DEVELOPMENT OF VARIANT OF SOFTWARE ARCHITECTURE IMPLEMENTATION FOR LOW-POWER GENERAL PURPOSE MICROCONTROLLERS BY FINITE STATE MACHINES

Pavlo Katin
Department of automation and Control in Technical Systems
National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute"
37 Peremogy ave., Kyiv, Ukraine, 03056
p.katin@kpi.ua

Abstract

As a result of the research, two directions for development of software architecture for low-power general purpose microcontrollers (LPGPM) are identified. The first, classical approach is the development using standard State patterns. The second is the development of programs, algorithms and structures based on mathematical analysis.

The first direction is chosen in the work. The variant of the implementation of a typical pattern for development of software architecture (SA) in the form of a finite state machine (FSM) is proposed to discussion. This pattern allows to divide the development of the architectural part of the program for LPGPM and programming the LPGPM hardware. This approach makes it possible to divide the work of the software architect and the work of LPGPM hardware specialists. Advantage of the solution in comparison with the real time operating system (RTOS) is the saving of LPGPM hardware resources. In addition, it improves the readability of code and good testing prospects. The resulting architecture makes it possible to easily accompany the software and switch to other types of microcontroller. The disadvantage is an increase in the required amount of RAM with an increase in the number of states. It is this disadvantage that requires the application not only of experimental and engineering-intuitive methods, but also to continue research in the second direction.

Keywords: finite state machine, low-power general purpose microcontrollers, software architecture, real time operating systems.

DOI: 10.21303/2461-4262.2017.00361

1. Introduction

This study proposes the version of the software architecture for LPGPM without the use of RTOS. In addition, the results of early experimental studies of architectural solutions in the form of the State pattern are presented. This pattern is a software implementation of the mathematical model of FSM. The peculiarity of the research is that the architecture of programs for embedded systems based on LPGPM is developed, which allows to divide the development process of the SA architecture and programming of hardware.

The first results of the theoretical foundations of the FSM were outlined in [1-5]. In these works, the theoretical part of the FSM issues and the hardware implementation of theoretical provisions are disclosed. In [6], nondeterministic and deterministic FSM are separated. In the future, the FSM theory has developed in the development of software architecture. In the first works of this subject FSM theory [7] is used to improve the algorithmization and simplification of debugging programs. In modern works, the study of software formalization in the form of FSM has been continued [8].

An important result of SA generalization in the context of object-oriented programming is [9]. It has ordered many commercial software architectures based on object-oriented programming technology. These results are summarized in the form of patterns. In turn, the selected patterns are systematized and organized into specific groups. The patterns described in [9] are still used in the development and continue to be improved. Among the many patterns for developing embedded systems, the State pattern deserves special attention. This pattern can be directly described in the form of FSM theory and is a software implementation of FSM mathematical model. The generalized description and architecture of the State pattern are given in [9].

The most interesting recent works, which are directly related to the research subject of this article, are [10, 11]. In these works, the FSM model for SA development for LPGPM is used. The

importance of the State pattern is confirmed by the presence of formal descriptions and standardization based on UML 2.0. This diagram is called the Finite State machine diagram [12]. In [13], the author tried to justify the shortcomings of this pattern, which has certain grounds for it and offered its solution. In [14, 15], the implementation of the State pattern is proposed, which makes it possible to create hierarchies of state classes.

Summarizing the above, it can be argued that the State pattern is widely used in ES programming, including for LPGPM. To further outline the conducted and prospective studies in this article, two areas are identified that are associated with the FSM model for SA development for LPGPM.

2. Materials, software and research methods

2. 1. Research Areas

One of the directions that are highlighted above is development of the software architecture for LPGPM using standard State patterns [16, 17]. An attempt to systematize various solutions of this pattern is described in [16]. To new works of this direction can be attributed [10, 11]. Obviously, the language of development of such systems is C, C ++, Assembler. The simple implementations of FSM without the use of OOP are well studied [11].

The second direction is development of programs, algorithms and structures based on mathematical analysis. Programming practice shows that such solutions are not a frequent occurrence. At the same time, papers are known in which the results of this approach are presented [18, 19]. Let's consider an example [17]. Let's skip the mathematical part and analyze the solution, which is presented in **Fig. 1** in [17]. The example can be implemented programmatically, including as an OOP program using the State pattern. Probably, the direction of further research [17] is the definitions of recommendations and possible options for SA architecture implementation, obtained on the basis of mathematical analysis. Thus, it is possible to formulate the task of the second direction: development of methods for transition of a mathematical model into a typical SA architecture and definition of typical patterns.

2. 2. Choice of hardware

At the beginning of the work, the hardware platform LPGPM is chosen, for which SA is developed and debugged on the basis of the State pattern. Given the chosen hardware architecture of LPGPM, a real program is developed that implements the FSM model in the State pattern architecture. It is this debugged real program is used for further research. It implements SA based on the State pattern. Thus, the first direction is chosen.

In this research, the architecture of a typical microcontroller MSP430g2453 Texas Instruments, USA is chosen. To repeat the research and verify the obtained results, it is advisable to select another hardware platform LPGPM or other manufacturer, for example, the microcontroller STM8S903F3 of the company STMicroelectronics, Switzerland; PIC10F220 of the company Microchip Technology Inc., US; Atyny4 of the company Atmel® (Microchip Technology Inc.), US. Recommendations for selection are presented below.

Significant differences in the modification are assumed in the software implementation of hardware interrupts. Simple example of States is the control of the general purpose input/output (GPIO) ports of the microcontroller. It is advisable to check the results in parallel on a software emulator, for example Proteus and on a real debug board. When choosing the architecture of a microcontroller, it must be taken into account that it must be of general purpose, of an entry level, no more than 16 bits. The use of complex architectures, for example ARM, is likely lead to distortion of results or a new direction of work. No use RTOS. It leads to distortion of results.

2. 3. Investigation of SA influence on LPGPM resources

Compilation and debugging of the source code is carried out using the IAR system IAR Embeded Workbench for MSP430 Version 6.50.4. (Free version). Then this medium is used for research. To conduct such study, it can be used other software that is suitable for the selected LPGPM type.

The experimental part of research consists in obtaining, debugging and testing the program for LPGPM with the same functionality for different architectural solutions. In this case, this architecture

is implemented in two versions. The first option is based on the SA software architecture, in a typical SWITCH solution for the State pattern [8]. The second option is based on the proposed architecture.

In the course of the experiment, the program for the two versions above-described options is developed and debugged. Further in the experiment, perform comparison of the two versions programs memory size Read Only Memory (ROM) and the amount of Random Access Memory (RAM).

3. Research results

Let's consider the our variant resulting sample pattern for SA development on the basis of FSM. (This pattern allows to divide the development of the architectural part of the program for LPGPM and the control process of LPGPM hardware. This approach makes it possible to build a good SA and share the work of the software architect and the work of LPGPM hardware specialists. It is necessary to confirm the expediency of using the known typical State patterns and the obtained pattern variant for embedded devices based on LPGPM, since similar results are not presented in [9–11, 19]. The importance of this is that the limitations of the program memory (Read Only Memory (ROM) and RAM are critical for LPGPM.

Let's introduce certain restrictions on conducted research and the possibility of comparing the results:

– the study considers SA implementation exclusively in programming languages C, C ++;

– architectural features of SA on the basis of Assembler or machine code must be put into a separate study, the obtained results should be analyzed separately;

– the work explores a different SA architecture that provides the same functionality, and checks its effect on the required ROM and RAM limitations;

– 3–4 states are implemented during the experiments in the software.

At the first stage, SA is implemented as a standard in the form of an infinite cycle [11]. This decision is subject to criticism from adherents of classical architecture in the form of a finite state machine. This approach is often used in practice for development under LPGPM. The software implementation of this architecture based on switch FSM is described in [11, 19].

The implementation of the switch FSM architecture, which is described in [11, 19], is in our investigated, the source code is attached (file FSM_SWITCH). It implements three states, with the nested FSM on two states. The code is written for the MSP430G microcontroller, compiled for IAR Version 6.50.4. As a result of the compilation, the amount of the programs code is 1.8 KB for loading to the ROM of the microcontroller. File type is *.hex. RAM overflow is not detected during running the program.

For the conditions described above, the classic implementation of FSM as a result of compilation has 2.55 Kbytes of the programs code is for (running) loading the program into the ROM of the microcontroller. File type is *.hex. RAM overflow is not detected during running the program.

The peculiarity of the new architecture is that, without using RTOS, it is easy to add new states to the architecture. At the same time, the work of the architect and LPGPM hardware specialist is easily shared. The SA states are maximally isolated from each other. In this case, it is possible to change the order of the transition from state to state.

The demo version of the architectural solution is represented in the form of C # code. The real results are obtained in the IAR Embeded Workbench for MSP430 Version 6.50.4 development environment.

Let's proceed to the description of the obtained variant of the pattern implementation in the form of the source code. **Fig. 1** shows the implementation of the interface of the standard State pattern. This interface shows methods that will be redefined later.

Fig. 2 shows an example of the implementation of the interface is shown (**Fig. 1**). In this case, there are no significant differences from the standard State pattern. As is known, the State pattern assumes a common interface for each of the FSM states. Other states (4 in this case) are not given, because they differ only in the content of the conditional functionality of the methods in the demo version. In the real program for LPGPM, the functionality of the methods is determined by the need to control the microcontroller hardware and will be developed by the hardware specialist that isn't required to delve into in other parts of the program.

```
using System;
// For demo only
namespace Semaphore
{
    public interface ISlpgm //States of LPGM (low power of microcontrollers general purpose)
    {
        void FunctionaLPGMHardware1();
        void FunctionaLPGMHardware2();
        // And many ...
    }
}
```

Fig. 1. The pattern interface

```
using System;
// For demo only

namespace Semaphore
{
    public class StateA : ISlpgm
    {// In this methods we can easy chene functional of LPGM Hardware
        public void FunctionaLPGMHardware1()
        {
            Console.WriteLine("StateA");// For demo only !!!
        }
        public void FunctionaLPGMHardware2()
        {
            //Any LPGM Hardware  programs control
        }
// And possible meny functions of LPGM Hardware
    }
}
```

Fig. 2. Demonstration implementation of the interface, one of four states: the first state, StateA

A new class is a new element that simplifies the overall control of the FSM transition sequence from state to state. It is shown in **Fig. 3.** In the source code below, the formatting of the code is somewhat distorted, which does not affect its performance.

The basis of the proposed solution is the dictionary (line 10). Elements of the dictionary are pairs: an integer that regulates the order, and an implementation of the interface that form the order of work. In this line, we can choose the order of transition from one state to another. The sequence is regulated by the numbers on the left side of the dictionary. The example in **Fig. 3** clearly shows this (line 11–14). It is enough to change the order of numbers or change the state. **Fig. 3** demonstrates the simplicity of changing the order of the sequence. The advantage of this approach is the possibility of easy order control.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Threading.Tasks;
4
5  namespace Semaphore
6  {
7      public class StateManager
8      {   int currentState = 0;
9          private int step = 1;
10         private Dictionary<int, ISlpgm> States = new Dictionary<int, ISlpgm>()
11         {   {0,new StateA()},// In this please we can change state easy !
12             {1,new StateB()},
13             {2,new StateC()},
14             {3, new StateD()} };
15          public ISlpgm GetState()
16         {
17             ISlpgm result;
18             States.TryGetValue(currentState, out result);
19             currentState += step;
20             if (currentState>=States.Count-1 || currentState<=0)
21             {step = -step; }
22          return result;
23         }}}
24
```

Fig. 3. Control system of transition order from state to state

The disadvantage of this solution is an increase in the RAM amount that is required for work. This will happen because of the need for resources due to the creation of new instances

of classes (line 11–14). To solve this issue, this code can be rewritten. For example, the references can be passed to instances of classes to the list. The evidence of this decision does not require confirmation by the source code. In the case when the number of states is large, the above solution can be limited to the operational memory of LPGPM and it is necessary to proceed to the solution.

4. Discussion of research results

The task of the experiment is to confirm a slight increase in the required program memory, using the obtained SA. In this case, it is required to show a smaller volume in comparison with the variants of using RTOS. These results are especially relevant when using links in lines 11–14 (**Fig. 3**), instead of generating an instance of the class.

Based on the research results, it can be argued that the advantage of such architectural solution is the following: readability of the code, the possibility of easy code control, good testing perspectives. The essence of the difference from the classical solution lies in the fact that the management of the order of the transition from one state to another is rendered in a separate class method. It can be implemented as a separate function. This preserves the advantage of the standard State pattern, which is the ability to separate states in one class.

The disadvantage of the approach is an increase in the code size with an increase in the number of states.

5. Conclusions

As a research results, two areas of research have been identified for development of software architecture for LPGPM. The first is development using standard State pattern. The second is development of programs, algorithms and structures based on mathematical analysis.

In the second direction, the results are obtained, which are proposed for discussion, in the form of a standard pattern for SA development in the form of FSM. The received solution differs from the typical State pattern. The difference lies in the possibility of separating the work of SA developer and the specialist in LPGPM hardware architecture.

Numerical experiments are carried out. They show a small amount of increase in program memory (Read Only Memory (ROM)) of the memory of LPGPM microcontroller when using the patterns.

Using of OOP and sample patterns does increase the used RAM and ROM in some cases up to 3 times. Therefore, it is necessary to study the obtained patterns and a more detailed study the issue under consideration.

In addition, the direction of combining the solutions obtained or searching for new ones is determined, which allow obtaining a relatively small amount of required memory and easy maintenance of the code.

Also, the open direction is the possibility of applying mathematical methods for solving problems of improving the architecture of software solutions, not only for development of devices based on LPGPM, but also for improving the debugging of network applications, analytically determining the size of the file for downloading, and analytical evaluation of the result.

References

[1] McCulloch, W. S., Pitts, W. (1990). A logical calculus of the ideas immanent in nervous activity. Bulletin of Mathematical Biology, 52 (1-2), 99–115. doi: 10.1007/bf02459570

[2] Huffman D.A. (1954). The Synthesis of Sequential Switching Circuits. Journal of the franklin Institute, 257, 3, 161–190.

[3] Mealy, G. H. (1955). A method for synthesizing sequential circuits. The Bell System Technical Journal, 34 (5), 1045–1079. doi: 10.1002/j.1538-7305.1955.tb03788.x

[4] Albrecht, A. J. (1979). Measuring Application Development Productivity. P Proceedings of the joint SHARE/GUIDE/IBM application development symposium, 10, 83–92.

[5] Shannon, C. E., McCarthy, J. (Eds.). (1956). Automata Studies. (AM-34), Princeton University Press, 285. doi: 10.1515/9781400882618

[6] Rubin M., Scott D. (1959). Finite automata and their decision problem. IBM journal of research and development, 3 (2), 114–125.

[7] Thompson K. (1968). Programming techniques: Regular expression search algorithm. Communications of the ACM, 11 (6), 419–422. doi: 10.1145/363347.363387

[8] Shalyto A. A. (1998). SWITCH-tehnologija. Algoritmizacija i programmirovanie zadach logicheskogo upravlenija. Saint Petersburg: Nauka, 628.

[9] Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995). Design Patterns Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 416.

[10] Spinke, V. (2013). An object-oriented implementation of concurrent and hierarchical state machines. Information and Software Technology, 55 (10), 1726–1740. doi: 10.1016/j.infsof.2013.03.005

[11] Nyman, S. (2012). IAR Application Note # 6811-003. Implementing a State Machine. Available at: http://supp.iar.com/FilesPublic/SUPPORT/000370/AppNote_6811-003_State_Machine.pdf

[12] Thomas, D. (2003). UML – Unified or Universal Modelling Language. Journal of Object Technology, 2 (1), 7–12.

[13] Shamgunov, N. N., Korneev, G. A., Shalyto, A. A. (2004). State Machine — novyj pattern obektno-orientirovannogo proektirovanija. «Informacionno-upravljajushhie sistemy», 5. 13–25.

[14] Sane, A., Campbell, R. (1995). Object-oriented state machines. Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications – OOPSLA'95. doi: 10.1145/217838.217841

[15] Shalyto, A. A., Tukkel', N. I. (2001). SWITCH-tehnologija - avtomatnyj podhod k sozdaniju programmnogo obespechenija « reaktivnyh» sistem. Programmirovanie. 8, 45–62.

[16] Adamczyk P. (2003). The Anthology of the Finite State Machine Design Patterns. The 10th Conference on Pattern Languages of Programs. Available at: http://hillside.net/plop/plop2003/Papers/Adamczyk-State-Machine.pdf

[17] Solodovnikov, A. (2016). Developing method for assessing functional complexity of software information system. EUREKA: Physics and Engineering, 5, 3–9. doi: 10.21303/2461-4262.2016.00157

[18] Chainikov, S., Solodovnikov, A. (2016). Information technology of software architecture structural synthesis of information system. EUREKA: Physics and Engineering, 4, 25–32. doi: 10.21303/2461-4262.2016.000125

[19] Frimen, Je., Frimen, Je., S'erra, K., Bejts, B. (2012). Patterny proektirovanija. Saint Petersburg: Piter, 656.