Configurable Reprogramming Methodology for Embedded Low-Power Devices

Ondrej Kachman, Marcel Balaz

Configurable Reprogramming Methodology for Embedded Low-Power Devices

Ondrej Kachman, Marcel Balaz

Institute of Informatics, Slovak Academy of Sciences, Dubravska cesta 9
84507 Bratislava, Slovakia
{ondrej.kachman, marcel.balaz}@savba.sk

Abstract. The embedded low-power devices are very important part of any smart system. With the large amounts of sensors and actuators used, it is a good practice to implement remote reprogramming capabilities into the firmwares of these devices. This paper presents a new configurable reprogramming methodology that can be applied to various platforms. It is built on the best reprogramming practices while giving developers more control over firmware outline, updated functions and modules. It also refers energy efficiency, as the data shared over the network and memory operations on the devices are minimal. The multiplatform capabilities make this scheme ideal for smart systems.

Keywords: reprogramming, embedded, low-power, configurable, over-the-air

1 Introduction

The area of remote and efficient reprogramming of the embedded devices has been researched since the introduction of the low-power devices and the wireless sensor networks. Various reprogramming methods have been developed over the past 15 years. The main goal of these methods is to replace the old firmware version with the new one while keeping the procedure energy efficient and secure. Recent progress in the internet of things technologies, cyber-physical systems and smart systems requires these methods to be evaluated and adjusted to the new trends in these areas.

The research topic of this paper is a configurable reprogramming methodology for low-power devices, as these devices are widely used in the modern intelligent systems, including smart systems. The main advantage of the solution that this paper will present is its platform independency. The configurability (see chapter 4.2) of our solution will provide programmers with more control over the outline of the updated firmware and we present experiments that show how it can help to share less update data on the network and spare program memory with less operations executed.

2 Relationship to Smart Systems

Smart system is a general term for a system that collects information, processes it and acts based on the results of the data analysis. Smart systems evolved from the simpler systems capable of sensing and basic actuation into the systems with perception superior to a human being. The design of a smart system includes cloud services and networking technologies, choices of the appropriate hardware and software components and their implementation [1]. The sizes of smart systems can be very different depending on their function. They range from nano-scales to meter-scales. Smart systems can be applied in many sectors, for example healthcare, automotive industry, retail, building management, military, etc [2]. Low-power embedded devices are usually used for sensing, actuation a control.

An embedded device used in a smart system is preloaded with a default firmware. Even the thorough testing may not detect all the errors that this initial firmware has. Once the devices are used in the real environment in great numbers, an identified firmware error may cause a long system downtime until the problem is fixed and the system can be considered reliable again. The over-the-air update methods are developed to update the faulty firmware as fast as possible. These methods can be focused on various sub problems, closely analyzed in the chapter 3.1. The new challenge for these methods regarding the smart systems is their adaptability to the devices based on various platforms that form a smart system. Most works in the area of remote reprogramming are focused on the wireless sensor networks (WSNs) and the TinyOS operating system [3][4]. Most WSNs are homogenous, using devices based on the same platform for data collection. Smart systems and may use different devices for sensing, different devices for networking and different devices for presentation of the information to the users of the system.

To create a multiplatform reprogramming method suitable for the smart systems, there should be no alterations to the compilers or the instructions of the linked firmware images, as these alterations are platform specific. Jump, branch and relative call instructions, that are often altered by these methods, may have different operation codes for each platform. Also, not every platform may support the same instruction set and relocation types. The ideal target for a platform independent solution are the object files, the product of compiler, and the executable file, the product of the linker, later transformed into the binary representation of the firmware.

Once a firmware fault is discovered, the developers may estimate if the fix will require one or more updates. In the case of a single update, it is not required for a device to change the outline of its firmware functions, it can just apply a simple update script. If the problem fix requires incremental updates and debugging, it is good to prepare the device for these patches. An example is to put every function that requires an update to its own memory section [5]. The amount of the data shared on the system's network must be as small as possible so it will not affect the other parts of system. Different approaches may be the most suitable for the different parts of the smart system.

3 State of the Art and Related Work

This section analyzes the most important principles and works in the area of the remote reprogramming for embedded devices. Analyzed papers serve as a base for our methodology that we present in the chapter 4 and evaluate in the chapter 5.

3.1 Main Challenges for the Remote Reprogramming Methods

The methods for remote reprogramming of the low power devices are mainly focused on the 4 main sub problems [6]:

Improving firmware similarity. Previous works used altered compilers [7], object file alterations [8] and custom linker directives [5] to make the firmware images as similar as possible.

Generating delta files. Many differencing algorithms have been developed for various networks of embedded devices [9] [10]. These algorithms are designed to generate very small delta files (deltas, patches). Smaller size improves the energy consumption of the wireless interfaces and prevents the network from congestion or desynchronization.

Network dissemination. Network protocols are responsible for the delivery of the delta files to the target devices. In the past, many protocols have been designed specifically for over-the-air updates [3]. Smart systems may take advantage of the newer and standard protocols used for the low-power devices. The integrity and security are also a concern in this area. Delta files are split into the number of packets protected by CRC codes and error correcting codes [11]. Network security of the smart systems and the dissemination of the delta files is out of scope of this paper.

Update execution. The last problem deals with the implementation of the update procedure on the target devices. Reprogramming code is often embedded into the bootloader [4] or resides in its own reserved space [10]. The updates of this code are very rare and critical.

3.2 Taking Advantage of the Standard ELF File Format and Relocatable Entries

Object files are the product of compilers. Probably the most popular format of these files is the executable and linkable format, ELF [12]. This format is supported by the variety of compilers, including the well-known GCC and its ports for embedded platforms. The file in this format is also the output of the linker and is later used to create a binary image of a firmware. Embedded systems use the following formats:

- Relocatable file – created by a compiler, relocatable entries are not resolved
- Executable file – created by a linker, all relocations are resolved

The process is shown in the Fig. 1. Relocatable files include unresolved relocations, for example calls, jumps, branches, etc., and these entries are resolved during linking. The authors of [4] managed to reduce the data shared on the network by setting all relocations to the same value for each firmware, making different versions more similar. This allows the differencing algorithm to generate very small

deltas. The real relocation values are propagated as a metadata with the delta files and written to their positions by a loader on the target device.

The authors of [8] alter ELF files in various ways. New functions are placed at the end of the program memory and calls to them are edited. This approach also changes the way sections for initialized and uninitialized variables are allocated in the RAM, so there are fewer shifts for these entries. Some instructions are changed directly, so the algorithm understands operation codes for the selected platform. However, it may not work for a completely different microcontroller.

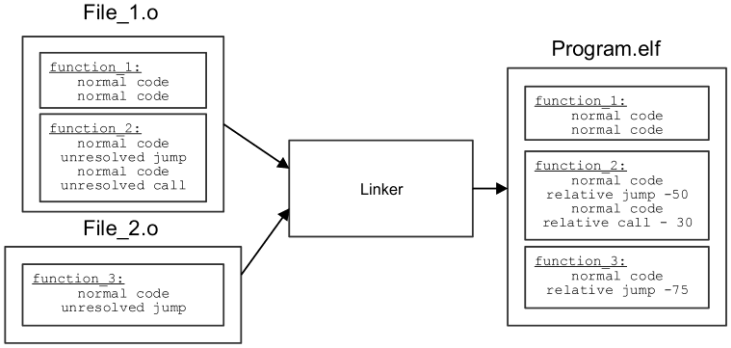

Fig. 1 Linking relocatable ELF files into the executable ELF file

Relaxation. During development of our own solution, we found some problems with the relocations that changed sizes after linking. This is called relaxation. For example, a compiler may allocate 4 bytes for a *call* instruction to some function. Linker discovers, that it placed the called function within 128 bytes of the relocation and thus reduces the relocation to a 2-byte *relative call*, shrinking the calling code. It is therefore important to check the resolved relocations in the executable ELF file.

3.3 Fragmented Firmware

Linkers can place code segments to any free address in the program memory. These segments are also called *.text* segments or sections. If there is a *.text* section generated for each function, linker can provide these functions with the slop region, a free space in which the function can grow or shrink without causing any other functions to shift [5]. This approach helps to generate small deltas and prolong the lifespan of the flash memories. Some works criticized this solution for an inefficient use of the program memory [4]. More jumps between pages in the memory cause more energy to be consumed, as activation of a different page consumes more power than activation of a different block within the same page [13].

4 Configurable Method for the Remote Reprogramming

This chapter presents our remote reprogramming methodology. It consists of methods that form a complete, platform independent solution capable of learning about the

changes in any firmware and generating as small delta files as possible. Our methods are applied in a software tool we developed. Our methodology includes methods responsible for the following tasks:

- Analyzing the object files, listing *.text* sections and relocations
- Tracking changes for each function, detecting new or deleted functions
- Assigning addresses to the functions based on their analysis
- Invoking linker, checking for relaxed entries in the final files
- Generating delta files

Our tool includes a wrapper for the GCC compiler and linker. We aim to test our method and tool on the three different platforms – 8-bit microcontroller from the AVR family, 16-bit microcontroller from the MSP430 family and a system-on-chip with the 32-bit ARM Cortex-M0 core. Each platform has its version of the GCC available and supports ELF files. We do not alter the instructions generated by the compiler. The flowchart of our method is in the Fig. 2.

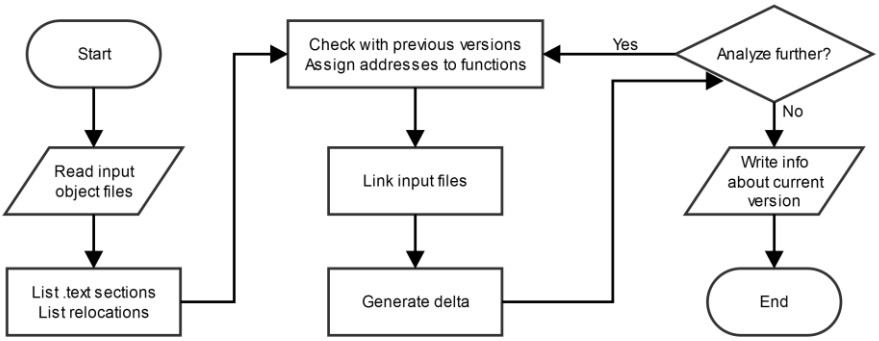

Fig. 2 Flowchart of the proposed method

4.1 Research Contribution and Innovation

None of the methods used in our methodology alter any ELF sections that hold instructions or relocations. There is no need to specify instruction formats or relocation types for any platform. This makes our solution platform independent.

Our methodology provides developers with various configurable options described in the following subchapter. Produced output files can be reverse analyzed and provide feedback on how the current configuration can be improved. This is a new approach. Various configurations may be the best for different scenarios.

Our methodology generates smaller deltas than basic differencing algorithms. Less data shared on the network improves the energy efficiency of our solution.

4.2 Description of the Processes and Configurability Options

This subchapter describes the configurable processes from the Fig. 2. These are the processes within our proposed method that contribute to its good performance.

List text sections, list relocations. Compilers enable developers to generate a *.text* section for each function, for example *.text.function1*, *.text.function2*. However, both will be linked into the *.text* section in the executable file. To prevent this, programmer must define a target section for each function in the source code. Our method includes two configurations:

1. Split functions into the sections defined explicitly by the developer
2. Edit string tables of the object files, place each function in its own section

The second option will simply iterate through the ELF string tables and add the number to each *.text* section, putting previously mentioned functions into sections *.text1* and *.text2*. This does not affect any firmware instructions.

Relocation entries can be read at this point, even though they are not resolved. List of these relocations may be later used before the effective delta generation by setting them to zero and making the firmware images more similar – inspired by [4].

Check with previous versions, assign addresses to the functions. In this step, our method collects information about every function that was present in the previous version. We evaluate its growth, previous positions, free space in its slop region (if present) and memory shifts its change may cause. The results of these evaluations can result in the different outline of the firmware's functions depending on the following possible configurations:

1. Static addresses – developer can assign some functions to a static address
2. No fragmentation – No function will be provided by a slop region
3. Partial fragmentation – Functions that would cause a lot of shifts are copied to a free space where they can be further edited
4. Full fragmentation – Every function is provided with a slop region.

Generate delta. After linking, when the relocations are resolved, there are two possible configurations:

1. Set all the relocations to zero, generate INSERT operations for the delta
2. Do not alter the relocations, run the differencing algorithm directly

Differencing algorithm also supports two different modes. These modes are relevant when the firmware is being fragmented or defragmented:

1. Dirty mode – unused sections that previously contained data are not erased
2. Clean mode – unused sections are erased and filled with 0xFF symbols.

5 Experimental Results and Discussion

This chapter provides experimental results that show, how various configurations of our methodology may be the most suitable for the different scenarios of over-the-air updates. In [10], we presented our differencing algorithm that outperforms existing solutions. Therefore, we use it for the following experiments executed on the ATmega32u4 microcontroller. We consider three scenarios:

1. In the first scenario, the devices on the network require one big update of the functions that sense, store and send data – the sensor module
2. The second scenario updates these three sensor module functions incrementally, one after another

3. The third scenario involves 9 incremental updates to various functions in the firmware code – the sensor module (3 functions), the communication module (3 functions), the flash storage module (3 functions)

We configure our tool to edit string tables of the ELF files, every function is placed in its own section. We do not set relocations to zero. For each scenario, we evaluate three different configurations:

1. No fragmentation; All deltas are generated in clean mode
2. Partial fragmentation; Changed functions are provided with a slop region (dirty mode), final delta generated in the clean mode defragments the memory
3. Full fragmentation; All functions are provided with slop regions before any update (clean mode), functions are then updated (dirty mode) and finally, the firmware is defragmented (clean mode)

Results. The results of our experiments are shown in the Table 1. Column "Delta files" shows, how many delta files were used for the update. Column "Total bytes" shows the sum of the delta file sizes used for reprogramming. Column "Frag. overhead" shows, how many of those bytes were used for memory fragmentation, defragmentation and cleanup. Table 2 shows the sizes of every delta file from the 3rd scenario. The sizes are in bytes.

Table 1 Experimental results for the three different scenarios and configurations

Config	Scenario 1 - single update			Scenario 2 -3 updates			Scenario 3 - 9 updates		
	Delta files	Total bytes	Frag. overhead	Delta files	Total bytes	Frag. overhead	Delta files	Total bytes	Frag. overhead
No frag.	1	98 (best)	0	3	216 (+7%)	0	9	1398 (+65%)	0
Partial frag.	2	164 (+64%)	112	4	202 (best)	102	10	844 (best)	260
Full frag.	3	680 (+593%)	634	5	638 (+215%)	576	11	1036 (+22%)	568

Table 2 Delta file sizes in bytes for the Scenario 3

Config	Δ_{frag}	$\Delta 1$	$\Delta 2$	$\Delta 3$	$\Delta 4$	$\Delta 5$	$\Delta 6$	$\Delta 7$	$\Delta 8$	$\Delta 9$	Δ_{defrag}	Total
No frag.	-	74	84	58	106	162	150	244	102	418	-	1398
Partial frag.	-	12	40	48	26	126	50	200	58	24	260	844
Full. frag.	302	14	14	34	26	126	20	200	20	14	266	1036

Discussion. The experiment shows, that the single firmware update does not require any memory fragmentation in order to be efficient. With more incremental updates required, configurations with memory fragmentation become more efficient – partial fragmentation performs the best for both 3 and 9 incremental updates. Full fragmentation configuration has the penalty of the big deltas that are used to fragment the memory at the beginning and to defragment it at the end. It is performing significantly better for more updates and it can be expected, that for a great number of incremental updates, this configuration will perform the best. Table 2. shows, that the delta sizes for the full fragmentation approach are mostly the smallest for the single function updates ($\Delta 1$- $\Delta 9$).

6 Conclusion

This paper presents a configurable reprogramming methodology that is built on the best practices in the area, and introduces configurations that make remote firmware updates more efficient for the different reprogramming scenarios. Experiments have shown, that various configurations may reduce total amount of the data shared on the network. The solution is multiplatform, which makes it ideal for the smart systems.

Acknowledgement. This work has been supported by Slovak national project VEGA 2/0192/15.

References

1. Arsan, T.: Smart Systems: From design to implementation of embedded Smart Systems. In: 13th HONET-ICT Int. Symp. on "Smart Microgrids for Sustainable Energy Sources enabled by Photonics and IoT Sensors", pp. 59—64. Haspolat (2016)
2. Moshnyaga, V.: Guidelines for Developers of Smart Systems. In: IEEE 8th International Conference on Intelligent Systems (IS), pp. 455—460. Sofia (2016)
3. Hui, J. W., Culler, D.: The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. In: Proc. of the 2nd Int. Conference on Information Processing in Sensor Networks (IPSN '08), pp. 81--94. ACM Press, New York (2004)
4. Dong, W., Mo, B., Huang, C., Liu, Y., Chen, C.: R3: Optimizing relocatable code for efficient reprogramming in networked embedded systems. In: IEEE INFOCOM Proceedings, pp. 315--319. Turin (2013)
5. Koshy, J., Pandey, R.: Remote Incremental Linking for Energy-Efficient Reprogramming for Sensor Networks. In: Proc. of the Second European Workshop on Wireless Sensor Networks, pp. 354--365. Istanbul (2005)
6. Kachman, O., Balaz, M.: Effective Over-the-Air Reprogramming for Low-Power Devices in Cyber-Physical Systems. In: IFIP Advances in Information and Communication Technology, vol. 470, pp. 284—292. Springer, Heidelberg (2016)
7. Huang, Y., Zhao, M., Xue, C. J.: WUCC: Joint WCET and Update Conscious Compilation for cyber-physical systems. In: 18th Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 65--70. Yokohama (2013)
8. Shafi, N. B., Ali, K., Hassanein, S.: No-reboot and Zero-Flash Over-the-air Programming for Wireless Sensor Networks. In: 9th Annual IEEE Communications Society Conf. on Sensor, Mesh and Ad Hoc Comm. and Networks (SECON), pp. 371--379. Seoul (2012
9. Hui, U. P., Jongsoo, J., Pyeong: Non-Invasive Rapid and Efficient Firmware Update for Wireless Sensor Networks. In: Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing, pp. 147—150. Seattle (2014)
10. Kachman, O., Balaz, M.: Optimized differencing algorithm for firmware updates of low-power devices. In: IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), Kosice (2016)
11. Unterschütz, S., Turau, V.: Fail-safe over-the-air programming and error recovery in wireless networks. In: Proc. Of the 10th International Workshop on Intelligent Solutions in Embedded Systems (WISES), pp. 27—32. Klagenfurt (2012)
12. TIS Committee: Executable and Linking Format (ELF) Specification Version 1.2. (1995)
13. Pallister, J., Eder, K., Hollis, S. J., Bennet, J.: A high-level model of embedded flash energy consumption. In: Int. Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES), ACM Press, New York (2014))