Adaptability of an Embedded Operating System: a Generator of a Platform Dependent Code

Martin Vojtko, Tibor Krajčovič
Institute of Computer Systems and Networks of Faculty of Informatics and Information Technologies
of Slovak University of Technology in Bratislava
Ilkovičova 2, 84216 Bratislava, Slovakia
Email: martin.vojtko@stuba.sk, tibor.krajcovic@stuba.sk

***Abstract*—Adaptation of operating systems to a new processor architecture is a complicated process during which incompatible parts of an embedded operating system have to be redesigned and missing parts have to be implemented. The complications grow when there is a need to adapt operating system to completely different processor architecture as was an operating system optimised for. During our work on this problem we proposed a tool which reduces effect of these complications to a minimum. The tool uses a processor formal description file, which can act as a standard for processor manufacturers, and could help during generation of a platform dependent code of the operating system kernel. As a result of a platform code generation the adaptation time of operating system was reduced into platform independent adjustments. In this paper reader will find the recent status of work on an operating system adaptation process. A generator of platform-dependent code is described together with a framework that will help to shorten the adaptation process of any operating system.**

Keywords*—*Adaptation of Embedded Operating systems, Formal description of Processors, Source Code Generation

I. Introduction

Adaptability, the ability to easily adapt existing system to a changing environment is and will be on a great concern. In a segment of operating systems is this ability useful because the needed time for an adaptation of operating system to a new processor is an important competitive factor. An operating system with well chosen architecture is easier to adapt as the other operating systems. Therefore, we need to choose proper architecture of an operating system and propose/use a proper methodology and development tools to decrease the complexity of an operating system adaptation.

Most of the modern embedded operating systems have a kernel architecture that reduces adaptation complexity. It means that the kernel manages devices of a processor, memory of a system, and scheduling of processes and its communication [1], [2]. Other services of an operating systems such as file management or device drivers are separated from the kernel. When the operating system has to be used on a new processor architecture, mentioned parts of the kernel have to be modified while services which are out of the kernel stays mostly unchanged. Difficulty of the change depends on internal structure of an operating system kernel. The worst scenario represents a monolithic kernel, where only small change in the code can result into lasting problems. On the other hand, a modular and layered kernel organization helps to reduce management of changes in an operating system code [3]. Mostly affected part of a layered operating system is then platform dependent layer. In this layer, the code connected to the processor architecture has to be modified. Modular aspect of a kernel helps to keep changes only in modules which have to be changed without affection of other modules.

Beside structure of the kernel architecture, a well chosen methodology has great impact on the adaptation time. a methodology or in other words an adaptation process can dramatically reduce the adaptation time of an operating system. Important phases of the adaptation process are analysis of a new platform (e.g. new processor) and analysis of the needed changes in the operating system kernel. It is obvious, that more differences in architecture are found between an existing working platform and the new platform, more time the adaptation process needs.

This paper summarises information about adaptation process of a generic embedded operating system with kernel which has layered and modular architecture [3] including current status and its optimizations and automation [4]. We will present the ways and means which will allow automatic generation of the platform dependent parts of an operating system. A concept of an adaptation framework is presented in this paper. This framework will consist of a set of tools that reduces complexity of adaptation of chosen operating system.

A. Adaptation Process

An adaptation process of an operating system can be divided into three main phases. It is an analysis phase, a design phase, and an implementation phase (Fig. 1.).

In the first phase an operating system code is compared to a new processor platform. During this phase developer analyses processor datasheet and description file. The datasheet file contains information about each processor device and each processing core needed for developer. On the other hand, the processor description file contains code which can be included into platform dependent code of an operating system, but the problem is that content of this description file is not standardized and varies between individual manufacturers. The result of this phase is the list of needed changes to the operating system kernel and the list of missing parts in the kernel.

In the design phase, developer proposes solutions to incompatibilities between the kernel of an operating system and a new platform. Missing parts of the kernel are designed too. The degree of a needed change is strongly connected to the

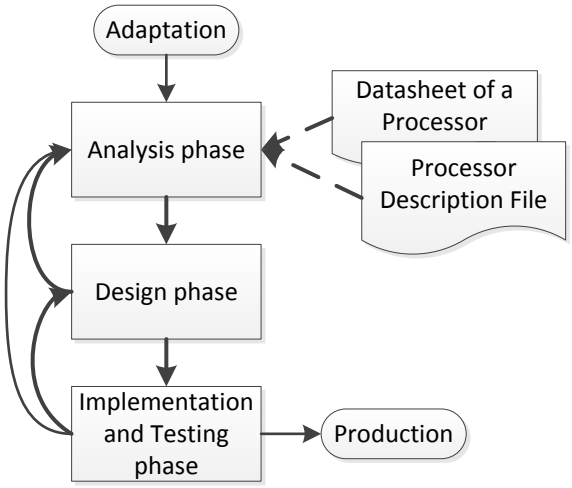

Fig. 1. Steps in adaptation process.

difference between a working platform and a new platform. As we mentioned modern kernel has layered structure ideally consisting of two layers [3]. The first layer (Platform Dependent Layer) of the kernel is mostly affected part of the kernel. An important role in the adaptation of the first layer plays a processor description file from which a platform dependent code is used [4]. The second layer (Platform Independent Layer) is affected only if in a new processor are used different or new devices which are not supported by the kernel. Next step of design phase contains the proposal of needed changes to an existing interface between the platform dependent layer and the platform independent layer because it is possible that the description file, which is the main part of a platform dependent layer, has different structure.

In the last phase developer implements proposed solutions and tests its right functioning. As can be seen in the Figure 1., each phase of an adaptation process can be re-established if defined properties, constraints and requirements do not meet during validation of the phase results.

B. Problems of an Adaptation Process

There are more issues in previously described adaptation process. First problem is with the description file of a processor. The problem is that each manufacturer has his own standard for structuring this files which leads into significant differences in the code content and the structure of this files. The result of this differences is problematic embedding of existing code into a kernel structures. Another problem is that the most of those description files are written in assembly language and/or in C programming language so developers are limited only to this preferred programming languages.

There is need for such standard which will standardise the content and structure of the processor description files and will eliminate programming language reference from it. A draft of such standard can be found in [4], where a processor formal description file acts as a computer readable datasheet. The structure of this formal description file allows us to do an automated process of the code generation in a format which is compatible with the operating system code and is written in a programming language which is preferred by a developer.

II. Related Work

As we described in previous section, the adaptation of an operating system is not a trivial task. Therefore issuers of embedded operating systems writes adaptation manuals which helps to lead developers step by step in the operating system adaptation process [5] [6] [7]. For example manual of FreeRTOS [5] advises to use an existing processor definition file for implementation of platform dependent parts. The FreeRTOS provides to user only functionality of task and memory management, the only think which has to be adapted are those services. Still this task is not always simple the difficulty of the task depends on a new processor. It is much easier to adapt operating system to a processor within the same family of processors as to adapt operating system to non supported family. Sometimes the difference can lead into reimplementation of the whole core of an operating system kernel.

Many of embedded operating systems provide to user only functionality of task switching, planing and memory allocation. This type of operating system is easier to adapt because the set of changes is reduced mostly to task switch procedure implementation. The other services of operating system such as I/O management has to be implemented by developer. Of course there exist many situations where this solution is the best solution, but in our situation we use operating system with a standard kernel architecture so proposed adaptation process provides to developer I/O management support which helps developer to concentrate on his application development.

Most of the processor manufacturers prepare for users processor definition files. In this file is information about existing devices, its registers and parts. The definition file is provided mostly as a header file written in a C programming language. The problem is that structure of definition files varies from manufacturer to manufacturer. The solution provides standardization as provides mentioned formal description of a processor.

III. Formal Description of a Processor

In a paper [4], is proposed the processor formal description (PFD). Main aim of this description is to describe processor in a form which is readable for a computer. The PFD structure is not written in programming language, that allows generation of a code for a platform dependent layer of chosen operating system and chosen programming language.

The PFD decomposes processor into one or more processing cores and a number of devices. Those cores and devices are merit of the PFD and they can be decomposed into smaller parts. Processing core is defined by Instruction Set Architecture which describes the instruction set, internal registers and operating modes. On the other hand, device is defined by a communication interface. This communication interface consists of registers, interrupt signals and signals, which are used for control and data transfer from or to device. Scope of this work is to generate a platform dependent code from description of a device communication interface that is stored in the PFD. In [4] is defined that each device

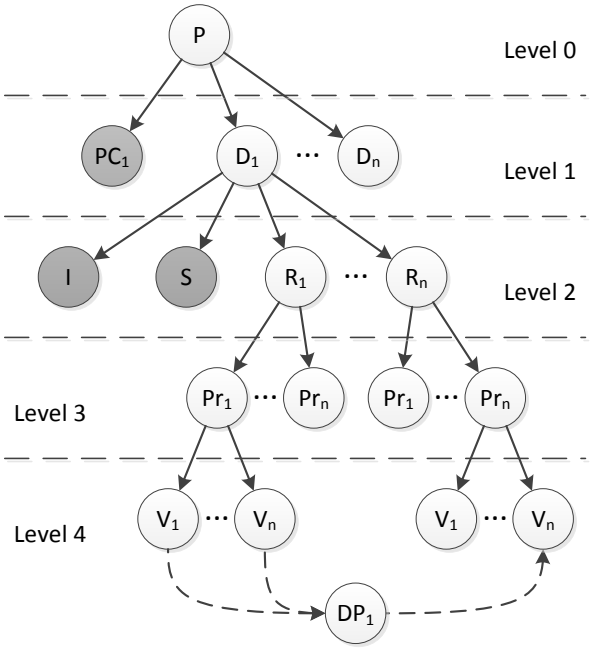

Fig. 2. Graphical visualisation of processor formal description structure. Where P is processor, PC is a processing core, D_i is a device, R_i is a device register, I is interrupt signal, S is a device signal, Pr_i is a register part, V_i is a named value of a part and DP_i is a dependence between parts.

contains a set of registers and each register contains a set of register parts. Each register can be accessed via its address defined in processor datasheet. Each processor part is placed in register at specific register bits defined in processor datasheet. In Figure 2. can be seen visualisation of processor formal description structure which is based on previously described structure of the processor [4].

IV. Code Generator

A code generator is a tool which processes chosen information from the PFD into platform dependent code of preferred language. Present version of the generator generates header file and source file in a C programming language. The structure of a processor formal description file allows simple transformation into another programming language by implementing another code generator.

A. Analysis of Description File

As an input for code generator we use the formal description of a processor [4]. This definition file is in a .json format [8] and contains description of all processor devices and processing cores. For now only devices descriptions are used for platform dependent code generation. According to the formal description specification, each device consists of registers and interrupt signals. Each register consists of register parts and each register part consists of part options and inter part dependence (Fig. 2.).

1) Device: An important information about device of processor is base address of a device and its ID. ID of a device is unique identification of a device in a processor description file and by convention it should be abbreviation of a device

```
1   #define RSTC_ADDRESS    0X0FFFFFD00
2   ...
3   #define RSTC_SR_OFFSET  0X000000008
4   #define RSTC_SR_ADDRESS 0X0FFFFFD08
5   ...
6   #define RSTC_SR_RSTTYP_MASK     0X00000700
7   #define RSTC_SR_RSTTYP_NMASK    0XFFFFF8FF
8   #define RSTC_SR_RSTTYP_PWRRST   0X00000000
9   #define RSTC_SR_RSTTYP_WDTRST   0X00000200
10  #define RSTC_SR_RSTTYP_SWRST    0X00000300
11  #define RSTC_SR_RSTTYP_USRRST   0X00000400
12  #define RSTC_SR_RSTTYP_BODRST   0X00000500
13  ...
```

Fig. 3. Selected lines of generated device definitions for Reset controller of ARM7tdmi based processor and its status register and Part defining type of detected reset signal.

name. Base address defines beginning address of a device at internal bus.

2) Register: A Register same as a device has its ID, offset address and register size. ID is defined in the same way as a device ID. Offset address defines address position in a parent device address space. Register size defines the size of register in bits (typical register size is in powers of twos).

3) Part: A Part is defined with an ID, mask and access. The mask specifies bits used by part in a parent register. Access defines how can be the part accessed. There exists three types of an access, it is write only, read only, and write and read access. (F.ex. Control register of timer/counter device can contain the part which defines whether the timer will increment or decrement a value register.)

4) Named value: A Named Value as a sibling of a part defines predefined datasheet value. Value is defined by ID and number which defines the value. (F.ex. Part which sets up direction of counting can be set to decrementing if value of direction part is set to 0 or to incrementing if value of direction part is set to 1.)

5) Dependence: A Dependence structure is useful when there exist more write or read relationships between two or more parts. Good example of dependence is write enable signal which allows to execute write to a protected part only if enable signal is set to enable state. (F.ex. A specific value will be written to a data register only if in previous step has been written write enable bit in control register).

B. Definitions Generation

Biggest part of a generated code forms definitions of register addresses, masks of register parts and values of parts. This part of generated code can be found also in other existing definition files.

1) Device Address Generation: The information about position of device in a processor address space is generated as a definition. The definition name is based on device ID (see Fig. 3. line 1).

```
1   #define set_EFC_MC_FMCN(var, value) setPart32((var), (value), EFC_MC_FMCN_NMASK)
2   #define set_EFC_MC_FWS(var, value)  setPart32((var), (value), EFC_MC_FWS_NMASK)
3   #define write_EFC_MC(var)           writeRegister32(EFC_MC_ADDRESS, var)
4   #define get_EFV_MC_FMCN(var)        getPart32(variable, EFC_MC_FMCN_MASK)
5   #define get_EFC_MC_FWS(var)         getPart32(variable, EFC_MC_FWS_MASK)
6   #define read_EFC_MC()               readRegister32(EFC_MC_ADDRESS)
```

Fig. 4. Selected lines of generated encapsulating macros for status register of reset controller device

2) Register Address Generation: If we want to be able to execute write or read command on a register we need to know the position of the register at address space. The code generator generates address definitions for each register from formal description file. Name of address definition is constructed by concatenation of a device ID and register ID. The address value is calculated as a product of device base address and register offset address (see Fig. 3. lines 3, 4).

3) Part Mask Generation: Part is placed into a register space. This space is addressed by mask. Mask definition is used to allow execution above one part of the register without affecting other parts of the register. The definition name is constructed by concatenation of a device ID, register ID and part ID. The value of the definition is chosen from part description (see Fig. 3. lines 6, 7). Beside mask a negative mask is generated to avoid calculation of this value by kernel which on the other hand increases kernel memory footprint.

4) Value Generation: Last part of definitions is definition of values. Definition name is generated as a concatenation of device ID, register ID, part ID and processed value ID. The value of definition is selected from value description (see Fig. 3. lines 8-12). As you can see in listing, the defined values are aligned with respect to a mask of the parent part.

C. Primitives Generation

Added value of generated platform dependent code is a generation of primitives which are simple functions that realize operations above managed parts and registers. In terms of granularity, there exist primitives which operates over whole register or primitives which operates over one part of a register. In terms of operation type there exist primitives which write data to a register or a part or read data from register or a part.

The main problem of code generating is optimization level of a generated code. The platform dependent layer connects only hardware with higher layer of an operating system therefore this layer is state-less and should be as thin as possible. The primitive executes only one operation (if we see this operation from higher level programming language perspective) so they should be generated as in-line functions. In other words the body of the function is placed right into a code so no procedure call instruction is used. This decision reduces procedure call overhead on the other hand the code memory footprint is larger.

The primitive generation can be merged into two steps.

In the first step are generated Set routines and $Read$ routines. $Read$ routine reads content of a register and stores it to a variable. Get routine reads value of a chosen part of a register. Then Set routines and $Write$ routines are generated. Set routine writes data to a specified part of a register to a variable without affecting data of other parts. $Write$ routine writes prepared variable with new values of parts to register. Examples of a routines definitions can be found at Figure 5. Because there can be defined registers of different sizes (8b, 16b, 32b, 64b) each routine type has to be defined in more register size versions. As a result of routine generation a four groups of routines are created. First group (Fig. 5. lines 1, 2) realizes set of value to defined part of register based on a mask of a part to a variable. Second group realizes write of a variable into a register (Fig. 5. lines 3, 4). Third group realises reading from a register part based on variable and mask of a part (Fig. 5. lines 5, 6). Last group realizes reading of a register and writing of its value to variable (Fig. 5. lines 7, 8). As can be seen at lines 1 and 2 the negation of mask is needed for a successful setting of a part value since a standard mask is used for the reading of part value.

In the second step an individual primitives are encapsulated by macro definitions. This step binds primitive calls with particular register. At Figure 4. is shown example of this macro definitions. This encapsulation helps to define full communication interface for each register of each device. This interface is then easily used as a connector to higher layer of an operating system kernel.

D. Dependence between Register Parts

Sometimes there exist situation when one part of register can be changed only if another part of same register or another register is set to defined value.

We identified that there exists three types of dependence in connection with time. It is *Before*, *After* and *Together*. The *Before* dependence exists between parts when there have to be set one part before the another part can be changed. The *Together* dependence exists only between two parts which are in a same register. The *After* dependence exist when the dependent part have to be changed before the control part.

```
1   var setPart8(var, value, !mask);
2   var setPart16(var, Value, !mask);
3   void writeRegister8(address, var);
4   void writeRegister16(address, var);
5   value getPart8(var, mask);
6   value getPart16(var, mask);
7   variable readRegister8(address);
8   variable readRegister16(address);
```

Fig. 5. Selected lines of generated function definitions.

In terms of operations there exist more types of dependence. First of all it is *Set/Reset* dependence where the value of a dependent part is changed to predefined value without direct write operation to a part. (F.ex. Reset signal of a device). Second type of dependence is *write* dependence which allows to write data to a dependent part. A *activation* dependence is used when some part of register is used in more defined contexts (F.ex. the value of on part can be used as a clock divider if another part is set to one or as a multiplier if another part is set to zero). A *mutual exclusion* dependence refers to a parts where one part can be set only if another part is cleared and vice versa.

Existence of dependence complicates generation of code and therefore the proper description of this dependence is important in generated code. This proper description is our recent work.

V. Results of Code Generation

We tested generator on five processors:

- ARM7tdmi based 32b processor AT91SAM7S256 [9].
- AVR32 based 32b processor AT32UC3L032 [10].
- PIC10 based 8b processor PIC10F200 [11].
- PIC32 based 32b processor PIC32MX [12].
- Atmel AVR8 based 8b processor XMEGA AU [13].

In the first step of the testing we prepared PFD for every mentioned processor. This process needed analysis of processors datasheets. In the second step we processed those files by the generator. In the Table I. a quantitative results of this generation are summarised. First part of table summarises

TABLE I. Number of Processed Devises, Registers, Parts and Options, and Generated number of source code lines.

Processor	AT91SAM	AT32UC	PIC10	PIC32	XMEGA
Architecture	32	32	8	32	8
Devices	20	25	3	19	18
Registers	337(16.9)	432(17.3)	8(2.7)	85(4.5)	125(6.9)
Parts	2342(6.9)	1327(3.1)	28(3.5)	453(5.3)	241(1.9)
Options	3428(1.5)	1298(0.9)	68(2.4)	644(1.5)	99(0.4)
Definitions	8806	4841	143	1739	849
Primitives	3008	2468	50	1006	707
Sum	11814	7309	193	2745	1556

number of processed devices, registers, parts and options for each processor. Average number of registers in device, parts in register and options in part is calculated in brackets. Second part of table shows generated definitions and routines. Processor PIC10F200 is one of the smaller processor and therefore number of generated routines is also smaller. This processor is way too small for such program as is operating system. On the other hand other processors are much bigger and they can be used with an operating system.

VI. OS Adaptation Framework

In a conclusion to this paper we want to present a concept of OS adaptation framework. This framework will support the OS adaptation process by set of services and databases that help manage operating system adaptation. In a Figure 6. is presented concept of the framework. The framework services are:

- processor formal description describing,
- processor formal description validating,
- platform dependent code generating,
- device module description modeling,
- processing core description modeling,
- device module to communication interface mapping,
- core module to communication interface mapping,
- device or core module validating,
- code implementation and/or generation, and
- selection of platform dependent code and modules code.

Beside framework's services the framework uses 3 databases:

- database of processor formal description,
- database of operating system modules, and
- database of operating system source codes.

1) Description of a Processor: The framework helps during preparation of processor formal description files (PFD) from datasheets. Prepared PFD is then validated and sent to a database. Stored description can be then used by generator to produce platform dependent code of operating system where from description of the processor can be generated platform dependent layer of an operating system. Descriptions are also used for mapping of operating system modules to communication interfaces of devices and processing cores.

Inserted description of processor will be decomposed into devices and cores. Each identified device will be inserted to a database under version which will guarantee that device is not presented in the database as a duplicity (We assume that each device and core will have unique identification).

2) Description of a Module: Since the formal description of a processor covers only a communication interface of a device or processing core there is still need for development of an OS module which manages this device or processing core. The formal description will be used for mapping of module to communication interface. Resulting model of an operating system module will be validated and then inserted to a database of operating system modules. Each module will have unique identification. The documentation to a model will be compulsory part of the design.

3) Module Code Generation/Implementation: From module description developer will have to implement operating system module. Some parts of module can be generated automatically as a skeleton of the module which will help to developer during implementation. Implemented module is then stored in a database of operating system source codes together with unique version, linkage to parent module description and compulsory documentation.

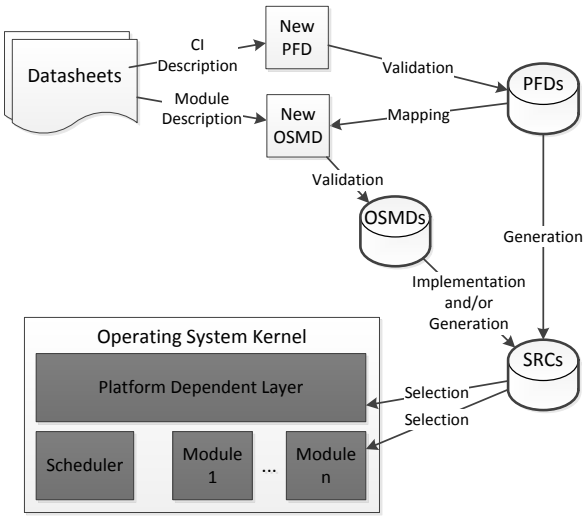

Fig. 6. Set of adaptation framework services, where *CI* is an communication interface of a device, *PFD* is an processor description file, *OSMD* is OS module description, *PFDs* is a database of processor formal descriptions, *OSMDs* is a database of OS module descriptions and *SRCs* is database of OS source codes.

4) Selection of Operating System Parts: As a part of a system design, developer of embedded system will have access to database of operating system source codes. From this database developer will choose platform dependent layer and select compatible device and processing core modules for chosen processor. He can also add/describe/implement missing modules or other parts.

A. Database of Descriptions

Full formal description files of processors can be found in a database of descriptions. Those files will be also decomposed into separate devices and processing cores. The problem can rise when same device exists in a more processors so this situation has to be solved by device unique identification. A protection from processor description duplication has to be created which does not allows upload again existing processor description. If existing processor was revised by manufacturer there will be possibility to revise formal description too.

The database can be used also during creation of new description files where developer can search existing devices and processing cores in database and include them into new description, which will reduce duplicity and description time.

B. Database of Modules

This database will store descriptions of operating system device and processing core modules. As in database of descriptions this database will use unique identification and versioning. Descriptions from database can be used for describing similar device modules as working examples. Existing modules then can be reused in module description which will reduce description time.

C. Database of Sources

In the database of sources will be stored unique versions and ports of operating system source code. Developer will be able to select platform dependent code for selected processor and he will be able to select module source codes based on description of a module, because there will be presented more versions of the module.

VII. Conclusions

By standardization of processor description a processor formal description file which acts as a datasheet for a computer was created. This formal description will have same structure for every processor therefore platform dependent code can be generated by code generator. By usage of processor formal description and generator we reduced complexity of platform dependent code adaptation which acts as biggest part of operating system which have to be changed during adaptation. As we covered by this generator only devices of processor we will extend this generator with code generation for processing units of a processor. A inter part dependence issues will be covered also. There are also issues with task switch routine because this routine is not trivial and mostly is written in assembly language. The generator of the platform dependent code will be a part of an operating system adaptation framework which gather tools that helps to adapt any operating system to any processor.

Acknowledgment

References

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

[2] J. J. Labrosse, J. Ganssle, and e. a. Robert Oshana, *Embedded Software: Know It All (Newnes Know It All)*. Newnes, 2007.

[3] M. Vojtko and T. Krajcovic, "Prototype of modular operating system for embedded applications," in *Applied Electronics (AE), 2013 International Conference on*, Sept 2013, pp. 1–4.

[4] M. Vojtko and T. Krajovi, "Adaptability of an Embedded Operating System: a Formal Description of a Processor," in *10th International Joint Conferences on Computer, Information, Systems Sciences, and Engineering*, Online, Dec. 2014, pp. 1–4.

[5] *The FreeRTOS Project*, 2011, http://www.freertos.org/.

[6] *TinyOS*, 2011, http://www.tinyos.net.

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

[8] ECMA international, *The JSON Data Interchange Format*, ECMA international, Geneva, 2013, http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

[9] *AT91 ARM Thumb-based Microcontrollers: AT91SAM7S256*, Atmel Corporation, 8 2010.

[10] *32-bit Atmel AVR Microcontroller AT32UC3L032*, Atmel Corporation, 1 2012.

[11] *PIC10F200/202/204/206*, Microchip Technology Inc., 9 2014, rev. F.

[12] *PIC32MX Family Data Sheet*, Microchip Technology Inc., 9 2008.

[13] *8-bit Atmel XMEGA AU Microcontroller*, Atmel Corporation, 4 2013.