Formal Description of Embedded Operating Systems

Martin Vojtko*
Institute of Computer Engineering and Applied Informatics
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava
Ilkovičova 2, 842 16 Bratislava, Slovakia
martin.vojtko@stuba.sk

Abstract

The fast development of new processors introduces problems with the adaptation of operating systems. When a new processor is presented on the market, the operating system needs to be adapted to the processor architecture and features. It is done by the reprogramming of a platform-dependent layer and the implementation of missing device modules of the operating system. The adaptation process of the operating system is more complicated when the new processor has a completely different architecture than the one of the operating system for which it was previously designed for. Another problem of the adaptation is in the processor datasheets, because they are not processable by the computer so the generation of the operating system code from datasheets is not possible. In this dissertation thesis, we present an updated adaptation process of embedded operating systems. We designed a Processor Formal Description that acts as a computer processable datasheet. This description is used for automated code generation of platform-dependent code. As a support to the adaptation process we present a concept of an adaptation framework that helps to reduce time needed for the adaptation of the operating system.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General—*hardware/software interfaces*; C.3 [**Computer Systems Organization**]: Special-purpose and Application-based Systems—*microprocessor/microcomputer applications, real-time and embedded systems*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*modules and interfaces*; D.3.4 [**Programming languages**]: Processors—*code generation*; D.4.7 [**Operating Systems**]: Organization and Design—*real-time systems and embedded systems, standardization*

*Recommended by thesis supervisor: Assoc. Prof. Tibor Krajčovič
To be defended at Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava, 2016.

Keywords

Processor Formal Description, Adaptation of Operating Systems, Code Generation, Adaptation Process, Modelling of Modules of Operating Systems, Modular Operating Systems, Layered Operating Systems, Embedded Operating Systems

1. Introduction

The growing number of processor architectures leads to the need for a methodology which allows fast and effective operating system adaptation to those architectures. Future embedded systems will have multi-core/many-core architectures [7] or mixed architectures consisting of multi-core processor clusters. New types of architectures will introduce new types of operating systems which will be self-adaptive [9]. New operating systems running in a heterogeneous environment will need a database of existing processor ports, device modules and processing cores. Modules and platform ports will be loaded to program memories of the processor during system initialization or will be loaded on-line during the system run-time.

Many-core systems are changing the traditional concept of the processor as a system with several devices and a few processing cores. The number of cores will grow in the future together with the number of intelligent devices that will be connected to a shared network [7]. This future highly scalable network architecture calls for a change of the standard architecture of operating systems into a distributed architecture.

Recent operating systems are seen as a software that interfaces and extends the processor. In the future it will be more than that. The operating system will be seen as a framework or a database of modules and platform ports. Consequently, the developer will choose modules and platform ports from the database that fits architecture of the processor. The operating system framework will also provide tools that help the developer to create modules and platform ports that are missing. Operating systems, like FreeRTOS [8] and many more, started this transformation but it is only a beginning and many aspects of the operating system will change in the future.

In this paper we analyse a generalized form of the adaptation process that is used during adaptation of any operating system nowadays. This process does not support an automated code generation that will be crucial in the future. We propose an extension to the process in order to add formalization techniques to the processor description. This extension allows the generation of platform-

dependent parts of the operating system. As a result of this formalization we specify the Processor Formal Description (PFD) in this paper. The PFD describes each device and processing core of the processor in a form that is processable by the computer. From the PFD we generate a glue code, which interfaces the processor and the operating system. Finally we use this glue code for the implementation of modules of the operating system.

We also propose a framework that will support the adaptation process of embedded operating systems. The framework consists of tools and services that help to describe the processor [10], generate glue code [11], describe operating system modules that encapsulate devices and processing cores of the processor [12], and implement those modules [12].

2. Adaptation Process of OS

The adaptation process of the operating system (OS) is mostly started because there is the need to use a specific feature of the OS or there is the need to port specific OS to an architecture that is in some way special or solves the specified problem. The experience of the developer with the OS plays a big role in this need. The adaptation of the OSE OS to the many-core architecture Tilepro64 is a good example [2]. The mapping of the OSE scheduler was done on the mentioned many-core architecture. The authors provide information about the steps of the adaptation of the operating system but the defined process of the adaptation is strongly application specific.

Well known operating systems, such as FreeRTOS [8], Avrx [3] or TinyOS [6], provide adaptation manuals to the developer. Those manuals explain which parts of the OS should be adapted during the adaptation to the processor. In the FreeRTOS example, there exists a vast amount of minimal working examples (MWEs) and adaptations to many existing platforms. This sort of database of examples increases the popularity of this OS. But what will happen when there is no existing example for the processor that the developer wants to use is that the developer will have to implement the support for it. After the adaptation the developer should provide the solution to the FreeRTOS community.

The FreeRTOS community has no strict rules for the platform ports so they can differ from port to port. The mostly affected part of the OS during the adaptation is a platform-dependent part that interfaces OS modules to the hardware. When each developer implements this layer differently the code between platform ports is not manageable.

Another aspect of the adaptation is a missing standard that will induce manufacturers of processors to provide datasheets in a standardized form. Each manufacturer has its own templates. Another problem is that those data-sheets were prepared for human so any computer processing is nearly impossible. Our idea is to propose such a standard that will describe the processor in a form that will be processable by a computer.

Processor manufacturers provide also many MWEs for their platforms. Many of the manufacturers implement their own header files, source files and glue code. This code is helpful when you use the OS on the processors issued by one manufacturer (sometimes from one family only). This code also differs between manufacturers.

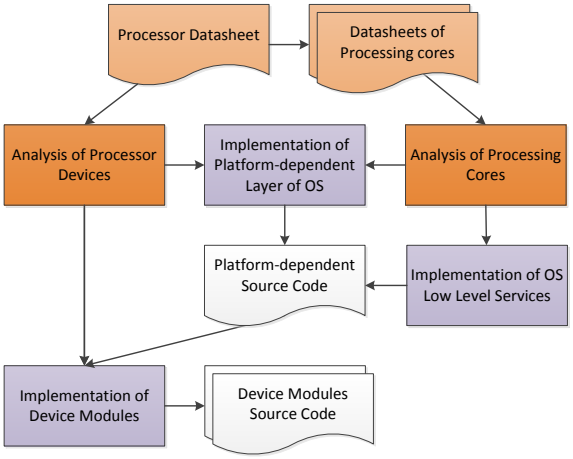

Figure 1: Generalized process of adaptation of an operating system [12].

The Figure 1 shows the generalised adaptation process that can be divided into two workflows. The first workflow shows an analysis of processing cores of the processor, a design of OS modules and an implementation of a code that uses features of the processor. The second workflow shows an analysis of existing processor devices, a design and an implementation of a code that manages processing cores of the processor.

The first step of the adaptation of the OS to the processor is an analysis. As we mentioned previously the analysis can be split into two parts where devices and processing cores are analysed separately. In this step the designer analyses all the materials that are provided by the manufacturer of the processor. Mostly it is in the form of a datasheet of the processor or datasheets of the processing cores of the processor.

2.1 Processing Cores

During the adaptation of processing cores the designer has to find out how main services of the OS can be implemented. The services are:

- core and OS initializing,
- interrupt handling and
- task switching.

During core and OS initialization all the operating modes of the processing core, and stacks and memories of the OS kernel are set. Most of the manufacturers provide MWEs for the core initialization but they have to be adjusted to the needs of the OS.

The interrupt handling is the service of the OS that is partially implemented in assembly language. Most of the processors provide an interrupt subsystem that can be used by the OS. The developer has to implement an interface to this subsystem and after that he can start implementing interrupt routines that are mapped to a specific interrupt source, as can be the task switch.

The task switch is crucial for any OS because it handles the correct storing of the old task and loading of the new task. During the task switch each register of the processing core has to be stored before a task can be replaced by another.

All previously mentioned services are highly platform-dependent so in most cases those services have to be implemented during almost every adaptation of the OS.

2.2 Devices

During the adaptation of devices the designer chooses devices that will be needed for the successful completion of the task.

The developer analyses the functionality and the communication interface of each device. The interface mostly consists of registers and signals by which the OS can send tasks.

The designer uses the registers of the device for the implementation of the glue code that acts as an interface that the OS can understand. The interface consists of simple read/write routines that access device registers. The designer can use this interface during the implementation of device modules of the OS. Many manufacturers implement their own glue code for their processors. This is very helpful because the developer can concentrate on the OS design. The problem is that this glue code differs between manufacturers.

2.3 Generation of the Glue Code

In the past, there were projects that tried to generate glue code for hardware. The glue code was mostly meant as a code that was needed to connect two hardware components with different interfaces [4]. Sometimes the resulting glue contained even new pieces of hardware that acted as a translator of communication [13] [5]. Those techniques were used for a connection of the processing core to the device through a set of separate signal lines. Nowadays the majority of devices is connected to the processing core via standardized interfaces, e.g. internal bus. Also the interfaces of hardware are standardized [1], so the complexity of the hardware interconnection is reduced.

3. Proposal of Novel Adaptation Process

The novel adaptation process was designed to help the developer of the embedded OS to generate the platform-dependent code of the OS for any chosen processor. Generation of code reduces the adaptation time and speeds up the preparation of working prototypes. The process also helps during the modelling and implementation of the modules of the OS. Those modules manage the processor devices and processing cores. The process described in the Figure 2 is suitable for embedded OSs that have a layered architecture that consists of at least one platform-dependent and one platform-independent layer [12].

The proposed process applies formalization techniques that allow the generation of the mayor part of the platform-dependent layer of the OS. The platform-dependent layer consists of simple routines that are applied above registers of devices and processing cores. Those routines are stateless and perform just one operation at a time. Together they create an interface that consists of many simple and similar routines that can be produced by automatic generation of the code.

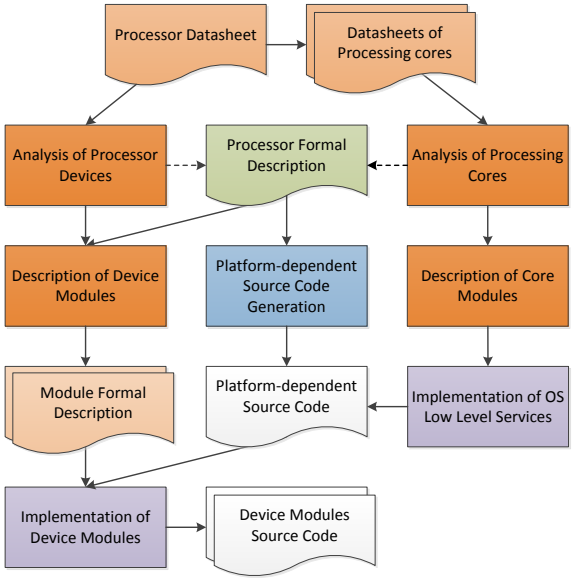

Figure 2: New proposal for the adaptation process of the embedded operating system [12].

The advantages of generated code are:

- fast prototyping,
- reduction of error probability,
- hiding of hardware complexity,
- consistent and similar result across most of architectures and
- the developer can concentrate on the application domain.

3.1 Inputs of the Adaptation Process

As the input for the novel adaptation process the developer needs the following documents [12]:

- Processor datasheet - provides information about processor devices;
- Processing cores datasheet - provides information about the processing cores of the processor;
- Processor description file - represents a computer-readable form of the processor datasheet.

The Processor Formal Description (PFD) is a new document introduced in the adaptation process. It is the result of the processor analysis. Currently, the preparation of PFD has to be done by the developer but in the future it could be provided by the manufacturer of the processor. The PFD stores information about each item of the processor that can be affected by an instruction from the instruction set of the processor. More information can be found in the section 4.

3.2 Description of Devices and Cores

The new process formalizes most of the aspects of the OS adaptation so there is no reference to any programming language until the implementation phase. This is

different in the old adaptation process where the description language is mostly the same as the implementation language. Programming languages have often limitations that are impacting also the design of modules. One of the limitations is that the programming language (in embedded systems it is mostly C) has poor ability to model parallel execution of tasks.

In the description of devices or cores there is no need for such parallel design but the independence of the description from the programming language can help to express aspects of devices or cores that can not be expressed by a programming language (e.g. connections between devices).

The glue code is a part of the code that has to be implemented but from the perspective of the developer it has no added value to the functionality of the OS. It just interfaces the hardware to higher levels of the OS. The nature of the glue code is simplicity that provides good space for a code generation.

The glue code is generated from the description of devices and cores (so from PFD). The device is accessed by writing to its registers or reading from them. Those simple operations can be fully covered by the generator of the glue code. The processing core is more complicated than the device so only a part of its description can be used for the generation of the code.

3.3 Description of OS modules

The PFD is also used during the design of the OS modules. The described processor parts are "named items" that can be used during the modelling of module behaviour (e.g. as the description in a flow chart diagram).

In the past we proposed the Module Formal Description that is based on workflow diagrams. In that case the description uses parts of the PFD as building blocks that can model whole behaviour of the OS module. As a consequence the model can be easily converted to programming language by another code generation that can create skeletons of whole OS module [12].

4. Formal Description of the Processor

The formal description is needed to allow the automatic generation of the platform-dependent code. The PFD describes a processor from the top to the bottom starting from processor devices and processing cores [10]. A whole mathematical model was implemented to cover any part of the processor that can be affected by an instruction from the instruction set of the processor, but there is no space to cover this model in this paper. The model is fully described in Vojtko et al. [12].

The Figure 3 shows the visualization of the PFD where processor is the center of the model. Black arrows represent "consists of" relationship and red dashed arrows represent "depends on" relationship. So we can say that processor consists of devices and processing cores. A device consists of registers, I/O signals and interrupt signals. A processing core consists of registers, I/O signals, instructions and operating modes. Any register consists of register parts and register parts can consist of options. So the model of processor is a 5 level hierarchy of items.

From the perspective of the code generation the most important parts of the PFD are levels 3, 4 and 5. From those levels the platform-dependent layer of the OS is generated. Levels 1 and 2 have their importance in organization of code into logical modules (e.g. devices and processing cores). Those two levels are helpful during the modelling of OS device modules.

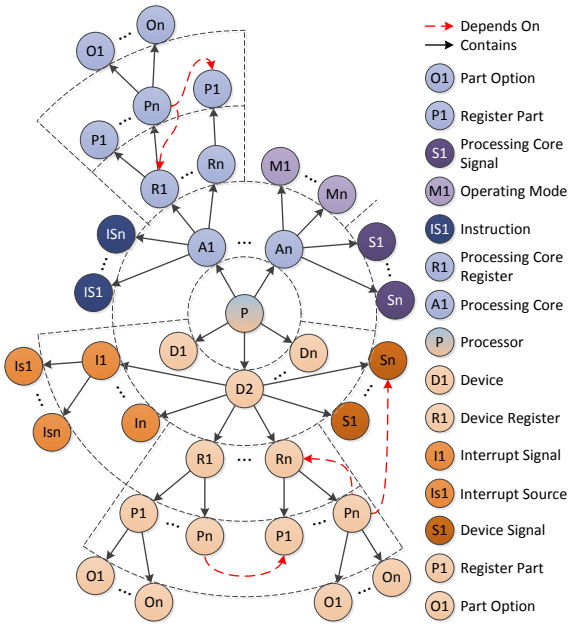

Figure 3: Visualization of the PFD items [12].

The "depends on" relationship reflects a dependence that can exist between items of third and fourth level (i.e. register, signal and register part) of the PFD. There is a dependence between two items when the change in one item triggers a change in another. A good example of dependence is the reset of the interrupt register that was accessed by a read operation. When you try to read this register you also start a sequence of events that resets the register parts to their default values. The coverage of dependencies in the PFD is extremely helpful during the implementation of OS modules because the description of dependencies informs the developer that he should be vigilant when working with dependent registers so he implements the module keeping those dependencies in mind.

In greater detail the PFD describes the communication interface of devices and processing cores. This means that the internal structure of hardware modules is hidden. This hiding of hardware structure is an advantage compared to other examples of descriptions as is VHDL

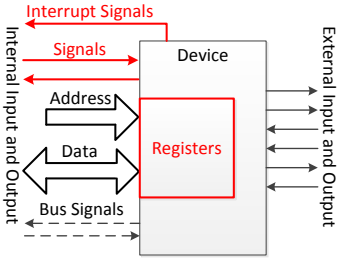

Figure 4: Communication interface of device [12].

or Verilog, because manufacturers do not want to publish their hardware architecture. The Figure 4 shows the input and output signals of the device and the registers of the device. The signals and registers marked by red color form the communication interface that is modelled by the PFD. Also other signals (as are bus signals) exist in the device but from the perspective of the PFD those signals are not directly accessible by the processor instructions.

5. Formal Description of OS Modules

The module of the OS manages and controls the processor device. It uses platform-dependent code prepared by the glue code generator. The formal description can be used during the design of OS modules, because it simplifies the adaptation process. In Vojtko et al. [12] such a formal description was proposed that uses existing register parts and registers described in the PFD as blocks of a workflow diagram (e.g. the Figure 5).

The module of the OS can be divided into 3 parts which are modelled independently[12]:

- Module initialization,
- Interrupt handling and
- Data processing.

The module initialization models the process of the device setup. The interrupt handling models the process of interrupt source selection and appropriate interrupt response routine. The data processing models the ways and means of data preparation, transferring and receiving.

The Figure 5 shows how an initialization of universal serial interface can be modelled. In the Figure there are two

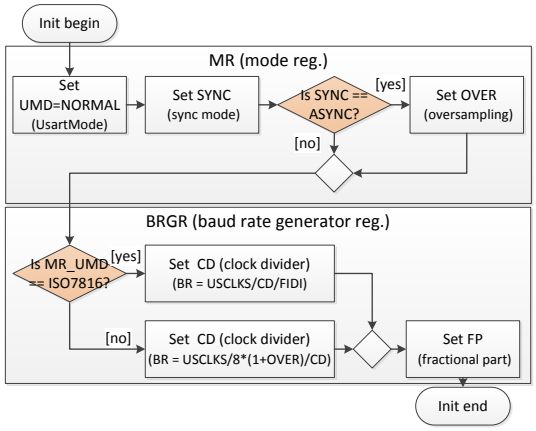

Figure 5: Init function of USART (diagram) [12].

```
void init(int SYNC, int OVER, int CD, int FP){
  int dataset = 0x0;
  dataset = set_USART_MR_UMD(0x0, USART_MR_UMD_NORMAL);
  dataset = set_USART_MR_SYNC(dataset, SYNC);
  dataset = set_USART_MR_OVER(dataset, OVER);
  write_USART_MR(dataset);

  dataset = set_USART_BRGR_CD(0x0, CD);
  dataset = set_USART_BRGR_FP(dataset, FP);
  write_USART_BRGR(dataset);
}
```

Figure 6: Init function of USART (code) [12].

envelopes (MR and BRGR) that represent two registers of the universal serial interface. Those registers contain parts that are set to a value specified by the option name. The diagram allows to model the dependence between parts of the register. In this example the setup of oversampling (OVER) will have effect only if synchronization (SYNC) of the serial interface is set to asynchronous mode (ASYNC).

From this diagram a code can be generated as is shown in the Figure 6. The generated function uses four parameters that are used for those blocks in the model that were not set right in the diagram. As the figure shows there is no condition generated for SYNC as was used in the diagram. This is because the diagram informs that the set of OVER will have no effect to behaviour of USART when SYNC is not set to ASYNC value. As can be seen in the code there is the variable *dataset* that is set to the needed value and then this variable is written to the mode register. All values of the register parts are written to the *dataset* so only internal registers of the processor are used until the write operation to the device register is done.

6. Framework for OS adaptation

The concept of the OS adaptation framework is based on the adaptation process. This framework will support the adaptation process by a set of services and databases. The Figure 7 represents the conceptual architecture of the framework.

The framework will positively impact these tasks:

- PFD creation and validation,
- platform dependent code generation,
- OS module modelling and validation,
- OS module mapping to the PFD,
- code implementation and/or generation, and
- selection of platforms and modules.

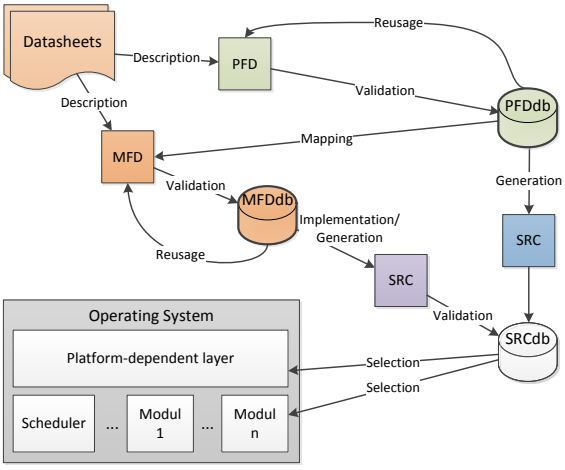

Figure 7: The Adaptation framework services. (MFD - module formal description, db - database, SRC - source code)

The framework will use 3 databases to store OS environment:

- database of PFDs,
- database of OS modules, and
- database of OS source codes.

6.1 Description of the Processor

The framework allows preparation of the PFD from the processor datasheets. The prepared PFD will by validated and sent to the database of PFDs. The stored description can be then used by the generator to produce the platform-dependent code of the OS. PFDs are also used for the mapping of the OS modules to communication interfaces of the devices and processing cores.

Stored PFDs will be decomposed into devices and cores. Each identified device will be inserted to a database under validation, which will guarantee that device is not presented in the database as a duplicity.

6.2 Description of the Module

Since the PFD covers only the communication interface of the processor device or core, there is still need for the development of the OS module that manages this device or core. Workflow diagrams will be used for mapping of the OS module to the communication interface. The resulting model of an OS module will be validated and then inserted to the database of Module Formal Descriptions (MFDs). The documentation of a model will be a compulsory part of the module description.

6.3 Module Code Generation/Implementation

The developer implements the OS module code from the MFD . Some parts of module can be generated automatically as a skeleton of the module which will help to the developer during implementation. Implemented module is then stored in a database of the OS source codes with linkage to parent module description and compulsory documentation.

6.4 Selection of the OS Parts

As a part of a system design, the developer of embedded system will have access to database of the OS source codes. From this database the developer will choose a platform-dependent layer and select a compatible device and processing core modules for the chosen processor. He can also add/describe/implement missing modules.

6.5 Databases

Full PFDs will be stored in the database of descriptions. Those files will be also decomposed into separate devices and processing cores. A problem can arise when the same device exists in more processors so this situation has to be solved by a unique identification of the device. In order to avoid a duplicate upload of the existing PFD it is necessary to create a protection mechanism. If existing processor was revised by the manufacturer it will be possible to revise the PFD too.

The database can be used also during the creation of a new PFD file where the developer can search for devices and cores of the existing PFDs in the database and include them into the new PFD, which will reduce duplicity and description time.

The Database of MFDs will store descriptions of OS devices and processing core modules. Similarly as the database of PFDs this database will also use unique identification of inserted MFDs. MFDs from the database can be used for describing similar device modules as MWEs. Existing modules can be reused in the module description, which will reduce description time.

Unique versions and ports of OS source code will be stored in the database of sources. The developer will be able to select platform-dependent code for the selected processor and he will be able to select source codes of modules based on the description of a module, because there can be presented more versions of the module.

7. Conclusions

The concept of the adaptation framework for embedded operating systems was presented in this paper. The framework will provide services for the developer of the embedded operating system. These services will help during the adaptation of the operating system to new processors. The adaptation time will be shorter and adaptation complexity simpler. Until now, the formal description of the processor and the generator of the platform dependent code was developed. The generator generates the platform-dependent code in programming language C. The next step in the work is the design of a module description tool that allows describing operating system modules.

Acknowledgements. This work was supported by the Ministry of Education, Science, Research and Sport of the Slovak Republic within the Research and Development Operational Program for the project: "University Science Park of STU Bratislava", ITMS 26240220084, co-funded by the European Regional Development Fund.

References

[1] Accellera Systems Iniciative Inc. *Open Core Protocol Specification*, 2013.

[2] V. Avula. Adapting operating systems to embedded manycores: Scheduling and inter-process communication. Master's thesis, Uppsala universitet, 2014.

[3] L. Barello. *AvrX Real Time Kernel*, 2007. http://www.barello.net/avrx/.

[4] P. Chou, R. Ortega, and G. Borriello. Synthesis of the hardware/software interface in microcontroller-based systems. In *Computer-Aided Design, 1992. ICCAD-92. Digest of Technical Papers., 1992 IEEE/ACM International Conference on*, pages 488–495, Nov 1992.

[5] Z. Guo, A. Mitra, and W. Najjar. Automation of ip core interface generation for reconfigurable computing. In *Int. Conference on Field Programmable Logic and Applications (FPL 2006), Madrid, Spain,*, page 6, Aug 2006.

[6] P. Levis and D. Gay. *TinyOS Programming*. Cambridge University Press, 2009.

[7] P. Ranganathan. From microprocessors to nanostores: Rethinking data-centric systems. *Computer*, 44(1):39–48, Jan 2011.

[8] Real Time Engineers Ltd. *The FreeRTOS Project*, 2015. http://www.freertos.org/.

[9] M. Seltzer and C. Small. Self-monitoring and self-adapting operating systems. In *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*, pages 124–129, May 1997.

[10] M. Vojtko and T. Krajčovič. Adaptability of an Embedded Operating System: a Formal Description of a Processor. In *10th International Joint Conferences on Computer, Information,*

Systems Sciences, and Engineering, page 4, Dec. 2014. in print, http://fiit.stuba.sk/%7evojtko/VojtkoAoEOS.pdf.

[11] M. Vojtko and T. Krajčovič. Adaptability of an Embedded Operating System: a Generator of a Platform Dependent Code. In *Cybernetics and informaticcs (K&I), 28th International Conference on*, page 6, Feb 2016.

[12] M. Vojtko and T. Krajčovič. Semi-automated process of adaptation of embedded operating systems. *Journal of Electrical Engineering*, page 10, 2016. in review process, http://fiit.stuba.sk/%7evojtko/VojtkoJEEEC.pdf.

[13] E. Walkup and G. Borriello. Automatic synthesis of device drivers for hardware/software co-design. Technical report, University of Washington, Department of Computer Science and Engineering, Seattle, Washington, Jun 1994.

Selected Papers by the Author

M. Vojtko, T. Krajčovič. Semi-Automated Process of Adaptation of Embedded Operating Systems. In *Journal of Electrical Engineering*, 2016. Sent for review.

M. Vojtko, T. Krajčovič. Adaptability of an Embedded Operating System: a Generator of a Platform Dependent Code. In *2016 Cybernetics & Informatics (K&I)*, Levoca, Slovakia, 2016, pp. 1-6.

M. Vojtko. Adaptability of embedded operating systems. In *PESW 2015 : proceedings of the 3rd embedded systems workshop*, July 2015, pp. 1.

M. Vojtko, T. Krajčovič. Adaptability of an Embedded Operating System: a Formal Description of a Processor. In *In 10th International Joint Conferences on Computer, Information, Systems Sciences, and Engineering*, 2014. Springer. In print.

M. Vojtko, T. Krajčovič, Prototype of Modular Operating System for embedded applications. In *Applied Electronics (AE)*, 2013 International Conference on, Pilsen, Czech Republic, 2013, pp. 1-4.