Marcin IWIŃSKI, **Janusz SOSNOWSKI**
INSTYTUT INFORMATYKI, POLITECHNIKA WARSZAWSKA,
ul. Nowowiejska 15/19, 00-665 Warszawa

Remote software reprogramming in embedded systems

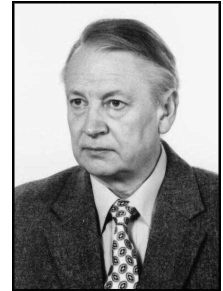
MSc. Marcin IWIŃSKI

He is a PhD student in Institute of Computer Science at the Warsaw University of Technology. He is coauthor of software for PW-SAT Satellite. He also works as embedded systems developer in Albatross System. Moreover, he participates in BRITE-PL satellite project in Space Research Center, Polish Academy of Sciences.

e-mail: m.iwinski@ii.pw.edu.pl

Prof. Janusz SOSNOWSKI

He is the professor in Institute of Computer Science at the Warsaw University of Technology. He chairs Department of Computer Software and Architecture. He is the author and coauthor of over 200 publications. His research area relates to computer dependability (diagnostics, fault tolerance, reliability), computer architecture and communication interfaces.



e-mail: jss@ii.pw.edu.pl

Abstract

In many applications basing on embedded systems we have the problem with limited access for servicing. During the exploitation of such systems it happens that various errors can appear in hardware or software. Many of these errors can be eliminated (e.g. single event upsets), avoided or repaired (e.g. software bugs) by reprogramming the system partially or completely, locally or remotely. The paper discusses strategies of this approach taking into account various limitations and presents a case study solution designed for a satellite microcontroller.

Keywords: embedded systems, reprogrammable software.

Zdalne rekonfigurowanie oprogramowania w systemach wbudowanych

Streszczenie

W systemach wbudowanych dość często pojawia się problem ograniczonej ich dostępności dla serwisowania. Jest to typowe dla systemów przemysłowych, telekomunikacyjnych i kosmicznych. Podczas eksploatacji takich systemów mogą się pojawiać różne błędy w sprzęcie i oprogramowaniu. Wiele spośród tych błędów może być wyeliminowanych, maskowanych lub naprawianych poprzez reprogramowanie częściowe bądź pełne, lokalne albo zdalne. W pracy przedstawiono strategie reprogramowania uwzgledniające szereg spotykanych ograniczeń (np. wymagania przetwarzania w czasie rzeczywistym, ciągłość pracy, ograniczone zasoby pamięciowe i komunikacyjne). Problem ten odniesiono do proponowanych rozwiązań w literaturze oraz wykorzystano tu zdobyte doświadczenia z projektowanymi systemami wbudowanymi pomiarowymi i kontrolera satelitarnego. Ponadto przedstawiono bardziej szczegółowo studium mikrokontrolera dla satelity. Bazuje ono na doświadczeniu zdobytym przy satelicie PW-SAT. W pracy została zaproponowana koncepcja podziału programu na segmenty celem zmniejszenia rozmiaru pliku aktualizacyjnego. Dla omówionego przykładu mikrokontrolera satelitarnego oraz wybranych scenariuszy aktualizacji zostały przedstawione wynikowe rozmiary plików aktualizacyjnych przed i po zastosowaniu proponowanej segmentacji.

Słowa kluczowe: systemy wbudowane, reprogramowanie system.

1. Introduction

In the literature reconfiguration problems are discussed mostly in relevance to FPGA based subsystems [5]. They are also encountered in SoC systems [5]. This issue is becoming quite important in the case of various measurement and control electronic devices with sophisticated microcontrollers and complex programs. According to various dependability requirements and limited access for the service special care is needed in the design phase. In particular, we have to assure the capability of system reprogramming taking into account various limitations and requirements. We have experienced a practical need of this approach in the previously developed satellite controller and gas counter. The lack of this capability was a great disadvantage due to some errors which appeared during exploitation. Unfortunately, in the first case this led to losing some important functionality. In the second case this resulted in longer service operations and long period of device unavailability (caused by classical service turnaround procedure). In the literature mostly coarse-grained reconfiguration is considered targeted at functional changes [2,4,6-8]. Fine- grained reconfiguration handling faults is neglected.

Section 2 describes various strategies of reconfiguration. Section 3 presents a case study of remote software reconfiguration in a developed satellite microcontroller. Final conclusions are given in Section 4.

2. Reconfiguration schemes for embedded systems

The main goals of system reconfiguration relate to: i) extending or changing its functionality, ii) improving performance, iii) temporal upgrading, iv) eliminating software bugs, v) tolerating hardware faults. Depending upon the system resources reconfiguration can be performed in hardware or software. In the latter case we can base on reconfigurable hardware platforms such as FPGA circuits or programmable CPU chips (e.g. with programmed list of instructions) or fixed platforms. Reconfigurable platforms give more flexibility (e.g. FPGAs). In fixed platforms we have limited capabilities of switching on or off some functional blocks (e.g. cache memory, I/O drivers) and using other available blocks. When designing some embedded systems, it is reasonable to make them more universal and configurable to specific tasks or environment requirements e.g. gas counters adapted to specific gas parameters.

Reconfiguration is an imminent feature of FPGA based systems. The configuration RAM memory is sensitive to internal or external disturbances. This may relate to single event upsets (SEUs) caused by cosmic radiation or electromagnetic disturbances. They result in bit flip faults which can be identified with periodical read back and checking the configuration data. Sometimes configuration memory sub blocks (frames) are protected with error detection codes so only such frames can be reprogrammed (partial reprogramming). Such reconfiguration takes much less time than complete reconfiguration which results in system operational discontinuity. Nevertheless, this approach needs some hardware resources (within FPGA) to handle this functionality as well as a communication channel for transmitting the requested configuration frames (e.g. from flash memory).

In the case of CPU based systems the reconfiguration mainly relates to programs. However, in complex systems reconfiguration at micro architectural level is also possible e.g. changing microcode, firmware, instruction set. Reconfiguration at this low level is rather not encountered in embedded systems. Here, we can base mostly on reconfiguration of the program and some selected architectural levels e.g. switching cache memory, replacing faulty lines in cache (assuming some spare lines are available). Typically, programs are stored in flash memory which is resistant to bit flip faults. However, the executed code can be fetched from cache memory susceptible to bit-flip faults. Such faults are critical and can be eliminated with on-

line or periodical detection followed by invalidation of cache lines or reloading the program code from flash memories. Hence, in some systems with lower time requirements programs are executed directly from the flash memory.

An important issue is the range of reconfiguration: total or partial. The first one is usually time consuming and blocks system operation for a longer time. In practice, we face quite often the need of partial reconfiguration involving limited system resources. Moreover, it may be required to assure limited interference with the system operation. In general reconfiguration can be performed automatically, semi automatically or manually. The first approach is typical for fault tolerant systems with error recovery capabilities and automatic software upgrades. Here, it is worth mentioning reconfigurations related to system environment changes e.g. replacing or removing a hardware module, adding a new one (fulfilling hot swap and plug&play capabilities).

In general, there is some risk of performing unacceptable reconfigurations e.g. by unauthorized activities as well as faulty reconfigurations due to some errors during this process. So special safety measures, e.g. checksums, encryption and authentication procedures, have to be included.

Quite often limited memory resources do not allow us to store alternative code, so reprogramming is needed. Replacing a segment of the program code in fact needs also some available resources and time reserves. This can be done remotely via some communication channels, sometimes with limited throughput and limited availability. This is especially critical in the case of cosmic equipment e.g. satellites. Moreover, here we have to take into account transmission blackouts e.g. due to the position of orbiting satellites. Hence, the reconfiguration has to be rolled over time and performed partially.

Partial reprogramming is effective in the case of small changes in the object code involving small areas of the program address space. Here, it is worth noting that when performing even small changes in the source code we can obtain big changes in the object code image. It seems interesting to use incremental compilers, which recompile only the changed lines of the source code. Unfortunately, these compilers are designed to improve turnaround time of software development. They assure recompilation time proportional to the changes in the design [1] and do not optimize changes in the object code. Another possibility is to design easy modifiable software by appropriate partitioning of the code. In [3] this problem is partially resolved by a special software environment including the operating system. In fact, it is too complex for many embedded systems. Section 3 illustrates these problems in relevance to a simple microcontroller.

3. Reprogramming satellite microcontroller

Having experienced some software bugs in the launched student satellite (PW-SAT, Cube Sat mission 2012/2013, see http://www.pw-sat.pl/) we have decided to develop the capability of remote reprogramming. For this purpose we use AX.25 based communication link which is a radio amateur protocol commonly used for Cube Sat satellites. Data throughput is relatively slow - up to 1200bps. This link is mainly dedicated for transmitting telecommands from the earth station and receiving telemetry data.

Most of Cube Sat missions are launched to LEO orbit like PW-SAT and other Cube Sats launched with the same Vega rocket. Typical visibility of a PW-SAT satellite is a few minutes every 1-2 hours depending on the orbit position, hence the communication link is a bottleneck. This imposes some restrictions on reprogramming, they are also combined with memory and CPU usage limitations within the on board microcontroller. Taking this into account we have extended basic functions of the microcontroller boot loader with managing the reprogramming processes. In particular, we assure partial reprogramming rolled over the time, so as to limit its interference with the normal operation.

The reprogramming is composed of 4 steps: uploading binary data for reconfiguration to temporal data memory, suspension of the normal operation (with stored system status), erasing and reprogramming the appropriate memory area, resuming the normal (updated) operation. AX.25 communication link provides only plain data transfer. Higher level data transfer protocols with packet acknowledgement and encryption must be implemented to provide a reliable and secure data link for transferring software update files. Storage memory for temporal firmware update buffering can be a dedicated part of data FLASH memory or even unused part of the microcontroller FLASH program memory. Our project bases on ATmega1280 microcontroller with 128kB of program FLASH memory. Software used in our experiment needed up to 17kB of program memory at worst. This provided us a possibility to use the surplus memory for temporal update data storage. After the completed transfer the microcontroller performs reset and enters boot loader mode. The program reset for the software update is not performed immediately but during not critical operations of the running control program. Suspension or termination of some processes with optional status saving can occur before the update process. The first step after resetting and entering boot loader mode is checking integrity (e.g. CRC) of the uploaded data. Checksum mismatch causes abortion of update process and resuming recent software. Another step is reprogramming the requested part of the program memory. Minimal reprogramming areas are 256 byte pages which must be erased and then programmed with the new data. Both operations take up to 4,5ms resulting in 9ms of total reprogramming time of every page. There is also some time overhead related to microcontroller resetting and boot loader activity (a few milliseconds). Comparing to FLASH programming time these values are negligible. Hence, complete program update (about 20kB) requires at least 720ms. After the whole reprogramming process microcontroller resets once again and starts executing the updated program.

In practice, we face the problem of partial reprogramming (e.g. to change some part of code, eliminate bugs) and the limited operational interference. The reprogramming command is followed by the new replacing code. This code is presented as standard Intel Hex file. It comprises consecutive lines of binary code data with attributed addresses in FLASH memory where new firmware should be loaded. The Hex file size is typically up to 3 times larger than the needed area in program FLASH memory. This results from the fact that the binary data is coded in hexadecimal ASCII characters with additional address, checksum, and line delimiters (start and stop characters). Typically, a single line comprises 44 characters.

When preparing code modifications, we should assure minimal changes in the object (binary) code to reduce transmission and operation interference. We should be conscious that even a small modification of the source code may result in significant changes of the object code (after compilation), specified by many transmitted lines of the reconfiguration Hex file. This is caused by the fact that some changes usually imply moving the unchanged program code to new addresses, keeping the whole code section consistent. To alleviate this problems, we can divide the program code into a few independent code sections (segments) by appropriate specifications during the code compilation. This partitioning can be consistent with the code functionality or source code files. Here, some knowledge of section sizes is needed to allocate them appropriate start addresses so as to assure some extra space between subsequent sections. The program size does not grow but additional FLASH memory is reserved at the end of each section to provide some space if the upgraded code exceeds the size of the old one. If the updates fit into the replaced section area, then the remaining sections do not need changes. Exceeding the section boundary with a program update needs reallocation of at least one of the following sections and sending a new data in the update file. An amount of the spare program memory for program sections should be allocated for every section individually taking into account plans for adding future functionality. If there is no need of further program development but just bug fixing, then only a few percent of additional space is sufficient to prevent section boundary crossing with most of the future updates.

Having developed the presented idea, we performed several experiments with various compiler options, in order to get better

knowledge of the reconfiguration load. The analyzed satellite subsystem program consisted of 8 different „*.c" source files resulting in 8 different sections of code. The developed control program (for ATmega1280 microcontroller) was written in Atmel Studio 6 environment and compiled using embedded C language compiler based on popular gcc compiler. The object code size depends on the compiler settings. We used 5 different compiler optimization modes which generated various hex file sizes. Option "-O0" relates to compilation without any optimization. Option "-O3" provides maximum optimization of the execution time and typically generates the object (binary) code file size larger than those received using other optimization modes. This results from the massive usage of inline functions and loops imposed by the compiler operating in "-O3" mode. The most interesting is "-Os" option which assures the minimal code size, however the execution time is longer than in mode "-O3". Options "-O1" and "-O2" offer various basic optimization and result in output code sizes between "-Os" and "-O3".

Tab. 1. Statistics of program file sizes without code segmentation
Tab. 1. Statystyka plików programu bez segmentacji kodu

Comp.	Hex	Hex'	Lines	Lines'	Lines*	Lines%
O0	47 571	48 360	1059	1076	1066	99,1
O1	26 302	26 969	586	601	591	98,3
O2	25 848	26 462	576	590	580	98,3
O3	35 355	36 002	787	802	792	98,8
Os	24 232	24 261	540	552	542	98,2

Tab. 2. Statistics of program file sizes - code segmentation (2 segments modified)
Tab. 2. Statystyka plików programu z segmentacją (modyfikacja 2 segmentów)

Comp.	Hex	Hex'	Lines	Lines'	Lines*	Lines%
O0	47 601	48 441	1061	1080	562	52,0
O1	26 345	27 012	589	604	390	64,6
O2	25 917	26 531	581	595	353	59.3
O3	35 424	36 058	792	806	415	51,5
Os	24 301	24 836	545	557	291	52,3

Tab. 3. Statistics of program file sizes with code segmentation (1 segment modified)
Tab. 3. Statystyka plików programu z segmentacją (modyfikacja 1 segmentu)

Comp.	Hex	Hex'	Lines	Lines'	Lines*	Lines%
O0	47 601	47 601	1061	1061	3	0,3
O1	26 345	26 345	589	589	2	0,3
O2	25 917	25 954	581	582	158	27,1
O3	35 424	35 461	792	793	180	22,7
Os	24 301	24 317	545	545	147	27,0

The developed control program was submitted to some modifications in the source code. Moreover, we considered two approaches without code segmentation and with code segmentation (8 segments). For each experiment (Tabs. 1-3) we specified the size (in bytes) of the original code in Hex file (Hex), the size of the modified code (Hex') and the corresponding to them numbers of lines in the hex file (Lines, and Lines', respectively). We also specified the number of differing lines (Lines*) and related percentage of code changes (Lines%). In the case of Tabs. 1 and 2 we used the same modified source code, which was generated from the original code version by adding a new function (20 additional lines of source code) and by modifying another existing function (4 lines of source code). These changes constitute a small percentage of the whole source code (several thousands of lines), however after the compilation they resulted in almost 99% of binary code changes in the case of classical non segmented source code (Tab.1). The results show that quite simple modifications of the source code caused almost every line of the hex file to change. Hence, practically full update is required to change the software.

Another version of the source code was developed by distinguishing 8 code segments. In two relatively large segments we performed the same source code changes as in the previous experiment. The results are given in Tab.2. In this case up to 52% reduction in update file size can be achieved depending on the compiler option (the lowest value for non-optimization). We expected lower number of lines needed for the program update. The relatively large value results from the fact that code modifications were introduced into 2 largest program functions resulting in a large portion of the program code to be relocated. Another fact is that the code sections are not sufficiently isolated. For example a modification of one section causes changes in function addresses, and in consequence changing function call instructions in other sections may be required. Tab. 3 shows results related to a code modification in a smaller program segment. The object code changes are marginal for lower optimization level of the compiler. However, for "- O1" option the code size is close to the optimal one and the upgrade cost is negligible. The same code modification performed without segmentation provided similar results for "O0" and "O1" options, for the remaining options the object code changed in 83-88%. The code segmentation increased the program size by 0.15-0.3% depending upon the optimization option.

When using standard software environment and compilers, some care is needed in designing reconfigurable programs satisfying timing and resource restrictions.

4. Conclusions

The capability of reconfiguring embedded system becomes popular requirement in devices mounted in the field with expensive or limited access. The main goals of reconfigurations relate to modification of implemented functions, their extensions as well as changes forced by hardware or software faults. Having presented strategies of reconfigurations, we have concentrated on CPU based microcontrollers and software reconfiguration issues. The general considerations resulted in development of reconfiguration capabilities in a satellite microcontroller taking into account the specificity of its environment and possible restrictions in transmission, available memory resources and normal operation interference. The gained experience can be used also in other projects, e.g. in complex measuring systems in industrial environment or distributed over a large area. More advanced solutions of partial software reprogramming will be developed.

5. References

[1] Cook Ph., Welsh J., Hayes I.J.: Building a flexible incremental compiler back-end, Technical Report SSE-2005-02, The University of Queensland, http://www.itee.uq.edu.au/~sse

[2] Deng Q., Wei S., Xu H., Han Y., Yu G.: 4 A Reconfigurable RTOS with HW/SW co-scheduling for SOPC, Proc. of the 2nd Inter. Conference on Embedded Software and Systems (IEEE ICESS'05), 2005.

[3] Gracioli G., Froehlich A.A.: ELUS: a dynamic software reconfiguration infrastructure for embedded systems, IEEE Int. Conf. on Telecommunication, 2010.

[4] Hu Y., Hang C.: A dynamic reconfigurable adaptive software architecture for federate in HLA-based simulation, The 8th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007.

[5] Legat U., Biasizzo A., Novak F.: On-line self-recovery of embedded multi-processor SoC on FPGA using dynamic partial reconfiguration, Information Technology and Control, ISSN 1392 – 124X, Vol. 41, No.2, 2012.

[6] Piłat A., Grega W.: Hardware and software architectures for reconfigurable time critical control tasks, Proc. of the International Multiconference on Computer Science and Information Technology, ISSN 1896-7094, 2006.

[7] Rummler R., Aghvami K.H., Boom S., Arram B.: Traffic modeling of software download for reconfigurable terminals, IEEE Int. Symp. PIMRC, 2001.

[8] Saha P.: Automatic software hardware co-design for reconfigurable computing systems, 17th International Conference on Field Programmable Logic and Applications (FPL 2007), 2007.

otrzymano / received: 20.05.2013
przyjęto do druku / accepted: 03.07.2013 *artykuł recenzowany / revised paper*