Effective Multiplatform Firmware Update Process for Embedded Low-Power Devices

Ondrej Kachman*
Department of Design and Diagnostics of Digital Systems
Institute of Informatics
Slovak Academy of Sciences
Dúbravska cesta 9, 845 07 Bratislava, Slovakia
ondrej.kachman@savba.sk

Abstract

Low-power devices can nowadays be found in many systems of collaborating computational devices. They are used in wireless sensor networks, cyber-physical systems, smart systems, etc., and their numbers can reach hundreds. Each system may include devices based on many different platforms in their structure. The devices may be often battery powered and physically inaccessible. It is almost a necessity to enable firmware updates for these devices. Firmware updates are used to add features to firmware, fix problems or change its functionality completely. Battery powered devices with constrained resources require energy efficient firmware update process. We have developed a novel multiplatform process for differential updates of embedded low-power devices. It is independent of the network protocols used, reduces data shared between devices during an update and saves energy on the memory operations required by target devices to finish the update. It supports multiple configurations to adapt to different devices and platforms. The presented solution is suitable for modern intelligent systems that use low-power devices.

Categories and Subject Descriptors

C.3 [**Special-purpose and Application-based Systems**]: Microprocessor/microcomputer applications; D.1.1 [**Programming techniques**]: Applicative (Functional) Programming; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*portability, version control*

Keywords

firmware, remote update, reprogramming, low-power, embedded

1. Introduction

Modern systems with low-power sensor and actuator devices in their structure may be composed of hundreds of such devices. The devices usually communicate wirelessly as they may be physically inaccessible. Firmware of these devices developed under test conditions may fail once deployed in the field. In such cases, physically collecting each device and reprogramming it with a programming device is undesirable. This can be solved by inclusion of an update module in the original firmware version. The update module can be a part of firmware or a device's bootloader and is responsible for reprogramming of firmware. Firmware updates do not have to be used only for bug fixes, they can also be used to add, remove or modify firmware features. Frequent, incremental reprogramming of a battery powered device could deplete its battery fast and shorten its lifespan significantly. Low-power devices in some systems are expected to run for years on a single battery. An update mechanism developed for these devices should aim to reduce energy consumption of the update process.

This extended abstract presents a multiplatform firmware update process for low-power devices. It uses differential updates to reduce the amount of data shared on a network during an update. Differential updates use delta files, binary files that encode differences between two files. Wireless interfaces of the low-power devices are usually their most energy-hungry components, so the update data reduction helps to save energy. The delta file format is designed to be decoded with linear complexity so the CPU of a device does not waste energy on some complex decompression. Another reduction to energy consumption is achieved by the proposed update module. Erase is the most expensive operation for NAND flash memories. Our update module, Patch module, rewrites altered pages at most once per update. This reduces energy consumption and preserves flash memories that are limited to approx. 10 000 erases. Multiplatform nature of the process is achieved through focus on the object files and no alterations to the code and optimizations generated by compilers.

The rest of the extended abstract is organized as follows. Section 2 describes existing solutions in the problem area. Short description of the design of a novel multiplatform firmware update process is in the section 3. Evaluation of the designed update process is described in section 4. Section 5 provides a short conclusion to this paper.

---

*Recommended by thesis supervisor: Assoc. Prof. Ladislav Hluchý
To be defended at Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava on [To be specified later].

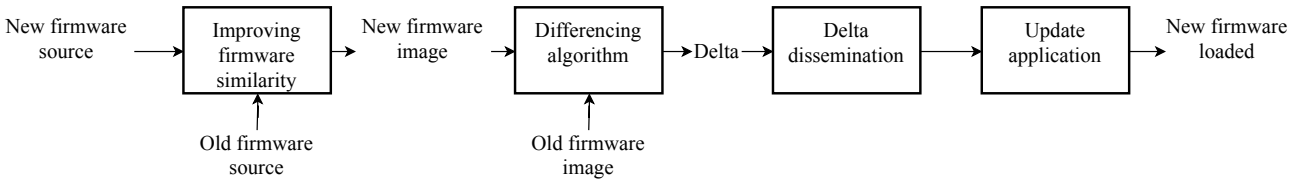

Figure 1: Four main stages of an update process [9].

2. Related Work

The area of remote firmware updates can be split to 4 main stages. These stages are shown in Figure 1. Some works in this area are focused on the whole process while some target only some of the stages. The research started with the development of wireless sensor networks (WSNs) and TinyOS sensor operating system in early 2000's. One of the most important early works is the Deluge [6] dissemination and reprogramming protocol. It is primarily focused on the network dissemination stage to reduce number of retransmissions and control messages shared between sensor devices. It detects changed blocks of fixed-size blocks and disseminates them through the whole network. Many works in the area built on this and improved the other stages of the update process.

2.1 Improving Firmware Similarity

Similarity improvement stage aims to produce as similar firmware binaries as possible. More similar binaries for old and new firmware will later result in a smaller delta file, thus less update data. There are multiple approaches to this problem and some of the existing solutions may use only one while others may consist of multiple. The main approaches are:

- **Changes to compilers** - Compiler can be edited to preserve register allocations for variables present in multiple firmware versions [11, 5]. The disadvantages of this approach are that it is highly platform specific and counters compiler optimizations.
- **Handling memory shifts** - New or modified sections can be placed to a new memory location and referenced using proper instructions [13, 16, 15]. Functions that grow will not cause memory shifts. The approach is also platform specific as it inserts CALL or JUMP instructions into the source code.
- **Handling data shifts** - Data in RAM memory can also be reorganized in every firmware version. It is possible to change layout of the RAM sections to counter these changes [12, 15]. However, these works cannot guarantee that data will not shift.
- **Handling relocatable code** - Instructions that reference various memory locations (relocatable entries) can be set to the same value and later filled in by a loader [2, 3]. Making these entries the same helps to detect more common code sequences.
- **Memory fragmentation** - Memory can be fragmented to provide sections with space to grow and shrink [10]. The space is called a slop region. The disadvantage of this approach is less data space left on a device.

2.2 Differencing Algorithm

A differencing algorithm compares an old firmware binary with a new firmware binary. It detects common sequences and new sequences. Using these sequences, it encodes a delta file. Delta files consist of delta operations that are translated to a physical memory operations on a target devices. Lighter differencing algorithms are block based - easier to implement but require more update data [6, 14]. More complex differencing algorithms are byte or word-based [3, 7, 4]. Basic operation types for delta files are:

- COPY - moves existing sequences
- ADD - adds new sequences
- INSERT - inserts a single data unit
- PAD - pads memory with a specified data unit

2.3 Delta Dissemination

The dissemination stage is heavily network focused and a research area of its own. The task is to transfer current delta file to all target devices in the network securely and reliably. A lot of research went into design of the protocols specialized for firmware reprogramming like Deluge [6, 13, 12]. In recent years, technology standards and standard protocols have been developed even for the systems that use low-power devices. The research shifted towards upper layers and determination of the most effective update trees [1, 17]. The modern research for the other three stages of the firmware update process should be agnostic to the networking technologies and protocols used to propagate the update data.

2.4 Update Application

The update module on a target device is responsible for application of an update. The operation should not take long and waste energy on memory operations. Also, rewriting the same page of a flash memory multiple times during an update reduces the memory's durability. Standard approach is to rebuild firmware in an external memory, reset the device and load the firmware from the external memory in bootloader [2, 3]. Modern devices may not have an external memory and must use their internal program memory to rebuild the firmware image. This is sometimes referenced as on-the-fly update [15, 8]. During on-the-fly update, firmware can be rebuilt completely in another part of the program memory or pieced together right in its location using swap space. While the first approach is more reliable, the latter requires less memory space and can be more suitable for devices with limited-size memories.

3. Design of a Multiplatform Solution

This section describes our multiplatform firmware update process. It is based on the state-of-the-art analysis and

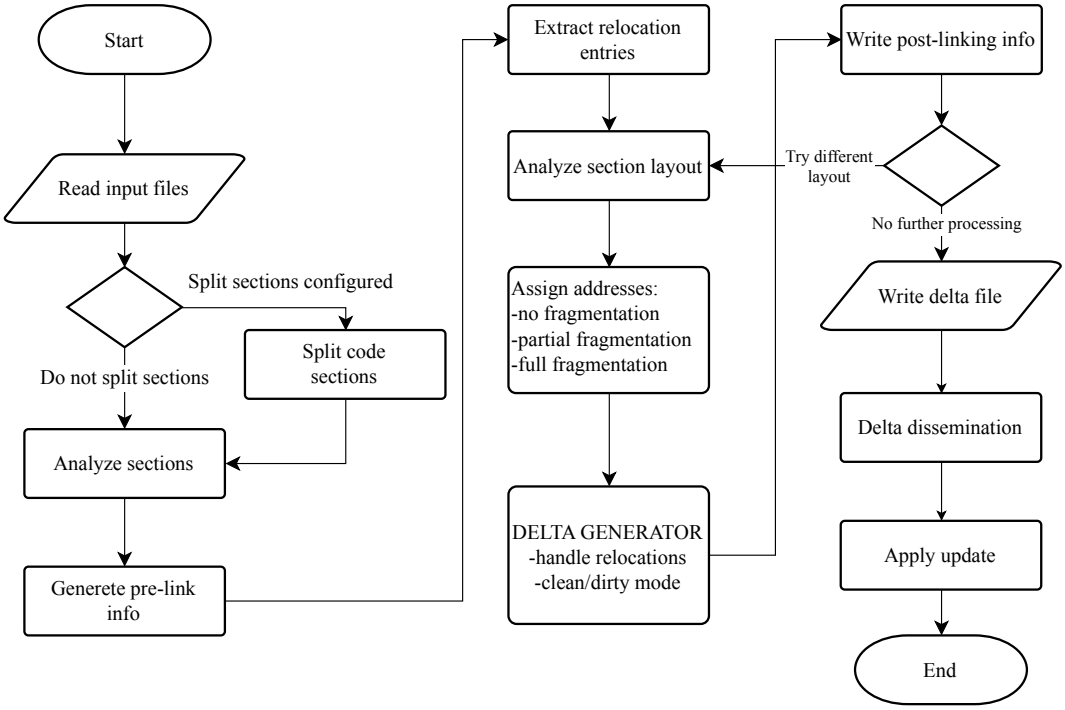

Figure 2: General flowchart of the designed firmware update process.

introduces new configurations and optimizations. During the analysis stage, it was identified that there are no multiplatform solutions. The main objective was to design and evaluate a multiplatform approach that would include all the best practices and ideally improve them. The main hypothesis regarding the multiplatform nature of the designed solution was:

If the remote firmware update process does not alter the source code of a firmware but rather operates on object files and various other data, then it has the full potential for multiplatform use.

The general flowchart of the designed solution is shown in Figure 2 and the following subsections describe the processes, configurations and decisions that are included in the solution.

3.1 Reading Input Files and Analyzing Code Sections

The process starts with analytical algorithms. The first one reads, analyzes and possibly slightly alters input files. The input files are all firmware object files produced by the used compiler. These files include optimized code and data sections along with additional sections that contain various information. The algorithm is responsible for identification of the code sections and their organisation. It supports two configurations:

- **Default sections** - The algorithm identifies default sections. Some of these sections may be defined in source code by firmware developers. They can be split to modules and group multiple functions into the same sections. This configuration gives the developers more control over defined sections and modules.
- **Split sections** - The algorithm places every firmware function into its own section that can be later put to a specified physical address by the linker. This configuration automates the process and does not require edits to the source code of the firmware.

3.2 Collecting Pre-linking Information

This algorithm works with the code sections prepared in the previous step and with the post-linking information of an old firmware version if there is one. If there are no previous versions, the only data required are the section sizes. If there is a previous firmware version, the following additional data are collected:

- New, removed, modified sections
- Size change of the modified sections
- Memory shifts caused by modified sections
- Section order changes
- Slop region sizes

3.3 Extracting Relocation Entries

This algorithm is adopted from [2] and modified. In order to make firmware images more similar, relocation entries that reference memory locations can be set to the same value. Our algorithm only resets the changed entries and encodes them as INSERT delta operations before delta optimization. In [2], the entries were compressed and added as metadata to delta files. Our algorithm managed to reduce the data required but the approach did not show positive results.

3.4 Analyzing Section Layout

Based on the pre-linking information, this algorithm detects any changes in the section order compared to a previous version. Changes can be caused by the source code

refactoring or addition of new sections. The order is corrected to counter the memory shifts of the unchanged data.

3.5 Assigning Section Addresses and Linking

This algorithm assigns physical addresses to the code sections, generates a linker command and runs the linker. While assigning the section addresses, three methods are supported [8]. The algorithm can be configured to use any of these methods:

- **No fragmentation** - There are no slop regions provided to sections
- **Partial fragmentation** - The most edited sections are provided with a slop region
- **Full fragmentation** - All sections are provided with a slop region

Slop regions significantly reduce memory shifts. However, if the firmware will not be updated many times, they are undesirable as they fragment memory and reduce data space. If there are many incremental updates, firmware can be defragmented once the updates are done. It costs additional operations but the firmware is cleaned up and data space expanded.

3.6 Delta Generator

Delta generator (DG) [7] is our original differencing algorithm. It is a word-based algorithm that supports variable word size. NAND flash memories are currently the most used program memories for low-power devices. Word is the smallest write unit of these memories. Word size can vary between different hardware platforms so the variable size word support is necessary for a multiplatform solution.

DG detects matching and non-matching segments of a firmware. Matching segments are left in their memory locations. For each non-matching segment, DG attempts to find matching sequences in the different locations of an old firmware version. Once complete, DG prepares the lists of common and changed code sequences. These are encoded as delta operations. DG supports following operations:

- SKIP - skips over an unmoved matching sequence
- COPY - moves matching sequence to a new location
- ADD - inserts a new sequence into the memory
- INSERT - inserts a single word or a relocation entry

After the operations are generated, DG tries to optimize out some small and redundant operations by merging them into ADD operations. When finished, the operations are encoded into their binary form from the lowest target address to the highest address. Merging these binary data, calculating a CRC code for them and appending the code to the end results in the final binary file - the delta file (Figure 3). This file is ready for the network dissemination stage.

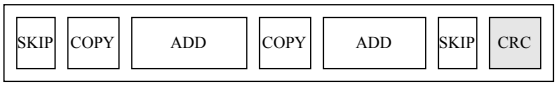

Figure 3: Simple representation of the delta file structure.

3.7 Write Post-linking Information

With the delta file prepared for the network dissemination, it is important to store some data about the current firmware version for future updates. Physical addresses assigned to the sections are stored along with the info about current size of the assigned slop regions. At this point, the process can return to the section order analysis algorithm and try out a different addressing method. This is used to compare, how the different configurations influence the delta file size. The delta file is then passed to the application that starts the update dissemination stage.

3.8 Delta Dissemination

Our process was designed to be agnostic to a networking protocol used. Delta files generated by DG can be disseminated using any standard protocol. It requires a server application that will connect to the target devices and transfer the delta files to them. Target devices must implement the necessary communication and support the used protocol. Dissemination stage is out of scope of this work.

3.9 Application of an Update

Patch module is our update module for target devices. It is designed to decode the delta files and to apply delta operations in the program memory. Decoding process has linear complexity so the Patch module does not waste CPU time and device's energy. It also rewrites each program memory page at most once per update to save energy on erase operations. Patch module has two stages:

1. Handle overlapping delta operations
2. Process delta operations

Our process is designed to apply an update on-the-fly and right in the space reserved for the firmware. This means that some data could be rewritten before a COPY operation would move them to another location. During the first stage of the Patch module, the algorithm iterates through the delta file and lists any COPY operation that has its data rewritten by any of the previous operations. The data that these COPY operations use are copied into a swap space that must be reserved on the target devices. The affected COPY operations are then rerouted through the swap space.

During the second stage, the Patch module starts to rebuild the firmware. It executes operations as they are stored in the delta file, but the pages are first reconstructed in the RAM memory. Once a page has been reconstructed, it is physically erased and written back from the RAM. No page has to be erased twice this way. Overlapping operations use data from the swap space. Once the update is finished, swap space must also be erased. After the update is finished, the Patch module can start the firmware itself or reset the device and let bootloader start the ex-

ecution of the new firmware version. This concludes the whole firmware update process.

4. Evaluation

We evaluated the designed firmware update process on three different hardware platforms. All of the tested platforms are used in low-power systems but have very different specifications. The experiments aimed to show that our solution can be applied to these platforms without implementation of any platform specific code. Also, the successful results confirm the main hypothesis. We used object files in ELF object file format and GCC compiler ported for each platform. The tested platforms are:

- 8-bit ATmega32u4 microcontroller from Atmel
- 16-bit MSP430 microcontroller from Texas Instruments
- 32-bit ARM Cortex-M4 processor on a bluetooth low-power system-on-chip from Nordic Semiconductor

For ATmega platform, we used a simple custom firmware. For MSP430 and ARM platform, we used example firmware from software development kits provided by the manufacturers. The changes made to firmware were:

1. Changed value of a constant
2. Added a new section
3. Changed a section
4. Removed a section
5. Added multiple sections
6. Changed multiple sections
7. Removed multiple sections
8. Changed order of some sections
9. Added and removed multiple sections
10. Added, changed, removed and reordered multiple sections

These test cases are the same for each tested platform. We compared our DG delta file size to differencing algorithms RMTD [4] and R3diff [3]. Delta file size percentages compared to the full firmware image size for all platforms are shown in Figures 4 - 6. DG can reduce delta file size by more than 90% compared to the full firmware image and also improves over existing methods by 5-50%. Additional algorithms for improved firmware similarity and various configurations further shrink the delta size by up to another 50%. Delta generator is able to produce delta files with just 25% of the size when compared to other methods (for example case 5 on ARM, Figure 6). For all of the change cases, the delta files successfully reconstructed the desired firmware image and did not erase any memory page more than once per update. This reduces energy consumption during an update significantly.

5. Conclusion

This extended abstract describes a novel multiplatform firmware update process. Its universal use for different hardware platforms is its main contribution. It also introduces various new configurations, new methods to handle and organize firmware sections, an update module that preserves memory durability of the low-power devices and a new differencing algorithm. With all of its algorithms, the designed solution improves over existing solutions. It was tested on three different platforms and showed mostly positive, in some cases even significant results. The presented update process is suitable for the modern networked systems that use low-power devices in their structure.

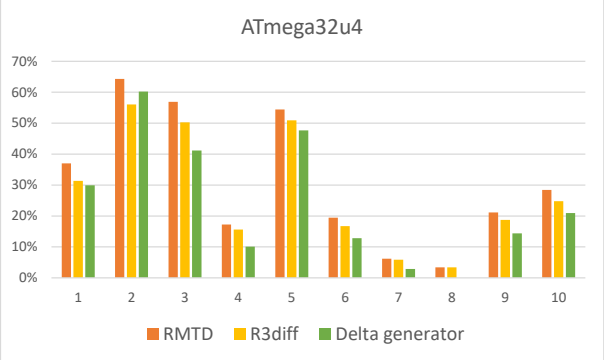

Figure 4: Delta file size percentages for the tests on the ATmega.

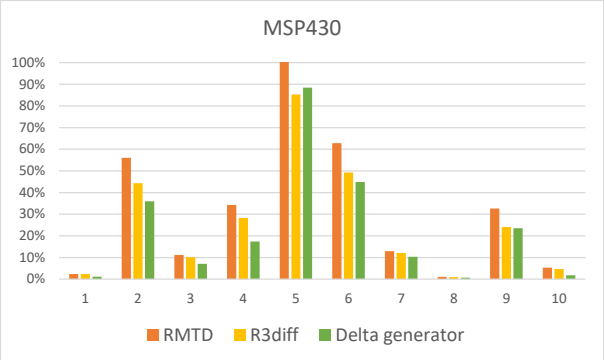

Figure 5: Delta file size percentages for the tests on the MSP430.

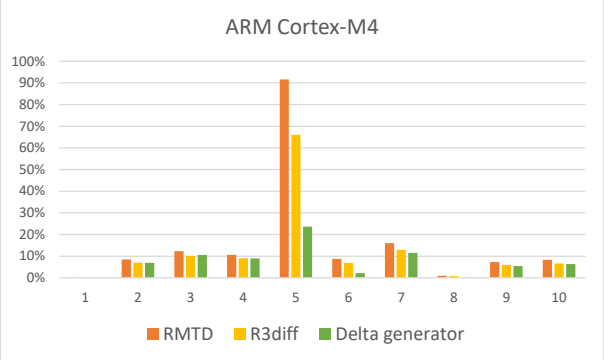

Figure 6: Delta file size percentages for the tests on the ARM Cortex-M4.

Acknowledgements. This work has been supported by Slovak national project VEGA 2/0192/15 and by the ECSEL Joint Undertaking (JU) under grant agreement No 737434.

References

[1] H. Asahina, I. Sasase, and H. Yamamoto. Efficient tree based code dissemination and search protocol for small subset of sensors. In *2017 IEEE International Conference on Communications Workshops (ICC Workshops)*, pages 283–288, May 2017.

[2] W. Dong, Y. Liu, C. Chen, J. Bu, C. Huang, and Z. Zhao. R2: Incremental reprogramming using relocatable code in networked embedded systems. *IEEE Transactions on Computers*, 62(9):1837–1849, Sept 2013.

[3] W. Dong, B. Mo, C. Huang, Y. Liu, and C. Chen. R3: Optimizing relocatable code for efficient reprogramming in networked embedded systems. In *2013 Proceedings IEEE INFOCOM*, pages 315–319, April 2013.

[4] J. Hu, C. J. Xue, Y. He, and E. H. . Sha. Reprogramming with minimal transferred data on wireless sensor network. In *2009 IEEE 6th International Conference on Mobile Adhoc and Sensor Systems*, pages 160–167, Oct 2009.

[5] Y. Huang, M. Zhao, and C. J. Xue. Wucc: Joint wcet and update conscious compilation for cyber-physical systems. In *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 65–70, Jan 2013.

[6] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, pages 81–94, New York, NY, USA, 2004. ACM.

[7] O. Kachman and M. Balaz. Optimized differencing algorithm for firmware updates of low-power devices. In *2016 IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 1–4, April 2016.

[8] O. Kachman and M. Balaz. Configurable reprogramming methodology for embedded low-power devices. *IFIP Advances in Information and Communication Technology*, 499:211–219, 2017.

[9] O. Kachman and M. Balaz. Firmware update manager: A remote firmware reprogramming tool for low-power devices. In *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 88–91, April 2017.

[10] J. Koshy and R. Pandey. Remote incremental linking for energy-efficient reprogramming of sensor networks. In *Proceeedings of the Second European Workshop on Wireless Sensor Networks, 2005.*, pages 354–365, Feb 2005.

[11] W. Li, Y. Zhang, J. Yang, and J. Zheng. Ucc: Update-conscious compilation for energy efficiency. In *Wireless Sensor Networks, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[12] R. K. Panta and S. Bagchi. Hermes: Fast and energy efficient incremental code updates for wireless sensor networks. In *IEEE INFOCOM 2009*, pages 639–647, April 2009.

[13] R. K. Panta, S. Bagchi, and S. P. Midkiff. Zephyr: Efficient incremental reprogramming of sensor nodes using function call indirections and difference computation. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, pages 32–32, Berkeley, CA, USA, 2009. USENIX Association.

[14] J. Qiu, S. Li, and B. Cao. Repage: A novel over-air reprogramming approach based on paging mechanism applied in fog computing. 2018, 2018.

[15] N. B. Shafi, K. Ali, and H. S. Hassanein. No-reboot and zero-flash over-the-air programming for wireless sensor networks. In *2012 9th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON)*, pages 371–379, June 2012.

[16] C. Zhang, W. Ahn, Y. Zhang, and B. R. Childers. Live code update for iot devices in energy harvesting environments. In *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–6, Aug 2016.

[17] Z. Zhao, J. Bu, W. Dong, T. Gu, and X. Xu. Coco+: Exploiting correlated core for energy efficient dissemination in wireless sensor networks. *Ad Hoc Networks*, 37:404 – 417, 2016.

Selected Papers by the Author

O. Kachman, M. Baláž. Optimized differencing algorithm for firmware updates of low-power devices. In *Formal Proceedings of the 2016 IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits and Systems, DDECS 2016*, Kosice, Slovakia, 2016. IEEE.

O. Kachman, M. Baláž. Firmware Update Manager: A remote firmware reprogramming tool for low-power devices. In *Proceedings - 2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuit and Systems, DDECS 2017*, Dresden, Germany, 2017. IEEE.

O. Kachman, M. Baláž. Configurable reprogramming methodology for embedded low-power devices. In *Technological Innovation for Smart systems*, volume 499 of *IFIP Advances in Information and Communication Technology*, pages 211-219. Springer, 2017.