRoutineHandle` xHandle, `unsigned portBASE_TYPE` uxIndex).

The first argument of the function is a handle to the coroutine. It is used by API functions and macros of the

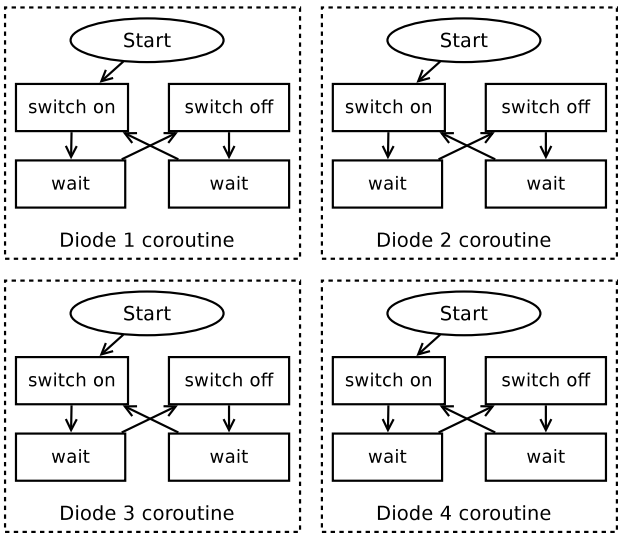

Figure 16. Algorithm for concurrent flashing of four light sources

FreeRTOS system that are executed within a function of the coroutine, e.g., to make a coroutine dormant during a given amount of time or to service a queue. The second argument is the index of the coroutine. With reference to the mentioned sample task, each light source has its own coroutine that performs the same function. Each light source has a different index for the coroutine, therefore by performing a common function, on the basis of the argument *uxIndex*, it is possible to determine the light source that the function has to control. For a coroutine to become dormant at a time t, the macro `crDELAY(`*`xCoRoutineHandle`* `xHandle,` *`portTickType`* `xTicksToDelay )` is used. The handle to the coroutine is the first argument, the second being the number of system (internal) clocks. System frequency is defined in the configuration file `FreeRTOSConfig.h`. One system clock includes many microcontroller's internal clocks. Their number is determined automatically by a special macro on the basis of the system frequency `configTICK_RATE_HZ` and the microprocessor clock frequency `configCPU_CLOCK_HZ`. FreeRTOS defines its own types that depend on the processor's architecture [35]. In the case of the Atmega processor, the type *portTickType* is a 16-bit indeterminate integer variable.

The code for the coroutine that implements the flashing algorithm can be written in the following way:

```
void vLed(xCoRoutineHandle xHandle,
      unsigned portBASE_TYPE uxIndex )
{
  // This macro is required
  crSTART( xHandle );

  for( ;; )
  {
```

```
        ledOn(uxIndex);
        crDELAY( xHandle, tLedOn[uxIndex]);
        ledOff(uxIndex);
        crDELAY( xHandle, tLedOff[uxIndex]);
    }

    // This macro is required
    crEND();
}
```

In the presented code, `tLedOn` and `tLedOff` are global tables. Each element of either of the tables denotes the glow discharge time of the diode and the idle time of the diode, respectively. The functions `ledOn` and `ledOff` are API functions of the executing module. The function, in which the coroutine is executed conventionally starts with the macro `crSTART(xHandle)` and ends with the macro `crEND`. Between the macros there is an infinite loop, in which the appropriate algorithm is executed.

Note that the low memory demand is a distinct advantage of coroutines – they operate on a common stack. Such a solution has a substantial disadvantage as well, which leads to certain limitations in their usage. After the coroutine resumes operation, the values for the local variables defined within the function executing the coroutine may be changed. Thus, if we want the variable to remain stable, such a variable has to be declared as a static or global variable. Thus revealed, this problem makes students stop and think how the compiler operates and in what way it places variables in the memory. Another limitation involves the absence of the possibility of preemption. A coroutine has to decide for itself when it is to be switched, so the only possible option here is the so-called collective multitasking. Switching of coroutines is effected at the time of the execution of blocking calls such as putting the coroutine into a dormant state for a specified time `cdDELAY` or operations on the buffer (sending or retrieving information from the buffer). All the mentioned operations can be performed only within the block of functions that service the coroutine. These cannot be executed within some other function recalled by the function that services the coroutine. In addition, the functions mentioned cannot be performed within the switch construction.

At the later stage of the envisaged laboratory classes students are asked to perform a task that involves creation of a coroutine that would allow each of the light sources (diode) to falsh with its own frequency. For the coroutine to be created the function `portBASE_TYPE xCoRoutineCreate(crCOROUTINE_CODE pxCoRoutineCode, unsigned portBASE_TYPE uxPriority, unsigned portBASE_TYPE uxIndex)` is used. The first argument is the function that the coroutine will perform, the second is the priority of the coroutine and the third argument is the coroutine index mentioned earlier in the text. The FreeRTOS text executes tasks or coroutines according to their priority. In the configuration file the number the levels for priorities is set `configMAX_CO_ROUTINE_PRIORITIES`. The higher the number is, the more operating memory is required by the system. Coroutines themselves are executed within the task or in the idle task. We do not create any tasks in the executing module, thus coroutines are executed in the idle task. This means that it is necessary to add an appropriate function that is executed in the idle task.

```
void vApplicationIdleHook( void )
{
  for( ;; )
  {
    vCoRoutineSchedule();
  }
}
```

The template for the project of the executing controller already includes the code presented above. What is necessary, however, is to create coroutines. This should be done in the main function.

```
portSHORT main( void )
{
  // Initializes hardware,
  // sets ports directions.
  hardwareInit();

  uint8_t ledNo;
  for (ledNo = 0; ledNo<4; ledNo++)
    xCoRoutineCreate(vLed, 0, ledNo);

  vTaskStartScheduler();
  return 0;
}
```

The presented main function ends with the function `vTaskStartScheduler`. Within this function, a scheduler is activated that performs all the required tasks according to their priorities. In the case of the executing module, there are no tasks, so the scheduler is always set to the idle task, in which coroutines are serviced.

The next proposed laboratory task is to expand the functionality of the software for the executing module with handling of keys. For example, pressing one of the keys will result a diode glowing for some time until the diode goes off. The state of the keys will be checked by separate coroutines. The architecture of the firmware of the executing module is presented in Figure 17.

After pressing of the key is detected, the coroutine sends a relevant message to the coroutine that services the light source. Therefore, it is necessary to introduce communication between the coroutines. This communication can be carried out using message queues. Each coroutine has its

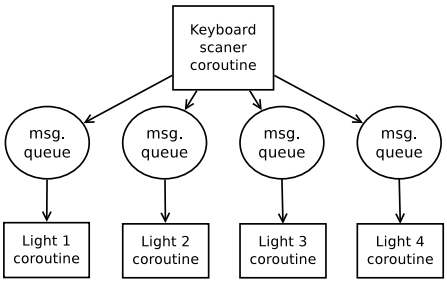

Figure 17. Architecture of executing module firmware controlling four light sources with keyboard

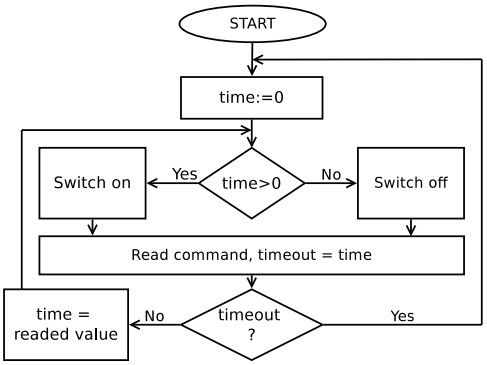

Figure 18. Algorithm for coroutine handling a single light source

own message queue, from which it receives messages. The information in the message includes information on 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 that controls a single light source is shown in Figure 18. Initially – during the initialization phase – the light is switched off, therefore the variable *time* is equal to zero (analogous to message format). In the next step, the coroutine checks the value of the *time* variable. If it is greater 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 time-out 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.

FreeRTOS provides a special API for handling semaphores. To read messages the macro `void crQUEUE_RECEIVE(xCoRoutineHandle xHandle, xQueueHandle pxQueue, void *pvBuffer, portTickType xTicksToWait, portBASE_TYPE *pxResult)` is used. The macro allows the determination of the maximum time-out for a message with the help of the fourth argument. After invoking the macro, the coroutine is suspended for a specified time (optionally specified time-out) or until the

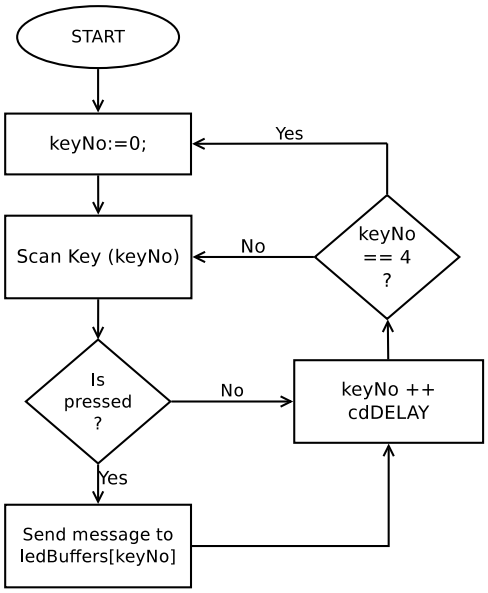

Figure 19. Algorithm for coroutine handling keyboard

message is received. The macro `crQUEUE_RECEIVE` is then capable of replacing the macro `crDELAY`.

The macro `crQUEUE_RECEIVE` requires additional arguments such as a handle to the coroutine (the first argument) and a handle to the queue, from which it will be reading a new message (the second argument). The third argument is the index to the memory, to which the received message will be written. The fourth argument defines the time dedicated for an operation to be performed, whereas the fifth argument is the index to the variable of the type *portBASE_TYPE*. The variable pointed to by pxResult will be set to pdPASS if data has been successfully retrieved from the queue, otherwise it will be set to an error code as defined within ProjDefs.h. Hence, if the variable has the value *pdPASS*, then the light source will be switched off. The light source can be switched off prior to the completion of the specified time-out when a successive message with the time value set to 0 is sent to the queue.

The keyboard coroutine, after detecting pressing of a key, sends a message to an appropriate coroutine through a message queue. For this purpose, the macro `void crQUEUE_SEND(xCoRoutineHandle xHandle, xQueueHandle pxQueue, void * pvItemToQueue, portTickType xTicksToWait, portBASE_TYPE * pxResult)` has to be used. The importance of individual arguments is the same as in the case of the macro `crQUEUE_RECEIVE`.

The algorithm for handling keyboard events is presented in Figure 19. The algorithm provides an opportunity to check successively the state of each of the keys. When pressing of a key is detected, then a message is sent to an appropriate

message queue through its handle. Handles to message queues are stored in the global table *ledBuffers*. Each element of the table corresponds to a different coroutine that handles a separate light source. To check the state of a key the function *uint8_t* readKey(*uint8_t* keyNo) is used. This function returns zero, when the key is pressed down, and a non-zero result when the key is depressed. The argument keyNo defines the number of the key that is being checked.

To ensure appropriate handling of messages related to keyboard handling, it is necessary to create buffers in the function main with the function *xQueueHandle* xQueueCreate(*unsigned portBASE_TYPE* uxQueueLength, *unsigned portBASE_TYPE* uxItemSize) provided by API of the FreeRTOS system. The first argument determines the length of a single message, whereas the number of messages that the buffer can accommodate is defined by the second argument. It should be noted that in the case of the AVR architecture the type *portBASE_TYPE* is a 8-bit variable and, hence, the length of a queue and the length of a single message cannot exceed 255 bytes. It is also necessary to create in the function main a coroutine that will be responsible for checking the state of the keyboard.

The last task to be performed by students during their classes devoted to handling of coroutines is to secure communication between the executing module and the rest of the system using the RS485 bus. To achieve this, another coroutine that is responsible for handling of the communication protocol has to be added. In line with the architecture shown in Figure 20, the coroutine handling the communication protocol makes use of the queues RS 485 RX and RS 485 TX that, respectively, send and receive data onto and from the RS 485 bus. Each byte that is received on the bus is placed as an individual (separate) message in the RS 485 RX buffer. Similarly, each byte that has to be sent onto the bus is placed by the coroutine handling the communication protocol in the RS 485 TX bus. The coroutine that handles the communication protocol receives successive bytes from the queue RS 485 RX and consolidates them in a message frame. Then, the coroutine checks whether the message address corresponds to the address in the executing module and, with the help of the control code CRC16, whether the message is not a malfunction message. If the received message includes the switch on the light source command or the switch off command, then the coroutine sends a message to an appropriate message queue that services the coroutine handling light source data. The coroutine handling the communication protocol is more complex. Students get its ready-made implementation. The task they are expected to perform is to create queues RS 485 TX and RX, as well as the coroutine handling the communication protocol.

Messages are sent to the RS 485 RX buffer by the function handling Receive Complete interrupt that appears after a byte is received by the serial port. Functions handling interrupts have a spacial API provided by the FreeRTOS system. The macro crQUEUE_SEND_FROM_ISR is designed to send messages to the program buffer. Similarly, messages from the RS 485 TX buffer are read by the function handling the interrupt "Data Register Empty" that determines whether the sending buffer of the serial port can receive another sign to be successively sent on. For this purpose, the macro crQUEUE_RECEIVE_FROM_ISR is used. If the RS 485 TX buffer is empty, then the interrupt "Data register Empty" is switched off. Switching the interrupt off is activated by the macro vInterruptOff() and is effected automatically within this interrupt handling. If the RS 485 TX buffer included any data to be sent, then, before placing them in the hardware sending buffer of the serial port, the MAX481 transmitter is switched on. The transmitter enables sending data onto the RS 485 bus that operates in the half duplex mode. The transmitter is switched off within "Transmit Complete" interrupt handling. The interrupt occurs only when the transmitter has sent all the content of its hardware buffer.

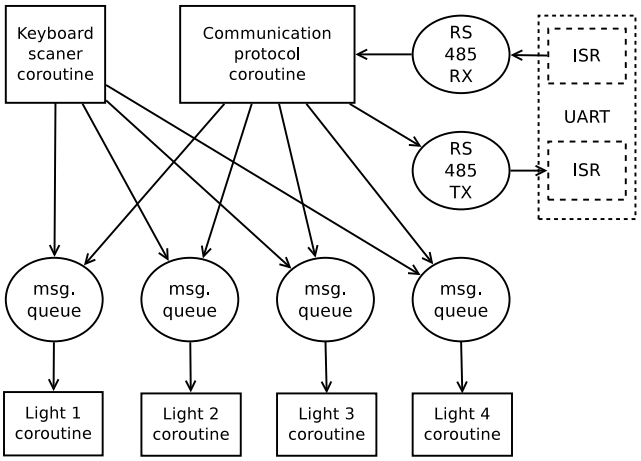

Figure 20. Architecture of executing module firmware controlling four light sources

After adding a new sign to the RS 485 TX sending buffer, interrupt handling "Data register Empty" should be switched on with the macro vInterruptOn(). The programmer must not forget about it. Such an approach has been adopted purposefully having in mind that the RS 485 bus should be blocked for as short a time as possible. A message should be sent only when it formed in full and after being placed in the buffer. Hence, after placing the whole of the message in the buffer, it is necessary to switch the interrupt "Data Register Empty" on. The function handling this interrupt will then switch on the transmitter and will send the message via the bus. On termination of the transmission, the transmitter will be automatically switched off (without the interference of the programmer).

VI. Conclusion and Future Work

The presented didactic system is a valuable addition to the theory of operating and embedded systems. It enables students to get familiarized with such aspects as 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 students can afford to build the presented platform and use it for didactic or practical purposes limited only by their imagination.

The article puts special attention to a detailed presentation of some selected laboratory exercises prepared for laboratory classes. These exercises familiarize students with practical aspects of issues related to the theory of operating and embedded systems. The presented exercises facilitate successful understanding of techniques of implementing multi-thread applications and the creation of commands and handling of file systems.

The presented didactic platform is still being developed. The future work is related to the implementation of SD card, as well as IPv6 or TCP protocols.

Simultaneously, the achieved platform's simplicity introduces some limitations, mainly concerning the size of random access memory, which cannot be extended. The limited size of random access memory hinders an implementation of the SSH protocol that supports the encrypted connections.

Acknowledgement

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

References

[1] A. Kaliszan and M. Głąbowski, "Didactic embedded platform and software tools for developing real time operating system," in *Proceedings of the The Seventh Advanced International Conference on Telecommunications (AICT 2011)*, M. Głąbowski and D. K. Mynbaev, Eds. St. Maarten, The Netherlands Antilles: IARIA, Mar. 2011, pp. 77–82.

[2] R. Love, *Linux Kernel Development*, 3rd ed. Novell Press, Jul. 2010.

[3] "Microsoft Windows Server." [Online]. Available: http://www.microsoft.com/windowsserver2008/en/us/default.aspx <retrieved: Jan, 2012>

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

[5] J. M. Corchado, J. C. Augusto, and P. Novais, *Ambient Intelligence and Future Trends*, 1st ed., ser. Advances in Intelligent and Soft Computing. Springer, 2010, vol. 72. [Online]. Available: http://www.springer.com/engineering/computational+intelligence+and+complexity/book/978-3-642-13267-4 <retrieved: Jan, 2012>

[6] L. Sydell, "Chasing a habitable 'home of the future'," May 2006. [Online]. Available: http://www.npr.org/templates/story/story.php?storyId=5360871 <retrieved: Jan, 2012>

[7] "Mach homepage." [Online]. Available: http://www.cs.cmu.edu/afs/cs.cmu.edu/project/mach/public/www/overview.html <retrieved: Jan, 2012>

[8] A. Tevanian, Jr., R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper, and M. W. Young, "Mach threads and the unix kernel: The battle for control," in *in Proceedings of the USENIX Summer Conference, USENIX Association*, 1987, pp. 185–197.

[9] A. Singh, *A Technical History of Apple's Operating Systems*. osxbook.com, 2001.

[10] R. Stallman, "The GNU manifesto," *Dr. Dobb's Journal*, vol. 10, no. 3, p. 30, Mar. 1985.

[11] ——, "GNU manifesto." [Online]. Available: http://www.gnu.org/gnu/manifesto.html <retrieved: Jan, 2012>

[12] "100 of the most significant events in linux history," *Linux Journal*, Aug. 2001. [Online]. Available: http://www.linuxjournal.com/article/6000 <retrieved: Jan, 2012>

[13] "FreeBSD homepage." [Online]. Available: http://www.freebsd.org <retrieved: Jan, 2012>

[14] "Ethernut homepage." [Online]. Available: http://www.ethernut.de <retrieved: Jan, 2012>

[15] "Arduino homepage." [Online]. Available: http://arduino.cc <retrieved: Jan, 2012>

[16] "Arduino programming language." [Online]. Available: http://arduino.cc/en/Reference/HomePage <retrieved: Jan, 2012>

[17] "Wiring homepage." [Online]. Available: http://wiring.org.co <retrieved: Jan, 2012>

[18] "The FreeRTOS project homepage." [Online]. Available: http://www.freertos.org <retrieved: Jan, 2012>

[19] FreeRTOS, "Copyright notice," http://www.freertos.org/copyright.html <retrieved: Jan, 2012>

[20] "Eagle." [Online]. Available: http://www.cadsoftusa.com/ <retrieved: Jan, 2012>

[21] A. Kaliszan, "AtMega128 RTOS hardware repository." [Online]. Available: http://rtosOnAvr.yum.pl/hardware/ssw <retrieved: Jan, 2012>

[22] Atmel, "Atmega128 data sheet." [Online]. Available: http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf <retrieved: Jan, 2012>

[23] ——, "Atmega168 data sheet." [Online]. Available: http://www.atmel.com/dyn/resources/prod_documents/doc2545.pdf <retrieved: Jan, 2012>

[24] FTDI, "FT232RL data sheet." [Online]. Available: www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT232R.pdf <retrieved: Jan, 2012>

[25] Maxim, "MAX481 data sheet." [Online]. Available: http://datasheets.maxim-ic.com/en/ds/MAX1487-MAX491.pdf <retrieved: Jan, 2012>

[26] Microchip, "Enc28j60 data sheet." [Online]. Available: ww1.microchip.com/downloads/en/devicedoc/39662a.pdf <retrieved: Jan, 2012>

[27] ——, "MPC23S17 data sheet." [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/21952b.pdf <retrieved: Jan, 2012>

[28] Texas Intruments, "ULN2003A data sheet." [Online]. Available: http://focus.ti.com/lit/ds/symlink/uln2003a.pdf <retrieved: Jan, 2012>

[29] Microchip, "MCP3008 data sheet." [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/21295d.pdf <retrieved: Jan, 2012>

[30] National, "LM35 data sheet." [Online]. Available: http://www.national.com/ds/LM/LM35.pdf <retrieved: Jan, 2012>

[31] Maxim, "DS1305 data sheet." [Online]. Available: http://datasheets.maxim-ic.com/en/ds/DS1305.pdf <retrieved: Jan, 2012>

[32] G. Socher, "AvrUsb500v2 – an open source Atmel AVR programmer, stk500 v2 compatible, with USB interface." [Online]. Available: http://tuxgraphics.org/electronics/200705/article07052.shtml <retrieved: Jan, 2012>

[33] "AVR JTAG ICE clone." [Online]. Available: http://www.scienceprog.com/build-your-own-avr-jtagice-clone <retrieved: Jan, 2012>

[34] A. Kaliszan, "AtMega128 RTOS firmware repository." [Online]. Available: http://rtosOnAvr.yum.pl/software/FreeRtos <retrieved: Jan, 2012>

[35] FreeRTOS, "FreeRTOS API reference." [Online]. Available: http://www.freertos.org/a00106.html <retrieved: Jan, 2012>

[36] P. Stang, "Procyon AVRlib API," 2006. [Online]. Available: http://www.procyonengineering.com/embedded/avr/avrlib/ <retrieved: Jan, 2012>

[37] "AVR-libc API," 2006. [Online]. Available: http://avr-libc.nongnu.org <retrieved: Jan, 2012>

[38] G. Socher, "HTTP/TCP with an Atmega88 microcontroller (AVR web server)," 2006. [Online]. Available: http://www.tuxgraphics.org/electronics/200611/embedded-webserver.shtml <retrieved: Jan, 2012>

[39] "GNU automake." [Online]. Available: http://www.gnu.org/software/automake <retrieved: Jan, 2012>

[40] "Cross platform make." [Online]. Available: http://www.cmake.org <retrieved: Jan, 2012>

[41] "GNU make." [Online]. Available: http://www.gnu.org/software/make <retrieved: Jan, 2012>