DE GRUYTER OPEN

Journal of ELECTRICAL ENGINEERING, VOL 68 (2017), NO2, 87–98

Semi-automated process of adaptation of platform dependent parts of embedded operating systems

Martin Vojtko, Tibor Krajčovič*

Each year manufacturers develop new processors. As a reaction to this continuous development, the developers of software have to adapt their software to those new processors. As a minimal requirement, the code of an operating system has to be changed to enable the execution of other user applications. This change is a complicated process during which incompatible parts of an operating system have to be redesigned and missing parts have to be implemented. Complications arise when there is a need to adapt an operating system to completely different processor architecture. In this paper we present a novel adaptation process that has preconditions to reduce the impact of these complications. This process uses a file for the formal description of a processor, which is also described in this paper. The formal description could act as a standard for processor manufacturers and could allow the generation of a platform dependent code of an operating system. This paper presents concepts, definitions and ideas of the adaptation process and shows possible solutions for an automatic generation of code parts of an operating system.

K e y w o r d s: formal description of embedded operating systems, formal description of processors, automatic source code generation, platform dependent code

1 Introduction

Adaptability, the ability to easily adapt an existing system to a changing environment is, and will be, a great concern in the segment of embedded operating systems. This ability is an important competitive factor as it shortens the adaptation time and in this way, shortens the time to launch on the market.

Each year many processors become obsolete and many new processors are introduced to a market as their successors. This continuous change regarding new processor architectures raises the need for a methodology that allows a fast and efficient adaptation of the operating systems and embedded applications. In the near future, embedded systems will have multi-core or many-core architectures [1, 2]. Those architectures will introduce new types of operating systems that will be self-adaptive [3]. The operating systems will operate in a heterogeneous environment and they will use databases of existing processor ports, device modules and processing cores. When a new processor is plugged to a system, the modules and platform ports of the operating system will be loaded to the processor memory from the database during the system initialization or even online during a system run-time. To create such a database, the developer needs to implement missing device modules and processing core modules for an operating system.

The selection of a fitting architecture of an operating system has large impact on its adaptation. An operating system with a well-chosen architecture is easier to adapt. Most modern embedded operating systems have kernel architecture. This means that the kernel manages the devices and memories of a processor, the scheduling of processes, and the communication between them [4, 5]. Other services of an operating system, such as file management or device drivers, are separated from the kernel. The mentioned parts of the kernel must be modified during the migration to the new processor architecture, while services which are out of the kernel stay mostly unchanged. The level of difficulty of the change depends on the internal structure of the operating system kernel. A monolithic kernel represents an example of the type of architecture where even a small change in the code of the kernel can result into lasting problems [4]. On the other hand, a modular and layered kernel organization helps to reduce the complexity of the adaptation because the parts of kernel are loosely coupled [6]. The most affected part of a layered operating system is the platform dependent layer that acts as the glue logic between a processor and the parts of an operating system [7]. In this layer, the code that encapsulates the processor architecture has to be modified first, even though it is the least important in the eyes of the developer of an operating system.

In this paper, we present a novel adaptation process that consists of steps that allow a partial automation of the adaptation of an embedded operating system. This process is build for simple embedded operating systems consisting of a task scheduler, a memory manager and an input/output manager. We present the process that allows an automatic generation of the platform dependent parts of the operating system.

In the first place, it will be the definition of the Processor Formal Description (PFD). This definition has such a

*Institute of Computer Engineering and Applied Informatics, Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava, Slovakia, martin.vojtko@stuba.sk, tibor.krajcovic@stuba.sk

DOI: 10.1515/jee-2017–0013, Print (till 2015) ISSN 1335-3632, On-line ISSN 1339-309X

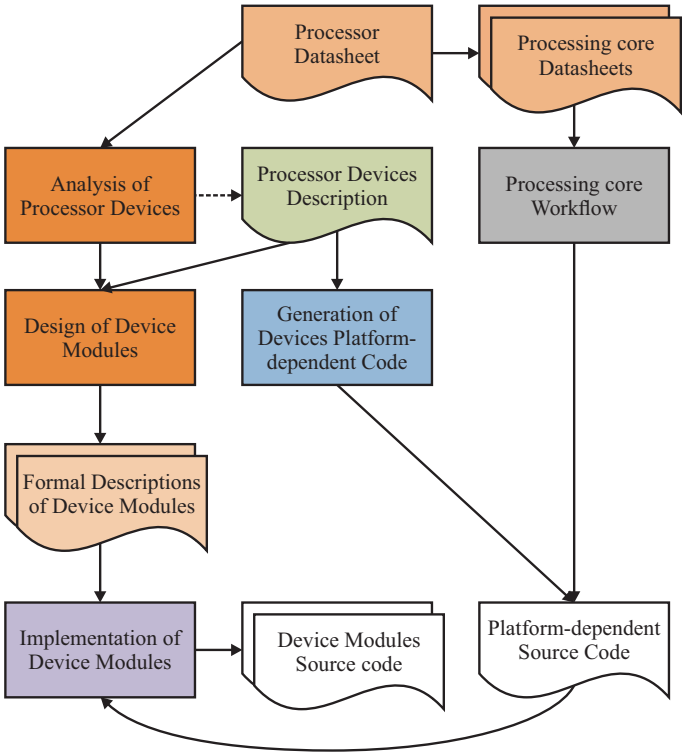


Fig. 1. Process of OS adaptation

format which can be automatically processed by a computer into operating system or any firmware source code.

In the second place, we present an example of simple generator of operating system code. The generator shows how the PFD can be processed into low-level code. A discussion about existing problems of the generated code is also included.

In the last place, the generated platform dependent code layer is used to design modules of the operating system that manage the existing devices of a processor. A simple example of a part of an operating system module shows how the generated code can be used during the design of parts of the operating system.

2 Process of OS adaptation

The adaptation process is designed to help the developer of an embedded operating system to produce a platform dependent code of an operating system for a chosen processor. It also helps to develop modules of an operating system that manage existing processor devices and processing cores. The process, described in Fig. 1, is suitable for embedded operating systems that have a layered architecture, consisting of at least one platform dependent and one platform independent layer (MOS [6], FreeRTOS [10] or others [11–14]). As the input for this process the developer needs the following documents:

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

The processor formal description (PFD) contains information about every device and core that is in the processor. This description helps to easily identify communication interfaces between devices and processing cores and the operating system. Those communication interfaces are the central part of the operating system platform dependent code. More information about the PFD can be found in Section 3.

According to Fig. 1, the adaptation process can be divided into two separate workflows. The workflow on the left side of Fig. 1 is focusing on processor devices. The workflow on the right side is focusing on processing cores that will not be presented in this paper.

The device workflow consists of the device analysis phase, the device module design phase, the platform dependent code generation phase and the device module implementation phase. The output of the workflow is the platform dependent code and the code of OS modules.

In the device analysis phase the functionality of each device is analyzed in the processor. The Device is any hardware component of processor that helps to receive or sent data (serial interface, analog to digital converter, ...) or can do auxiliary work (timer/counter, floating-point unit, ...).

In this phase, the developer of an operating system processes information about devices from a processor

datasheet. The developer searches for information such as device initialization, possible working modes, device timing diagrams, device behavior, and sources of interrupts. The output of this phase is the formal description of all processor devices. In the future, the manufacturer of the processor can perform this phase so that the developer will need to focus only on the operating system adaptation. In other words, the developer will have no extra work on creation of the PFD.

In the device module design phase, the developer produces the description of the device module. The device module is software that encapsulates processor device. In this phase, the description of the device communication interface is used from the PFD. The developer maps the interface to the module description and designs the connection between the module and the processor device. The result of this phase is the module description file, which allows full or partial code generation depending on the complexity of the device and the created module description.

A parallel phase to the module design phase is the device platform dependent code generation. In this phase, a platform dependent code is generated for the device communication interface. The resulting code based on the device formal description creates an interface layer between the processor device and the operating system module.

The last phase of the device workflow is the phase during which the description of the module is implemented and/or generated to a chosen programming language. In this phase, the device module is implemented and linked with the platform dependent code.

Similarly, as the device workflow, the core workflow consists of an analysis phase, a module design phase, a platform dependent code generation phase and a module implementation phase. In this work, we focus on the device workflow only.

3 Processor formal description

A formal description is needed to allow the automatic generation of the platform dependent code. The description proposed in this chapter describes a processor from the top to the bottom starting from processor devices and processing cores. First draft of the PFD was presented in [15], in this paper we present full definition.

Definition 1 Processor. A processor p is a triple $\{A, B, ap\}$, where $A = \{a_1, a_2, \ldots, a_n \mid n \in \mathbb{N}^+\}$ is a finite set of processing cores of a processor, $B = \{b_1, b_2, \ldots, b_n \mid n \in \mathbb{N}\}$ is a finite set of devices of a processor, and ap is address space of a processor defined as set of accessible addresses.

Set $B(p)$ is a finite set of devices B of a processor p.

Definition 2 Device. A device $b_i \in B(p)$, $i = 1, 2, \ldots, |B(p)|\}$ is a triple $\{R, I, S\}$, where $R = \{r_1, r_2, \ldots, r_n \mid n \in \mathbb{N}\}$ is a finite set of registers of a device, $I = \{i_1, i_2, \ldots, i_n \mid n \in \mathbb{N}\}$ is a finite set of interrupt signals of a device, and $S = s_1, s_2, \ldots, s_n \mid n \in \mathbb{N}\}$ is a finite set of manageable signals of a device.

Set $R(b)$ is a finite set of registers R of a device$b$.

Set $I(b)$ is a finite set of interrupts I of a device b.

Set $S(b)$ is a finite set of manageable signals s of a device b.

Similarly, as the devices, a processing core can be defined as a tuple of registers, signals, modes and intructions. A manageable signal is a signal that can be read or written by an instruction from the instruction set of a processor.

Definition 3 Register. A register of a device $r_i \in R(b)$, $i = 1, 2, \ldots, |R(b)|$ is a quadruple $\{ad, t, w, F\}$, where $F = \{f_1, f_2, \ldots, f_n \mid n \in \mathbb{N}^+\}$ is a finite set of register parts ad is an address of a register, t is a type of register which can assume a value from the set {'control', 'data', 'state'}, and w is the width of a register in bits.

Register of any processor device can be divided into smaller parts. Each part represents one configurable or measurable variable of the device. (*eg* serial interface mode register contains configuration bit for enabling parity check). One configuration represents one register part that can reach values from limited range (Fig. 2). Some of the values can have specific meanings. For those values, it is possible to give them descriptive names. We call those values as named values.

Definition 4 Register Part. A register part $f_i \in F(r)$, $i = 1, 2, \ldots, |F(r)|$ is a quadruple $\{m, g, O, D\}$, where m is a mask of a register part that represents an occupied place in a register, g is a type of register part operation that declares whether a part is read-able, write-able or both, $O = \{o_1, o_2, \ldots, o_n \mid n \in \mathbb{N}\}$ is a finite set of named values of a register part and the number of those values satisfies (1), and $D = \{d_1, d_2, \ldots, d_n \mid n \in \mathbb{N}\}$ is a finite set of structural dependencies of a register part.

Set $O(f)$ is a finite set of named values O of a register part f.

Set $D(f)$ is a finite set of structural dependencies D of a register part f.

$$0 \leq |O(f)| \leq 2^{\sum_{k=1}^{w(r) m(f)_k}}. \tag{1}$$

Definition 5. Let $w(r)$ be a register width, then a mask $m(f)$ of a register part is a binary vector that has width $w(r)$ and the sum of the coordinates in this vector satisfies (2). If coordinate $m(f)_k = 1$ of binary vector $m(f)$, then for register r it is true that the bit of register with coordinate number k is occupied by a register part.

$$1 \leq \sum_{k=1}^{w(r)} m(f)_k \leq |m(f)| = w(r)\,. \tag{2}$$

THEOREM 1. *Let us have register parts* $f_i, f_j \in F(r)$; $i, j = 1, 2 \ldots, |F(r)|, i \neq j$ *and a finite set of activation and deactivation dependencies* $D_{AD}(f_i) = \emptyset \wedge D_{AD}(f_j) = \emptyset$, *then the masks of register parts* $m(f_i)$ *and* $m(f_j)$ *are orthogonal and their scalar product satisfies*

$$\sum_{k=1}^{w(r)} m(f_i)_k m(f_j)_k = 0\,. \tag{3}$$

P r o o f . Set $D_{AD}(f)$ is explained in Definition 15 for now assume that this set is empty. Let us say that (3) is not satisfied, then at least two register parts f_i, f_j exist in a register that occupies the same bits in a register (they are not orthogonal). If this situation occurs the change in one part will lead to a change in the colliding part, so the register has poor design. □

THEOREM 2. *Let us have the register part* $f_i \in F(r)$, $i = 1, 2, \ldots, |F(r)|$ *and a finite set of activation and deactivation dependencies* $D_{AD}(f_i) = \emptyset$, *then*

$$|F(r)| \leq \sum_{i=1}^{|} F(r)| \sum_{k=1}^{w(r)} m(f_i)_k \leq w(r)\,. \tag{4}$$

P r o o f . Upper bound: From (3) we can deduce that for each $k = 1, 2, \ldots, w(r)$ we obtain

$$0 \leq \sum_{i=1}^{|F(r)|} m(f_i)_k \leq 1\,. \tag{5}$$

If we apply the sum of every coordinate to (5) we have

$$\sum_{k=1}^{w(r)} 0 \leq \sum_{k=1}^{w(r)} \sum_{i=1}^{|F(r)|} m(f_i)_k \leq \sum_{k=1}^{w(r)} 1$$

and this equals to

$$0 \leq \sum_{i=1}^{|F(r)|} \sum_{k=1}^{w(r)} m(f_i)_k \leq w(r)\,. \tag{6}$$

From the equation (6) it is true that from the upper bound the equation (4) is true. For the bottom bound, from the definition of a register it is true that $|F(r)| \geq 0$.

Lower Bound: If we apply the sum for each register part on (2) we have

$$\sum_{i=1}^{|F(r)|} 1 \leq \sum_{i=1}^{|F(r)|} \sum_{k=1}^{w(r)} m(f)_k \leq \sum_{i=1}^{|F(r)|} w(r)$$

and this equals to

$$|F(r)| \leq \sum_{i=1}^{|F(r)|} \sum_{k=1}^{w(r)} m(f)_k \leq w(r)|F(r)|\,. \tag{7}$$

From (7) it is true that from the lower bound the equation (4) is true. For the upper bound it is true that $w(r) \leq |F(r)|$. By combining of boundaries from (6) and (7) it is proven that equation (4) is correct. □

DEFINITION 6 NAMED VALUE. A named value of a part $o_i \in O(f)$, $i = 1, 2, \ldots, |O(f)|$ is a double $\{n, v\}$, where n is a name of a named value and v is a value of a named value of a register part represented by a binary vector, which satisfies

$$|v| = |m(f)| \wedge v + m(f) = m(f)\,. \tag{8}$$

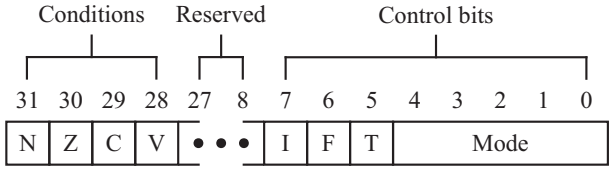


Fig. 2. Status register of arm7tdmi containing seven 1b register parts and one 5b register part. Part I is bit that enables interrupts on the processor. It has two named values: 0 = *disable* and 1 = *enable*.

DEFINITION 7. A signal of a device $s_i \in S(b)$, $i = 1, 2, \ldots, |S(b)|$ is a quadruple $\{ss, w, O, C\}$, where ss is the direction of a signal, w is the width of a signal, $O = \{o_1, o_2, \ldots, o_n \mid n \in \mathbb{N}\}$ is a finite set of named values of a signal and the number of those elements $|O|$ satisfies (1), $C = \{c_1, c_2, \ldots, c_n \mid n \in \mathbb{N}\}$ is a finite set of processing core inputs to which a signal is connected. If ss is an input then $C = \emptyset$.

DEFINITION 8. An interrupt signal $is \in I(b)$, $i = 1, 2, \ldots, |I(b)|$ is a double $\{c, Z\}$, where c is the input signal of a processing core or a device that the interrupt signal is connected to, and $Z = \{z_1, z_2, \ldots, z_n \mid n \in \mathbb{N}^+\}$ is a finite set of interrupt sources of an interrupt signal. Set $Z(is)$ is a finite set of interrupt sources Z of an interrupt signal is.

DEFINITION 9. An interrupt source $z_i \in Z(is)$, $i = 1, 2, \ldots, |Z(is)|$ is a double $\{f, o\}$, where f is a register part to which an interrupt source is coupled, and o is a named value of the register part which defines whether an interrupt source is active.

DEFINITION 10 STRUCTURAL DEPENDENCE. A structural depen-dence $d_i \in D(f)$, $i = 1, 2, \ldots, |D(f)|$ is a double $\{E, o\}$, where E is the expression of dependence, and o is the operation of dependence.

A structural dependence exists between register parts of a device or a processing core if a change of one register part leads to a change of another. The element that causes the change will be called originator and the element which is affected by the change will be called dependant. The dependant can be a register part only, while the originator can be a register, a signal or a register part. There is always one dependant and one or more originators in a dependence.

Definition 11 Expression. An expression of a dependence $E(d)$ is an abstract tree $\{Pz, Op, H\}$, where $Pz = \{pz_1, pz_2, \ldots, pz_n \mid n \in \mathbb{N}^+\}$ is a finite set of leafs (conditions of dependence) of a tree, $Op = \{op_1, op_2, \ldots, op_n \mid n \in \mathbb{N}\}$ is a finite set of internal vertices (logical operators) of a tree, and $H = h_1, h_2, \ldots, h_n \mid n \in \mathbb{N}\}$ is a finite set of oriented edges of a tree (Fig. 3).

Set $Pz(E)$ is a finite set of conditions Pz of dependence expression E.

Set $Op(E)$ is a finite set of operators Op of dependence expression E.

Theorem 3. *An expression of dependence E(d) can be decomposed into conjunctive clauses* $E_1 \vee E_2 \vee \cdots \vee E_n = E(d)$, *where* n *is number of conjunctive clauses. For every conjunctive clause* E_i, $i = 1, 2, \ldots, n$ *a new structural dependence* d_i *is created with the same operation of dependence* $o(d_i) = o(d)$. *A set of newly formed structural dependences is equivalent to the original structural dependence* $\{d_1, d_2, \ldots, d_n\} \equiv \{d\}$.

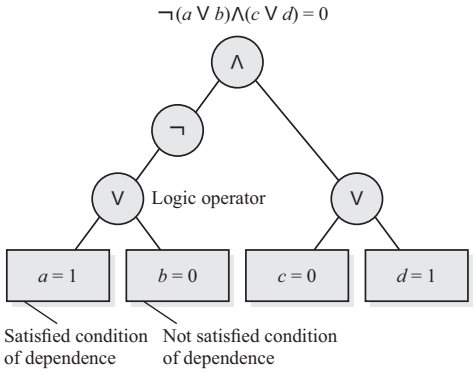


Fig. 3. An example of an expression of an dependence in a tree form and in a logical expression form

P r o o f . It is true that if at least one conjunctive clause in the expression $E(d)$ is satisfied then the whole expression is satisfied. Newly formed structural dependences have identical dependence operations so they do not violate each other when more of them are satisfied at the same time. □

The possibility to decompose expression of dependence into simpler clauses containing only disjunctions allows specifying sets of dependants that are changed when the expression is satisfied. It also allows simpler analysis of the processor model when platform dependent code is generated.

Definition 12. An oriented edge $h_i \in H$, $i = 1, 2, \ldots, |H|$ is a double $\{pv, pk\}$, where $pv \in Op \cup Pz$ is a vertex where the edge starts, and $pk \in Op$ is a vertex where the edge ends. $pv \neq pk$.

Definition 13 Condition. A condition of a dependence $pz_i \in Pz(E)$, $i = 1, 2, \ldots, |Pz(E)|$ is a triple $\{t, p, v\}$, where t is the type of the dependence, p is the originator of the dependence, and v is a value of the originator that will satisfy a condition. There exist 4 types of conditions (Fig. 4):

- *Read*, a condition pz is satisfied when a read operation is done on an originator $p(pz)$. An originator can only be a register and the value $v(pz)$ is undefined;
- *Write*, condition of dependence pz is satisfied when a write operation of any desired value is done to a $p(pz)$. An originator can only be a register;
- *WriteTo*, a condition of dependence pz is satisfied when a write operation of a defined value is done on a $p(pz)$. An originator can be a register or a register part and the value $v(pz)$ should be defined;
- *WrittenTo*, condition of dependence pz is satisfied when a $p(pz)$ is set to the defined value v.

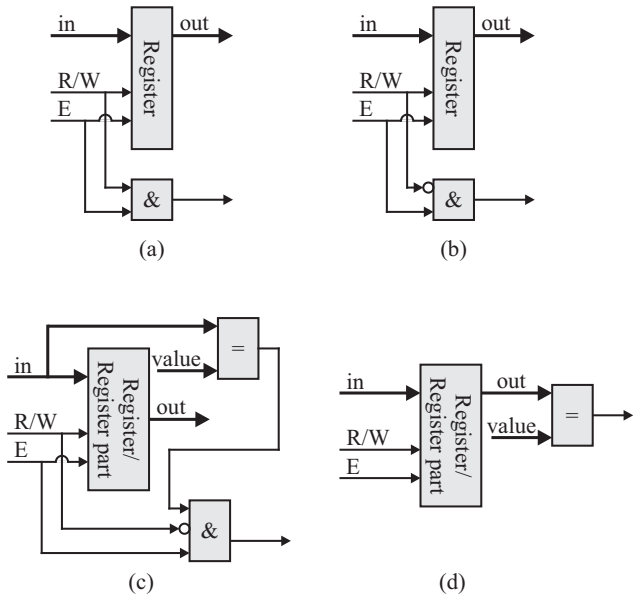


Fig. 4. Curcuit exapmles of the dependence conditions (a) Read, (b) Write, (c) WriteTo, (d) WrittenTo

The conditions of the types *Read, Write and WriteTo* (Fig. 4(a-c)) are satisfied in the time when the signal enabling the operation above the register is active. On the other hand, the conditions of type *WrittenTo* (Fig. 4(d)) are satisfied when the needed value is stored in a register or a register part regardless if the signal that enables the register is active.

Definition 14 Operation. An operation of dependence o is a double $\{t, v\}$, where t is a type of an operation, and v is a value which will be written to the dependant part when the expression of dependence is satisfied. There exist 5 types of operations:

- Set, an operation of dependence writes a constant value to a dependant;
- Reset, an operation of dependence writes zero value to a dependant;
- Write, an operation of dependence writes any desired value to a dependant;

- Activate, an operation of dependence activates the dependant;
- Deactivate, an operation of dependence deactivates the dependant.

Definition 15. Set $D_{AD} \subseteq D(f)$ is a set of all dependencies of register part f whose type of operation of dependence $t(o)$ is Activate or Deactivate.

Dependants with common expressions of dependence $E(d)$ and common operation type $t\big(o(d)\big)$ can be grouped into a domain of dependence. This domain represents set of dependants on which an operation is triggered when the condition of the dependence is satisfied.

Definition 16 Domain. A domain of dependence dz is a triple $\{Fz, Ez, t\}$, where $Fz = \{fz_1, fz_2, \ldots, fz_n \mid n \in \mathbb{N}^+\}$ is a finite set of dependants and for all $f \in Fz$ there exists at least one $d \in D(f)$: $E(d) = Ez$ where Ez is an expression of the domain, and $t = t\big(o(d)\big)$ is an operation type for all $f \in Fz$.

Set $Fz(dz)$ is a finite set of dependants Fz of dependence domain dz.

Set $Ez(dz)$ is an expression Ez of domain dz.

Definition 17. Let $F(r)$ be a finite set of register parts and let $Dz_A = \{dz_1, dz_2, \ldots, dz_n \mid n \in \mathbb{N}\}$ be a finite set of activation domains. Then $Dz_A(r) = \{dz_i \in Dz_A(r) \mid F(r) \cap Fz(dz_i) \neq \emptyset\}$ is a finite set of activation domains that affects the register r.

Definition 18. Let $F(r)$ be a finite set of register parts and let $Dz_D = \{dz_1, dz_2, \ldots, dz_n \mid n \in \mathbb{N}\}$ be a finite set of deactivation domains. Then $Dz_D(r) = \{dz_i \in Dz_D(r) \mid F(r) \cap Fz(dz_i) = \emptyset\}$ is a finite set of deactivation domains that affects the register r.

Definition 19. Let us have $F(r)$ and let $Dz_{AD} = Dz_A \cup Dz_D$ be a finite set of all activation and deactivation domains. Then $F_v(r) = F(r) - \big(F(r) \cap \big(\bigcup_{Dz_{AD}(r)} Fz(dz)\big)\big)$ is a finite set of all register parts that do not belong to any activation or deactivation dependence.

Definition 20. Let us have $F(r)$ and let Dz_A be a finite set of all activation domains. Then $Fn(r)(T) = F(r) \cap \big(\bigcup_{Dz_A(r)} Fz(dz) \mid E(dz) = 1\big)$ is a finite set of register parts that belongs to a set of activation domains that are active (their dependence expression is satisfied) in a defined time T.

Theorem 4. *Let $Fa(r)(T) = Fn(r)(T) \cup Fv(r)$ be a finite set of such register parts that do not belong to any activation or deactivation dependence or such register parts that belong to activation domains that are active. For any two desired $f_i, f_j \in Fa(r)(T)$; $i, j = 1, 2, \ldots, |Fa(r)(T)|$, $i \neq j$, equation (3) is satisfied.*

Proof. The activation and deactivation dependences affect Theorem 1, because there can exist register parts that can overlap other parts. In a specified point in time T some register parts are activated and others are deactivated and in this time, point the activated register parts do not overlap. So, the proof is the same as for Theorem 1. □

Theorem 5. *Let us have $Fa(r)(T)$, then (4) can be transformed to*

$$|F(r)(T)| \leq \sum_{i=1}^{|F(r)(T)|} \sum_{k=1}^{w(r)} m(f_i)_k \leq w(r)\,. \qquad (9)$$

Proof. If we use a specific time point T, the same proof can be applied as for Theorem 2. □

The concept of domains allows us to group all dependants into a set of common dependences. In the next definition, we present concept of grouping domains that have common originator.

Definition 21 Originator. An originator of dependence pw is a double $\{pz, Dz\}$, where pz is the originator of dependence, and $Dz = \{dz_1, dz_2, \ldots, dz_n \mid n \in \mathbb{N}^+\}$ is a finite set of domains where $oz \in Pz\big(E(dz)\big)$ for all $dz \in Dz$.

The defined PFD can be used to describe any desired processor; however, we have no proof. To proof of our statement, we would need to apply PFD on any existing processor. We described 5 existing processors from different manufacturers and for those processors the PFD was applicable. The way how the PFD was defined allows its extension in the future.

Figure 5 shows a graphical representation of the resulting PFD. The representation shows the relationships between the defined parts of the formal description. A model of the PFD is represented by the file in JSON format described in [15]. Each item in the formal description has its identification that helps in the naming of the generated functions and code. The expressions of structural dependences are written as a logical expression in a full disjunctive normal form (FDNF). This form allows the decomposition of structural dependences into conjunctive clauses.

4 Generation of platform dependent code

A generator of code processes information obtained from the PFD. In this paper, we describe how the generator can in theory process the PFD and how the generated platform-dependent code can look. We discuss negative effects of structural dependences on the generated code and we propose solutions that can suppress them.

At the first we should stress that the implementation of the generator is in hands of the developers of the OS. Each OS has its own specifics and that is why it is not possible to implement one generator for any operating system without changing its upper platform-independent application layers.

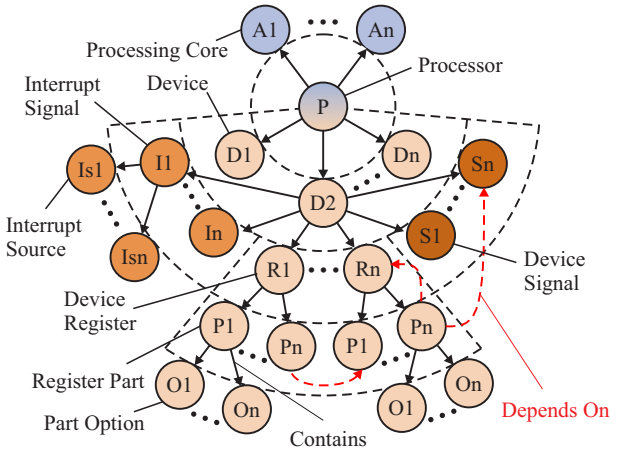


Fig. 5. Graphical representation of the PFD

Table 1. Definitions of a platform dependent code. idD - id of a device, idR - id of a register, idP - id of a register part, idH - id of a named value, idS - id of a signal.

Item	Definition Code
Device address	#define <idD>_ADDRESS <adr>
Register offset address	#define <idD>_<idR>_OFFSET <offset>
Register address	#define <idD>_<idR>_ADDRESS <offset + adr>
Register part mask	#define <idD>_<idR>_<idP>_MASK <mask>
Named value of a register part	#define <idD>_<idR>_<idP>_<idH> <value>
Named value of a signal	#define <idD >_<idS>_<idN> <value>

Table 2. Definitions of functions. idD - id of a device, idR - id of a register, idP - id of a register part

Target	Output	Name	Parameter
register	dataset	read_<idD>_<idR> ()	-
	-	write_<idD>_<idR> ()	dataset
register part	dataset	get_<idD>_<idR>_<idP> ()	dataset
	dataset	set_<idD>_<idR>_<idP> ()	dataset, data

The process of code generation can be divided into three phases. In the first phase, that is called definition phase, definitions are generated for every item of the PFD. Those definitions link every item with an identifier which helps to understand the communication interface of a device. In the second phase, that is called function phase, the interface functions that manage write and read operations above registers and register parts are generated. The generated functions use definitions to address the needed register or a register part of a device without any loops or conditions. In the last phase, the existing structural dependences are processed. A code commentary is generated for every dependant and for every originator [16].

The simplicity of the PFD allows us to implement a generator which produces the code in the programming language preferred by the developer of an operating system and in the structure and nomenclature which is compatible with any operating system structure. For the presented generator, we chose programming language C. Another advantage of the generator is that it reduces the possibility of bugs in the code that could be created by the developer of the operating system.

The platform dependent code is simple but errors like wrong addressing of the register or a register part are common.

4.1 Definitions

A definition is a part of code that specifies the name and value of a constant that is taken from the PFD. Table 1 shows a list of items from the PFD for which the definitions are generated. The generated definitions are used in function bodies.

4.2 Functions

The functions of the platform dependent code are used to perform a write or a read operation above the register of the device, or to perform a set or a get operation above the register part.

Table 3. Examples of function bodies. idD - id of a device, idR - id of a register, idP - id of a register part

Function	Function body
read	int *part = <idD>_<idR>_ADDRESS; return *part;
write	int *part = <idD>_<idR>_ADDRESS; *part = data;
set	data |= (dataset & <idD>_<idR>_<idP>_NMASK); return data;
get	dataset &= <idD>_<idR>_<idP>_MASK; return dataset;

The name of the functions (declarations) consists of the identification of the device, the register and the register part. Table 2 shows prototypes of function declarations. The bodies of the functions will be built from definitions generated in the previous step. Table 3 shows examples of function bodies.

Those simple functions encapsulate every register of every device so the developer of an operating system does not need to know the addresses of registers and the placement of the register parts in a register. The encapsulation simplifies the design of the upper layers of an operating system. The function bodies shown in Table 3 can be represented as functions where a function call is needed but the source code size is smaller, or as a macro, or as an inline function that decreases the overhead of the operating system but increases its memory footprint.

Table 4. Possible combinations of conditions in an expression of dependence. Usr are conditions where an action of the program is needed. Sys are conditions where action of a system is needed. x possible, R if originators are in same register, ? questionable, - not possible

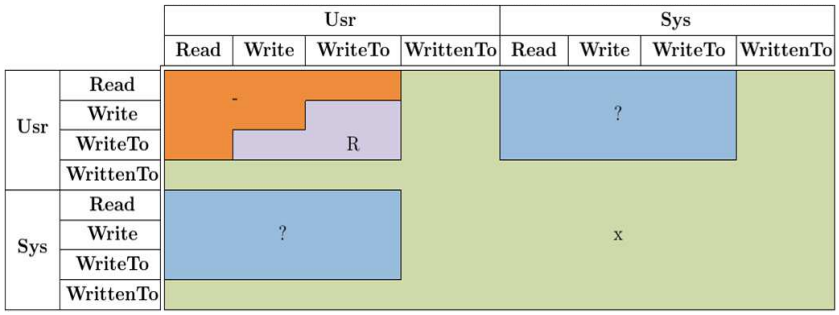
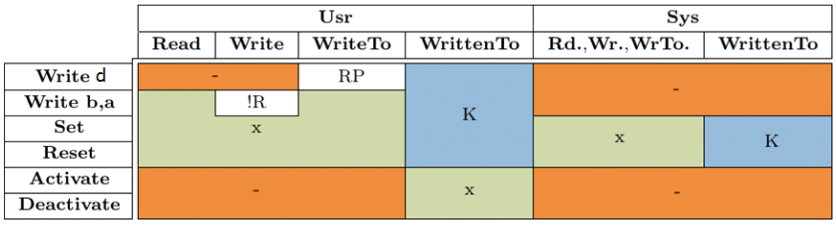


Table 5. Possible combinations of conditions with operation of dependence. Usr are conditions where the action of the program is needed. Sys are conditions where the action of a system is needed. x possible, R if originators are in same register, P if originator is a register part, K if combined with another condition, - not possible. d during, b before, a after



4.3 Structural dependence

The existence of structural dependence between the register parts in a device, interposes problems in the adaptation process. The change of a value of one register part can lead to a change of other register parts in a device. The developer has to know about these changes and that is why the generator has to generate blocks of code that inform about existing dependence or provide a secure interface that handles dependence.

A simpler solution is that the generator generates commentaries for each dependence. Those commentaries inform the developer of the operating system about dependences so he can produce code that avoids problems. This solution is used in the current working example of the generator.

A harder but more effective solution is that the generator tries to cover the dependence in a function that hides it from the developer so he can focus on the development of more important parts of an operating system. This solution is complicated by more aspects that are further discussed in the next paragraphs.

Before we can discuss the problems that complicate the encapsulation of structural dependences we briefly describe the relations between dependence conditions in a dependence expression, dependence conditions and dependence operations. We identified that a structural dependence is activated when the dependence expression is satisfied. We identified that there exist 4 types of conditions. In advance, there exist conditions that can be satisfied by a system event (Sys conditions, no function call affecting the originator is needed) or there exist conditions that are satisfied when a specific function is called by the user (Usr conditions).

Constraints in conjunctive clauses of expression restrict combinations of condition types, as is shown in Table 4 *Read* and *Write* conditions, or *Read* and *WriteTo* conditions cannot be combined because there can be executed only one operation at a time. Conditions of the *WriteTo* type can be combined in an expression only if the originators of those conditions belong to one register. The conditions that are managed by the system can be combined without restriction because they can be satisfied by events that were created by the environment. It is questionable if the System and User conditions can be combined, because the system event has to occur in a time when the user condition is satisfied and this is highly improbable.

The type of the dependence operation constrains the condition type as is shown in Table 5. The *Write* operation can be done *before, after* or *during* the time when the expression of the dependence is satisfied. *During Write* operation is an operation that writes user data to a register part at the same time when the dependence expression is satisfied. On the other hand, *Before Write* operation writes user data before expression is satisfied and *After Write* operation writes user data after expression is satisfied.

Constraints in dependence expressions reduce the amount of possible combinations of condition types in a one structural dependence. This can help in finding a

Table 6. Proportion of origintator types, domain types and dependants of the processor AT91SAM7S256

								Domains	Dependants
Originators							1183	1444	1803
of that	Simple 1-domain						1017		
	of that	Simple						1017	1163
		of that	1-item					932	932
			of that	Usr				918	918
				Sys				14	14
			1-register					82	222
			of that	Usr				82	222
				Sys				0	0
		Other						3	9
	Simple n-domain						94		
	of that	Simple						302	503
		Other						0	0
	Complex						72		
		Other						125	137
Sum of items for which a code generation is possible:								1000	1140

solution for dependence encapsulation by the platform dependent code. The next paragraphs discuss the problem of this encapsulation.

Let us say that there exists a structural dependence that changes the value of the dependant when the originator of the dependence is set to a specific value. For example, let us say that the dependant is placed to a read-only register. There could be a generated function that sets the originator to a specific value with the purpose of changing the value of the dependant. The newly formed function partially substitutes the missing write function of the dependant. The developer of an operating system can use this function instead of using the write function of the originator that writes the specific value that can be read in a commentary. The generated function simplifies the process of adaptation.

The indicated example can be applied only if the originator is not coupled to more dependence domains. A critical collision of domain expressions can exist and can result into changes of dependants that we did not want to change.

Definition 22. A domain dz is a **user** domain when for all conditions of dependence $pz \in Pz\big(Cz(dz)\big)$ applies that the type of the condition $t(dz) \in$ $\{WriteUsr,\ ReadUsr,\ WriteToUsr,\ WrittenToUsr\}$.

Definition 23. A domain dz is a **system** domain when for all conditions of dependence $pz \in Pz\big(Cz(dz)\big)$ applies that the type of the condition $t(dz) \in$ $\{WriteSys,\ ReadSys,\ WriteToSys,\ WrittenToSys\}$.

Definition 24. A domain dz is a **simple** domain when the cardinality of all conditions of the dependence $\big|Pz\big(Cz(dz)\big)\big| = 1$.

Definition 25. A domain dz is an **n-item** domain when the cardinality of the all dependants $|Fz(dz)| = n$.

Definition 26. A domain dz is a **1-register** domain when there exist only one register r where for all $fz \in Fz(dz)\colon fz \in F(r)$.

Definition 27. An originator pv is a **simple** originator when for all $dz \in Dz(pv)$ it applies that dz is a simple domain and $p\big(Pz\big(Cz(dz)\big)\big) = pv$.

Definition 28. An originator pv is a **n-domain** originator when the cardinality of its domains $|Dz(pv)| = n$.

The classification of domains and originators allows a deeper analysis of the existing connections between the dependants and the originators. The best scenario for the developer is achieved when all structural dependences are hidden behind the functions so that the platform dependent code is easy to understand. The reality shows that this is not possible for all structural dependences. Table 6 summarizes the analysis of all structural dependencies in the PFD of the processor AT91SAM7S256.

It is shown that for this processor much of the originators are simple 1-domain originators (nearly 86 %). For those originators, there exists an equal number of domains because one originator belongs to only one domain. The larger part of those domains consists of 1-item domains (they contain only one dependence) so there is no possibility of collision. The second group of domains is a group of 1-register domains which affect dependants that are in the same register. For 1-item and 1-register groups it is possible to generate a function without collision. There is no need for function generation of Sys domains because the dependence expression is satisfied automatically by the related event that occurs in a device.

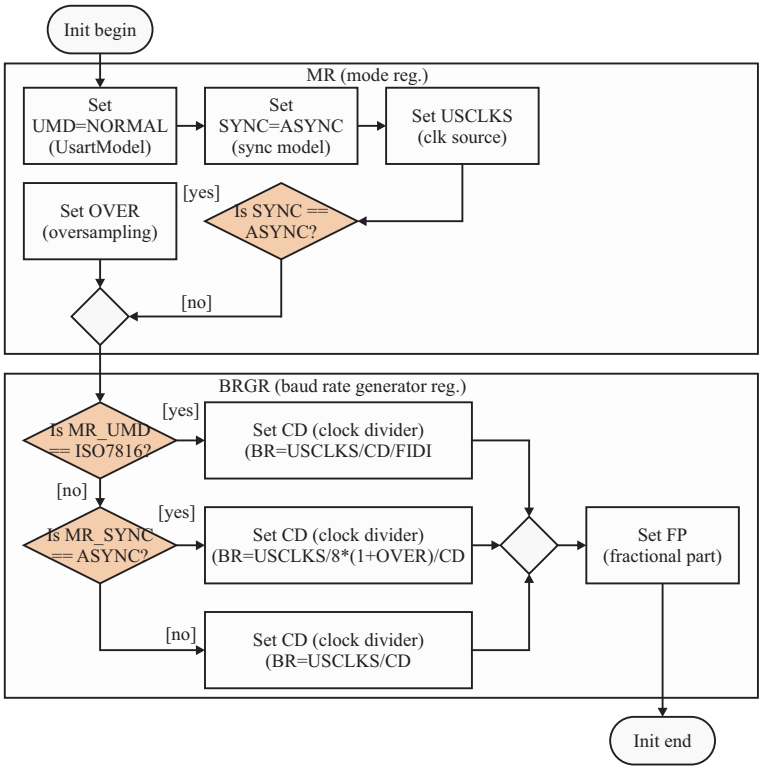


Fig. 6. Init function of universal receiver/transmitter

The second group of originators is a group of simple n-domain originators. Those originators originate in more domains but in all those domains there is only one originator in a dependence expression. The change of the originator can satisfy expressions of more domains with different operations. Those domains can violate each other. The function that hides dependencies should solve those violations that complicate the generated code (the code loses scalability and effectiveness).

The last group consists of originators that exist in complex domains. For dependencies that are part of those domains, the relations between the originators are too complicated for an effective generation of a code. The code of the platform dependent layer can change into a hardly manageable and ineffective bulk. This is against the idea of the platform dependent code: it should be as simple as possible to act as the glue logic [9].

Between the parts of the device, situations can exist where the change of one part can start a chain of events. The change of one originator changes the dependant that is the originator of the next dependence and so on. Those situations complicate the generation of the code even more.

As a result, we decided that for now the generator will produce only simple commentaries that can be processed by specific tools that help to design the operating system modules.

5 Formal description of OS modules

The device module of the operating system manages the mapped device of the processor. It uses platform dependent code prepared by the generator. A formal description can be used in the design of modules in order to simplify the adaptation. In this chapter, we describe the design of operating modules that uses workflow diagrams.

The module of operating system can be divided into 3 parts which are modeled independently:

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

The processor device is often a complex structure which can operate in more modes and under many influences. The initialization declares how a device will be used or how the device operation will be altered during the system execution. This part of the module is represented by a workflow diagram that represents the initialization of the device. This workflow can be easily transformed into the initializing function of the device module.

Most processor devices can produce an interrupt signal which informs the processor about a specific situation that occurred in a device. The description of the interrupt describes how each interrupt source of a device will be processed. The interrupt handler is most often represented by a simple set of conditions that search for the source of the interrupt in a device and call predefined routines that will be executed (*eg* the calling of a data processing routine).

```
void init(int USCLKS, 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, USART_MR_UMD_ASYNC);
  dataset = set_USART_MR_USCLKS(dataset, USCLKS);
  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);
}
```

Fig. 7. Example of init function code

Device data processing routines perform write or read operations to or from the device. The work with data is managed with control commands that provide the communication interface of the device and the status information that informs about the state of data processing.

A module of operating system can be described by a workflow diagram, as is shown in Fig. 6. The example shows a diagram that represents the initialization function for a universal receiver/transmitter interface. In the diagram, the mode and baud rate registers are set to needed values. Each block of the diagram represents a call of a defined set of functions of the register part. Those functions were generated by the generator of the platform-dependent code. The register part can be initialized into a constant (see UMD block in Fig. 6) or it can be set by the user during the function call. The envelope of diagram blocks (see MR in Fig. 6 represents the register. The envelope is transformed into a register write or read function. Fig. 7 shows an example of code that can be created from the previously presented diagram. As can be seen, some register parts are set by the input parameter of the function (USCLKS, OVER, ...) while others are set into a constant value.

In the previous chapter, we mentioned that the generator creates commentaries for structural dependences. Those commentaries can be used by the tool that is used for workflow diagram creation. The tool can generate warnings when the user writes a code that can result into changes that could be unwanted.

6 Conclusions

The presented process of adaptation applies automatic generation techniques to create platform dependent parts of any operating system. The core of this process is the processor formal description that allows the generation of the code. The description stores information about processor devices and cores in a form that can be processed by the computer. The drawback of this description is that the developer should prepare this description with his own effort. We believe that some sort of processor description should be used by the processor manufacturers to provide better service for their products (a standard should be stated).

We also identified that structural dependences exist between the devices of the processor. Those dependences are the biggest source of errors during the adaptation or implementation of platform dependent code. The generator of code that processes data from the Processor Formal Description was presented in this work. The latest version of the generator generates simple commentaries for the discovered dependences. In this work, we discussed the possibilities that can cover the dependencies by simple routines that hide dependences so that the developer can work on more important parts of the operating system.

The generated code can be used not only for the operating systems but it can also help in embedded software development of any kind. The adaptation process can be used to adapt existing firmware built for one processor to another processor with smaller effort.

Acknowledgements

This article was created with the support of 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] P. Ranganathan, "From Microprocessors to Nanostores: Rethinking Data Centric Systems", *Computer*, vol. 44, no. 1, pp. 39-48, January 2011.

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

[3] M. Seltzer and C. Small, "Self-monitoring and self-adapting operating systems", *Proceedings of Operating Systems*, 1997 The Sixth Workshop on Hot Topics, pp. 124-129, May 1997.

[4] A. S. Tanenbaum and A. S. Woodhull,*Operating Systems Design and Implementation* (3rd Edition) Prentice Hall, 2006.

[5] J. J. Labrosse, J. Ganssle and E. A. Oshana, "Embedded Software: Know It All (Newnes Know It All)", *Newnes*, 2007.

[6] M. Vojtko and T. Krajcovic, "Prototype of Modular Operating System for embedded application", *Applied Electronics (AE)*, 2013 International Conference, pp. 1-4, September 2013.

[7] P. Chou, R. Ortega and G. Borriello, "Synthesis of the hardware/software interface microcontroller-based systems", *Computer-Aided Design*, ICCAD-92, Digest of Technical Papers, IEEE/ACM International Conference, pp. 488495, November 1992.

[8] P. Chou, R. Ortega and G. Borriello, "Interface co-synthesis techniques for embedded systems", *Computer-Aided Design*, 1995, ICCAD-95, Digest of Technical Papers, IEEE/ACM International Conference on, ISSN 1092-3152, pp. 280287, November 1995.

[9] G. Borriello, P. Chou and R. Ortega,*Embedded system co-design - towards portability and rapid integration* Integration Hardware/Software Co-Design, M. G. Sami and G. De Micheli, Eds., Kluwer Academic Publishers, pp. 21, 1995.

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

[11] TinyOS, 2011, http://www. inyos.net.

[12] S. Bogan, "Formal Specification of a Simple Operating System", *PhD dissertation*, der Naturwissenschaftlich-Technischen Fakultten der Universitt des Saarlandes, August 2008.

[13] J. Drrenbcher, "Vamos microkernel: Formal models and verification, 2006", *Talk given at the International Work-shop on Systems Software Verification*, August 2006, [Online], Available at http://www.cse.unsw.edu.au/formalmethods/events/svws-06/ VAMOS Microkernel.pdf.

[14] S. Beyer, C. Jacobi, D. Krning, D. Leinenbach and W. J. Paul, "Putting it all together formal verification of the vamp", *International Journal on Software Tools for Technology Transfer (STTT)*, 2006.

[15] M. Vojtko and T. Krajcovic, "Adaptability of an Embedded Operating System: A Formal Description of a Processor", *10th International Joint Conferences on Computer*, Information, Systems Sciences, and Engineering, pp. 1-4, December 2014.

[16] M. Vojtko and T. Krajčovič, "Adaptability of an Embedded Operating System: A Generator of a Platform Dependent Code", *International Conference on Cybernetics and Informatics 2016*, 1.2.2016.

Received 10 May 2016

Martin Vojtko (Ing, PhD) was born in Bratislava, Slovakia, in 1987. He received MSc degree at Slovak University of Technology in Bratislava in 2012. In 2016 he received PhD degree at the Institute of Computer Systems and Networks of Slovak University of Technology in Bratislava. His research is oriented on design of operating systems for embedded applications and for automated generation of code for those systems.

Tibor Krajčovič (Assoc Prof, Ing, PhD) was born in Nitra, Slovakia, in 1960. He received his MSc degree at the Slovak University of Technology (STU) in Bratislava in 1984 and his PhD degree from the same university in 1989. Since 2001, he has been an associate professor at the Faculty of Electrical Engineering and Information Technology of the STU in Bratislava. Now, he is a staff member of the Computer Engineering and Applied Informatics Institute at the Faculty of Informatics and Information Technologies of the STU in Bratislava. His research interests include microcomputers, embedded systems and Internet of Things. He is an author the one patent and more than 60 published scientific and research papers from these areas. He is also the member of the Slovak Commission for UNESCO, National Committee for the Information for All Programme.