Adaptability of an Embedded Operating System: a Formal Description of a Processor

Martin Vojtko, Tibor Krajčovič

***Abstract*—The heterogeneity and complexity of future processors will introduce problems with the adaptation of operating systems. When a new processor is presented on a market, operating systems need to be adapted to it. It is done by reprogramming of a platform dependent layer and device modules of an operating system. In this paper, we introduce a formal description of processor structure by which we reduce adaptation time of the operating system to a new processor. This is achieved by automated code generation of platform dependent layer of the operating system. In connection to the adaptation process we present a concept of an operating system development framework that reduces time of the design process. A classification based on the connection of processor device to a processing unit of a processor is introduced to make a connection between code from different layers of an operating system to processor devices and processing units. In addition to the above mentioned classification we also introduce a classification of an operating system code.**

***Index Terms*—Instruction Set Architecture, Modular Operating Systems, Layered Operating Systems, Platform Dependent Code.**

I. Introduction

EMBEDDED systems are more and more complex and heterogeneous nowadays. Techniques of embedded system design such as the hardware-software co-design [1] has to be used to decrease the time to market and the price of a development. Decomposition of a problem into a set of tasks is another way how the developer can reduce the complexity of a design, but the decomposition introduces new problems connected with task management (e.g. task planning, inter-task communication). An operating system helps to solve this problem by a complexity encapsulation [2].

An operating system is often used in an embedded system when a complicated set of tasks is needed to meet design needs [3]. When the system is constrained by deadlines, the need for an operating system is even higher [4].

The growing number of different processor architectures and families introduces problems in the adaptation of an operating system. This problem is more significant in the segment of embedded systems. Adaptation problems are reduced by the introduction of a layered and modular architecture of an operating system [5], [6], [7]. The layers in the architecture help to keep changes of an operating system code in the lower layers of the architecture without affecting the compatibility with user applications in the higher layers. On the other hand, the modularity helps to keep the changes of Operating system code only in the affected modules.

The adaptation of an operating system to a new processor or to a new processor family will be more complex in the future [8]. When an operating system needs to be adapted to a new processor, parts of an operating system code (in the platform dependent layer) need to be changed. Since the platform dependent layer of an operating system is tightly connected to processor architecture, the complexity of the platform dependent layer will grow with complexity of a processor.

This leads to the idea that this layer can be built automatically from a defined processor description. In other words, the developer can focus on other aspects of a system design.

II. Related Work

The majority of processor manufacturers sell their products together with software development tools that allow simpler development of embedded systems. Those tools provide to the developer a set of standard device drivers and code samples for processor devices. A good example is *Keil MDK tool,* which supports a wide range of *ARM* microcontrollers and provides software packs prepared for processor devices. The developer can use this tool to create own embedded applications in an easier way [9].

The problem arises when the developer wants to use a processor that is not supported by the chosen development tool. The solution to this problem can be a transfer of a processor description to the preferred development tool of a developer. We think that there does not exist any standard for a processor description that allows the transfer.

Furthermore, the majority of manufacturers provides its solutions, code samples, and headers to its processors. For example, the header file can contain an initializing code that is useful for a developer, but existing code cannot be used for a platform dependent layer because they are not compatible with a chosen operating system.

Manuscript received November 9, 2014. This work was supported by the Scientific Grant Agency of the Ministry of Education, Science, Research and Sport of the Slovak Republic under Grant VEGA 1/0676/12.

Martin Vojtko, is with the Institute of Computer Systems and Networks of Faculty of Informatics and Information Technologies STU in Bratislava, Ilkovicova 4, 84216 Bratislava, Slovakia (e-mail: martin.vojtko@ stuba.sk).

Tibor Krajčovič, is with the Institute of Computer Systems and Networks of Faculty of Informatics and Information Technologies STU in Bratislava, Ilkovicova 4, 84216 Bratislava, Slovakia (e-mail: tkraj@stuba.sk).

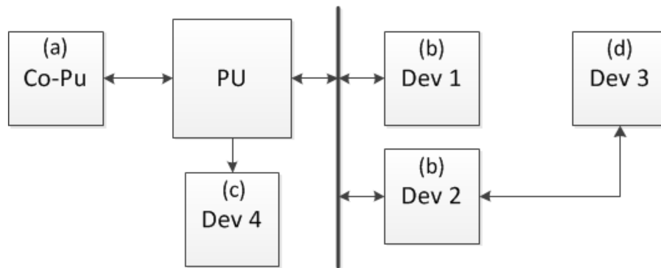

Fig. 1. The device classification based on a connection to a processing unit. (a) co-processing device CoPU, (b) bus-internal device Dev1 and Dev2, (c) signal-internal device Dev4, (d) external device Dev3 connected through Dev2.

In addition, header files provide a simple device register mapping to higher language structures that helps in embedded application development. The problem is that this initializing code is different from manufacturer to manufacturer, which makes usability of this code for an operating system unthinkable.

III. Processor

A standard processor consists of one or more processing units (processor cores), and a set of peripheral devices.

The task of a processing unit is to process instructions that use data as an input and produce other data as an output.

On one hand, processing units process instructions and its data, on the other hand, peripheral devices help to distribute data from/to processing units. In other words, those devices help to communicate with a surrounding environment (e.g. Analog-to-Digital Converter, Universal Asynchronous Receiver/Transmitter) or perform extra services to processing units (e.g. Timer/Counter, Watchdog Timer).

A. Instruction Set Architecture (ISA)

An Instruction Set Architecture (ISA) describes a user interface of a processing unit that is represented by an instruction set, internal registers, possible operating modes (e.g. real mode and system mode presented in x86 architecture), a width of an address bus, and a width of a register word. In other words, the ISA tells how to work with a processing unit, which instruction has to be used, how a task context during a task switch can be stored, and how different modes of a processing unit works.

B. Device Classification of a Processor

A device is a separate functional unit which provides services to a processing unit. In general it behaves as a slave to a processing unit. A device listens to commands from a processing unit and can inform a processing unit about certain specific events only by an interrupt signal. From our point of view, it is important to introduce a classification of devices based on a type of connection between a device and a processing unit (see Fig. 1.).

We identified four types of devices. First of all, it is a separate, dedicated bus connection between a processing unit and a co-processing unit. A connection between those two units can be controlled only by a particular set of instructions. We call this type of a device a co-processing device. The second connection type is an internal bus connection. An internal bus (or system bus or an address/data bus) provides a direct connection between a processing unit and a device that is accessible in an assigned range of addresses. The device connected to an internal bus is called a bus-internal device. The third type of a connection is the direct connection of a device to I/O pins of a processing unit. This type of a connection is not often used; usually, a special device is preferred for the management of I/O pins of a processor. A device connected to I/O pins of a processing unit is called a signal-internal device. The connection of a device to a processing unit through an internal device is the last type in our classification. These devices have no internal address that can be accessed by the processing unit directly. A driver needs to be written to allow access to the device. This type of device is called an external device.

A bus arbiter (e.g. Direct Memory Access device, Bridge, Memory controller) often exists between a processing unit (or a group of processing units) and other devices. In spite of this, a device connected to this arbiter can be still marked as a bus-internal device, because the arbiter behaves transparently to a processing unit. In other words, a device can still be accessed directly by a processing unit at a specific range of assigned addresses.

C. Device description

A device provides a communication interface to a processing unit or to other master devices. The interface is used to exchange data between the device and a master, and it is used to accept commands that change the device behavior.

An operating system has to be able to understand this communicating interface. In other words, an operating system must have a mapping of the device communication interface to its structures. When the mapping is created, an appropriate module of an operating system can use it to manage and to control the device.

D. Code Classification

The classification based on connection is important to clearly determine the difference between types of an operating system code used to work with a device. Co-processing device, bus-internal device, and signal-internal device can be accessed directly by a simple control routine. On the other hand, an external device must be accessed by a more complicated code such as a driver. The control routines can write or read data from/to registers of a device, or I/O signal wires. Since write and read operations are dependent on hardware, these routines are placed into a platform dependent layer of an operating system. The module of an operating system managing an appropriate device (device module) can use these routines to manage and to control a device. Operating system device modules represent a platform independent layer of an operating system which is placed on top of a platform dependent layer of operating system. This platform independent layer is a modular layer, where each module manages one device of a processor.

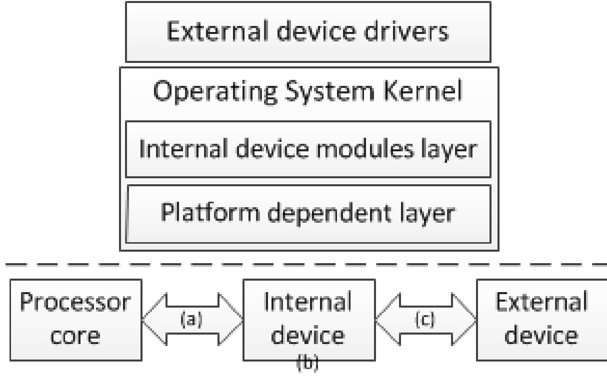

Fig. 2. Layers of Operating system code (top) in connection with processor hardware (bottom).

The control code of an external device is more complex than the previous types of code. An operating system will need to use a device module of an internal device through which the external device is connected to a processor. In this situation, we are talking about a device driver. A device driver is not a part of an operating system kernel and is out of scope of this work.

To summarize this chapter, in an operating system it is possible to identify three layers of a code (see Fig. 2.). The first layer (Platform dependent layer) contains routines of an internal devices used to access device registers. The platform dependent layer encapsulates the communication interface between a processor core and an internal device (see Fig. 2.a). The second layer (Internal device modules layer) contains internal device modules used to manage and to control processor devices (see Fig. 2.b). The third layer (External device drivers) contains an external device drivers used to manage and to control external devices (see Fig.2.c).

IV. Formal Description of a Processor

A formal description of a processor describes all internal components of the processor. By this description we want to translate a processor datasheet into a computer readable format, which allows future platform dependent code generation. In this chapter, we present the description.

A processor P is a triple $\{A, D, M\}$, where ISA is an Instruction Set Architecture of a processor, D is a set of all devices in a processor, and M is a set of all addressable memories. In this paper, we will present the ISA briefly, and the set of devices D in detail.

A. *ISA*

The instruction set architecture (ISA) describes functionality of a processing unit. The ISA is a pentad $\{R, M, IS, AW, RW\}$, where R is a set of all internal registers $\{r_1,\ r_2 \cdots r_n\}$ of a processing unit, M is a set of all unit modes $\{m_1,\ m_2 \cdots m_n\}$, IS is an instruction set of a core, AW is width of address bus, and RW is width of a processing unit registers (architecture).

B. *Set of Devices*

The set of all devices $D = \{d_1,\ d_2 \cdots d_n\}$ consist of four subsets, according to the classification based on a connection (III-B). In this paper, we focus on bus-internal devices.

Assume a set of bus-internal devices $D_{bi} \subseteq D$. For each $d_i \in D_{bi}$, where $i = 1, 2 \cdots n : d_i$ is a triple $\{DR, I, B\}$, where DR is a set of all device registers, I is a set of all device interrupt signals, and B is a set of all possible behaviors of a device (V-B). The set of device registers and the set of interrupts form the communication interface of a device, which will be managed by platform dependent layer of an operating system. The behavior B of a device will be managed by a device module of the operating system from the platform independent modular layer [5].

C. *Device registers*

Each device register can be divided into smaller logical parts that manage pieces of communication interface or control the mechanism of a device (e.g. set clock division factor in prescaler register of a timer/counter device). Based on previous information, each device register $dr_i \in DR$, where $i = 1, 2 \cdots n : dr_i$ is a triple $\{A, W, Pr\}$, where A is an address of a device register, W is a width of a device register and Pr is a set of all device register parts $\{p_1, p_2 \cdots p_n\}$. For each part $pi \in Pr$, where $i = 1, 2 \ldots n : p_i$ is pentad $\{Ma, A, RS, O, De\}$, where Ma is a mask of the part pi, A is a type of access to the part pi, RS is a value of the part pi after system reset, O is a set of options, and De is a set of dependencies to the part pi.

The mask Ma is important during reads and writes from/to a device register where a part belongs to. The mask allows making changes to a register part without influencing other register parts.

The access type A of a part pi is important for creation of operating system routines. There are three types of access to a register part: read only access, write only access, and read write access.

The reset value RS informs an operating system to which value a part p_i is set after a reset. However, it is always a good habit to perform an initialization of a register even if the reset value is defined.

The set of options $O = \{o_1, o_2 \ldots o_n\}$ defines the values which a register part can achieve, and what effect these values have to a device (e.g. set clock division factor). The set of options O is empty if the values of a register part have no special meaning (e.g. data register or address register).

The set of dependencies $Dp = \{dp_1, dp_2 \ldots dp_n\}$ covers a problem of relations between two or more parts. For example, write to register part p_1 is allowed only if register part p_2 is set to *1*. For each $dpi \in Dp$, where $i = 1, 2 \ldots n : dp_i$ is a triple $\{Dtp, Dtm, C\}$, where Dtp is a dependence type, Dtm is a dependence time, and C is a condition which needs to be met to successfully perform operation above the register part which is under dependence. The dependence type defines what happens when a dependence condition is met. It can be a set type, a reset type, and a write type. The set type sets the part p_1 to a value of the part p_2. The reset type sets value of the part p_1 to its reset value when p_2 meets condition. The write part writes a value to the part p_1 when p_2 meets condition. The

dependence time Dtm defines when the condition has to be met. It can be before, together with, or after writing to the part that is under dependence.

D. Device interrupts

The set of interrupts $I = \{itr_1, itr_2 \dots itr_n\}$ describes all interrupt signals of a device. For all $itr_i \in I$, where $i = 1, 2 \dots n : itr_i$ is defined as an index of an interrupt in an interrupt controller device. Based on implementation, this index can refer to a predefined address of an interrupt in a program memory (e.g. 8051 architecture) or to an interrupt register of an interrupt controller, where address of an interrupt can be written by the user.

E. Platform-dependent code generation

The formal description of a processor can be used for the automated creation of the operating system platform dependent code, which can be realized as a set of routines and macros. According to the type of a register part, there can be a routine which writes data to a part or reads data from a part. Furthermore there can be a routine which reads or writes data from/to a whole register, in order to perform operations related to more parts of the same register as a single operation.

The benefit of the proposed formal description is that the developer does not need to develop these routines; they are created automatically - which saves development time.

V. Formal Description of a Device

In the previous chapter we proposed the formal description of a processor which describes Instruction set architecture and communication interfaces of devices. On top of this description we place description of a device which will use defined communication interface to create device module of an operating system.

A. Standard Devices

Processor manufacturers use devices of a same type in more processors (e.g. Analog-to-Digital Converter, Universal Asynchronous Receiver/Transmitter). Many of those devices are created under the same standard. So there is a considerable probability that same device with the same standard communication interface and behavior will be used in more processors of a different type or family.

From an operating system point of view, the device module controlling one device in one type of a processor can be reused for same device in another type of a processor with no or minimal changes.

B. Device Behavior

A device behavior describes possible configurations of a device under which a device operates. One device can change behavior partially or completely. The device changes behavior partially when the changes are applied only to a timing (e.g. change of watchdog reset interval). The behavior change of a device is complete when the set of possible states or set of possible state transitions is changed (e.g. set parity check in a USART). A more complex description of the behavior of a device is out of scope of this paper.

C. Operating system development framework

A broad spectrum of operating system modules together with the platform dependent code generation can be put together into an operating system development framework. This framework can combine a set of different processor descriptions with a set of standard operating system device modules, which results into easier development and adaptation of operating systems for embedded applications.

VI. Conclusions and Future work

Starting from the formal description (IV) we proposed a processor description based on JavaScript Object Notation (JSON) format [10] which applies mentioned theory into a practical realization. The JSON description covers the communication interface of processor's bus-internal devices and partially covers Instruction set architecture. We implemented a generator of code which generates the platform dependent code (written in C language) from the processor description.

According to the classification based on connection of devices to a processor unit we will extend the formal description to signal-internal devices and co-processing devices.

In connection with the formal description of a processor we are proposing a description of operating system modules. Furthermore, we are planning mentioned operating system development framework, which connects the formal description of a processor (including ISA description and communication interface description) and a device behavior description.

References

[1] P. Schaumont, "*A Practical Introduction to Hardware/Software Codesign*" Springer 2010, ISBN 978-1-4419-5999-7.

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

[3] B. Lamie, "*Multitasking Mysteries Revealed*", Embedded System Design 1997.

[4] N. Lethaby, "*Why Use a Real-Time Operating System in MCU Applications*", Texas Instruments Incorporated, 2013.

[5] M. Vojtko, T. Krajčovič, "*Prototype of Modular Operating System for Embedded Applications*", In Applied Electronics – 18th International Conference on Applied Electronics, Pilsen 2013, ISBN 978-80-261-0166-6, ISSN 1803-7232, p. 317–320.

[6] S. Bogan, "Formal Specification of a Simple Operating System" Ph.D. dissertation, der Naturwissenschaftlich-Technischen Fakultäten der Universität des Saarlandes, 2008.

[7] A. Štrba, "*Wireless Embedded System Powered by Energy Harvesting*" Ph.D. dissertation, Faculty of Informatics and Information Technologies at Slovak University of Technology in Bratislava, 2011.

[8] G. E. Moore, "Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april19, 1965, pp.114 ff." *Solid-State Circuits Newsletter, IEEE*, vol. 11, no. 5, pp. 33 –35, sept. 2006.

[9] ARM KEIL Microcontrollers Tools, "Getting Started", ARM Germany GmbH, 2014.

[10] ECMA international, "The JSON Data Interchange Format", Standard, Geneva, 2013.