Відомі технічні системи реального часу (real-time technical systems RTTS), вимагають високої швидкості програмних рішень. Крім того, для них потрібно забезпечити уніфікацію вихідного коду, якість супроводу програмного забезпечення та математичне моделювання. Все це потрібно реалізувати з відносно невисокою вартістю програмного і апаратного рішення. Апаратну частину можна реалізувати на основі розповсюджених мікроконтролерів архітектури Cortex-M.

Програмна частина даних мікроконтролерів може бути реалізована на основі операційної системи реального часу (ОСРЧ) (real-time operation systems RTOS). В ході досліджень було виявлено два недоліки. Першим є те, що використання ОСРЧ призводить до обмеження швидкості. Другим недоліком є труднощі уніфікації, підтримки вихідного коду (Source)і математичного моделювання.

Для усунення недоліків розроблені типові програмні патерни Стан для допоміжного контролера у колі виконавчих механізмів або датчиків на основі мікроконтролерів архітектури Cortex-M в режимі реального часу, в процедурній парадигмі. Особливістю таких патернів є висока швидкість програмного рішення (software)у порівнянні з рішеннями на основі ОСРЧ.

Розроблені патерни дозволяють уніфікувати вихідний код (Source) для мікроконтролерів архітектури Cortex-M різних виробників, покращити супроводження програмного забезпечення (software)і адаптувати його до математичної моделі кінцевого автомата (mathematical model finite state machine).

Результати пройшли випробування на мікроконтролері STM32F103 з використанням бібліотеки Cortex microcontroller software interface system (CMSIS). Це дозволяє поширити отримане рішення на МК інших виробників, що підтверджує практичну цінність розроблених патернів

Ключові слова: реальний час, керуючий контролер, кінцевий автомат, мікроконтролер Cortex-M, шаблон Стан

UDC 004.451:004.354-022
DOI: 10.15587/1729-4061.2020.205377

DEVELOPMENT OF TYPICAL "STATE" SOFTWARE PATTERNS FOR CORTEX-M MICROCONTROLLERS IN REAL TIME

P. Katin
PhD, Associate Professor*
E-mail: p.katin@kpi.ua
V. Chmelov
PhD, Associate Professor**
E-mail: viacheslavchmelov@gmail.com
V. Shemaev
Doctor Military Sciences, Professor*
E-mail: shemaev@niss.gov.ua
*Department of Automation and Control in Technical Systems***
Department of Radio Engineering Devices and Systems*
***National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute"
Peremohy ave., 37, Kyiv, Ukraine, 03056

Received date 18.04.2020
Accepted date 16.06.2020
Published date 26.06.2020

1. Introduction

There are many options to categorize software for microcontrollers. We shall use one of the variants for the informal classification of real-time operating systems (RTOS) for microcontrollers (MC). It was proposed by the Massachusetts Institute of Technology (MIT). This classification takes into consideration the software for real-time technical systems (RTTS) based on microcontrollers and microprocessors [1, 2]. According to this classification, there are 4 classes of hard-real time operating systems:

– the pattern template based on Polled Loop Systems;

– the pattern template of software based on the microcontroller interruption system (Interrupt Driven);

– the simple patterns of real-time operating systems based on a multi-tasking core, such as SafeRTOS, the IEC 61508 standard-certified, µC/OS-II, RTXC Quadr, and others;

– a full-featured operating system (Full Featured RTOS) maintaining the POSIX standard (portable operating system interface). For a given option, there are typical RTOS standards such as POSIX 1003.1a, the POSIX 1003.1b standard, the POSIX 1003.1c standard. Aviation engineering employs the RTOS of the DO-178B, ARINC-653 standards. The European aviation standard ED-12B is analogous to DO-178B.

Many modern real-time software and hardware technical systems (RTTS) are built on the basis of a master controller.

More complex systems typically consist of a main controller based on the Cortex-A (Cortex-M) architecture microcontrollers and auxiliary controllers (based on the Cortex-M MCs or simpler 8-, 16-bit MCs). In most cases, the auxiliary controller in a circuit of actuator and sensor controllers (ASCs) may be subject to higher requirements for the speed of computation and the unification of MCs from different manufacturers. The software solution is subject to requirements for the ease of source code maintenance. In addition, the issue of mathematical modeling is important.

At present, one of the acceptable solutions for the creation of ASC hardware is the Cortex-M architecture MCs due to their relatively high power and low price. The price of these 32-bit Cortex-M architecture microcontrollers is comparable to some 8-bit MCs for similar purposes while and the speed of operation is much greater. Using the Cortex-M architecture makes it possible to unify the source code for the MCs from different manufacturers.

The ASC software can be implemented on the basis of the first three classes of RTOS, namely, based on the Polled Loop Systems, based on the Interrupt Driven microcontroller system, or on the basis of simple real-time operating systems based on a multi-tasking core [1, 2]. The use of a full-featured RTOS [1, 2] for ASC is not acceptable due to additional unreasonable costs.

We shall consider the pros and cons of using a multi-tasking-core-based RTOS as a base for the ASC software compared to other solutions [1, 2]. The advantage of a multi-tasking-core-based RTOS is the relative simplicity of software development and project support.

At the same time, we can note two drawbacks of the ASC software based on the RTOS that maintains multi-tasking. The first is the relative speed limit of the software compared to solutions based on the Polled Loop Systems or based on the Microcontroller Interrupt Driven system. The second drawback is the need to study the source code of the RTOS for the presence of hidden errors, to test the RTOS for systems requiring great reliability (aircraft, missile technology, medicine, etc.). Relatively simple solutions based on the Polled Loop Systems or the Interrupt Driven systems are easier to find hidden programming errors.

One can achieve greater performance speed of the ASC software, compared to a multi-tasking RTOS, by using the Polled Loop Systems or on the basis of the microcontroller Interrupt Driven system [1, 2]. However, developing the software without patterns makes it difficult to unify, to maintain the code, and to model mathematically. Therefore, it is a relevant task to develop typical State design patterns for the Cortex-M architecture MCs to address the above shortcomings related to unification, maintenance, and mathematical modeling.

The Cortex-M architecture microcontrollers, for which patterns are designed, are produced by many manufacturers and are now widely used in equipment. Using the Universal Cortex-Microcontroller Software Interface System (CMSIS) libraries makes it possible to adapt the obtained patterns to a large number of Cortex-M architecture microcontrollers from many manufacturers. That makes it possible to apply the resulting solution for a broad range of Cortex-M architecture MCs and thus ensure unification. Therefore, it seems expedient to explore a given field of research as the developed patterns could be widely used in practice to create the ASC software libraries based on a wide range of the Cortex-M architecture MCs.

2. Literature review and problem statement

It is shown in [1, 2] that in most cases a real-time operating system from any of the 4 classes must be described along with the hardware and cannot be considered separately from the physical environment and the processes it manages. The authors revealed the hardware and software problems related to increasing the speed of software, standardizing, maintaining, and mathematical modeling.

Paper [3] explores the possibility of increasing the speed of the software by increasing the number of MC computational cores. At the same time, the task of increasing the speed of performance at minimal cost remains unresolved. This is due to that the cost of multi-core MCs is much higher than the cost of single-core MCs. An option to solve this problem is to use a relatively inexpensive and universal MC architecture.

Study [4] reports the results of achieving the maximum switching speed by using an 8-bit microcontroller. However, such hardware implies the difficulty of unifying the source code of the software. The reason for this is that the architecture of 8-bit MCs is typically not universal for several manufacturing firms.

The general results of RTTS testing based on high-speed controllers are described in [5]. At the same time, there are no specific implementations of the software and hardware of RTTS controllers.

The results of the development of a controller using the Cortex-M architecture for scalable solutions are reported in [6]. However, the issue of mathematical modeling remains unresolved, which complicates the possibility of mathematical analysis of the ASC software.

A system of food production by evaporating the raw materials was implemented in [7]. The solution was obtained on the basis of a Cortex-A MC. At the same time, the issue of achieving an acceptable cost of the ASC prototype was not resolved. This is due to the high cost of the Cortex-M architecture MCs.

The infrastructure management solutions built on Internet technology are outlined in [8]. The implementation of high-speed software remains a problematic issue. This is due to that Internet technologies are a limiting factor in a hard-real-time mode.

The software that makes it possible to reach high speeds for image processing is described in [9]. The authors do not address the mathematical notation and software for formalizing and switching the RTTS states within the State pattern. A solution to the problem is to develop typical State templates (patterns) for the Cortex-M architecture MCs.

Work [10] reveals the implementation of the software based on the State pattern in the object-oriented programming (OOP) variant. The downside of this solution is the redundant code and computational requirements, which is not always acceptable for ASC. Therefore, the possibility of eliminating the above flaw is to develop a pattern in a simpler, procedural version.

A study of the software implementation of the State pattern in OOP for low-power microcontrollers in a generalized form was conducted in [11]. At the same time, the problem of universalization of the received solutions for a wide range of MCs was not solved. A solution to this problem at present is the development of the State pattern for the universal Cortex-M architecture MCs.

A study of the software implementation of the finite state machine in OOP was carried out in [12]. The issues of the mathematical modeling of the software remained unresolved. One solution is to bring the developed pattern to the mathematical model of the finite state machine.

The most high-speed version of the State pattern hardware implementation is offered in [13]. However, at the highest potential speed, a given solution cannot be quickly reprogrammed and reconfigured, which is its main drawback. The most acceptable option is to develop a software pattern for MC.

The mathematical theory of the description of the finite state machine is set out in [14] but there are no software examples, nor any connection to practical tasks.

The general descriptions of the finite state machine software templates are outlined in [15]. However, there are no examples of practical software solutions.

The State patterns designs and their software implementation were developed in [16]. In this case, the solutions are

given in the OOP version, which entails the shortcomings specified for [11]. A solution to this problem is to develop a universal State pattern for a wide range of MCs.

A solution using RTOS is proposed in [17]. At the same time, there is a potential drawback – reducing the software speed by switching the RTOS context. To address this shortcoming, one can use solutions based on the Polled Loop Systems or based on the microcontroller Interrupt Driven system.

One of the solutions to the above issues concerning the hardware of the ASC prototype, outlined in [3–8] is the use of relatively inexpensive general-purpose Cortex-M architecture MCs.

A variant of solving the tasks related to the software of the ASC prototype, outlined in [9–17], concerning the issues of increasing the software speed, unification, maintenance, and mathematical modeling, is the development of typical software State templates (patterns) for the Cortex-M architecture MCs. These patterns can be implemented in the Polled Loop System variant or through the microcontroller Interrupt Driven system.

3. The aim and objectives of the study

The aim of this study is to develop a software solution to increase the speed of the ASC prototype compared to RTOS. At the same time, the result to be obtained should ensure the unification of the source code, facilitate the maintenance process, as well as the mathematical modeling of the software.

To accomplish the aim, the following tasks have been set:

– to develop and implement a typical State software pattern for ASC based on Cortex-M in a procedural paradigm, maximally adapted for the mathematical model of the finite state machine;

– to analyze the shortcomings of a typical pattern; in order to eliminate them, design a State pattern for ASC based on Cortex-M in the form of a linked list (a typical construct of the C language), to maximally adapt the pattern for the mathematical model of the finite state machine;

– to objectively assess the speed of the software based on the developed patterns compared to a multi-tasking-core-based RTOS [1, 2].

4. Materials and methods for implementing the hardware of an auxiliary high-speed controller

4. 1. The hardware of the Cortex-M-based high-speed controller prototype

Chapter 2 shows that there are options for building the RTTS prototypes based on a single MC. The generalized diagram of such a system is shown in Fig. 1. It depicts the software and hardware components of a real-time technical system. It can be implemented on the basis of MC of any architecture and power, depending on the tasks. This can employ a complex control system that requires the power of Cortex-A MC, for example, to control a video camera.

This system includes sensors, actuators, it enables communication with a personal computer. The software shown in Fig. 1 implies a procedural paradigm.

The more complex RTTSs include actuators, sensors, information transfer systems, etc. The most generalized scheme of such RTTSs is shown in Fig. 2. Unlike Fig. 1, it has an auxiliary ASC (there may be several of them). It manages auxiliary systems and can be built on the basis of the Cortex-M architecture MCs, or the simpler ones, 8-, 16-bit MCs of other architecture. A given ASC is designed to process sensor signals and control actuators and subsystems. All these subsystems and elements imply the increased performance speed, so there are higher requirements for ASC in terms of the software speed. The ASC controller is almost autonomous but can receive control signals from the main controller.

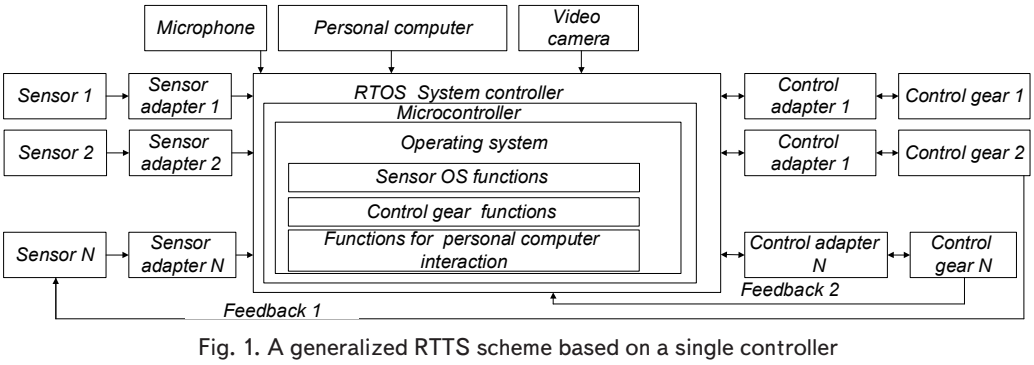

Fig. 1. A generalized RTTS scheme based on a single controller

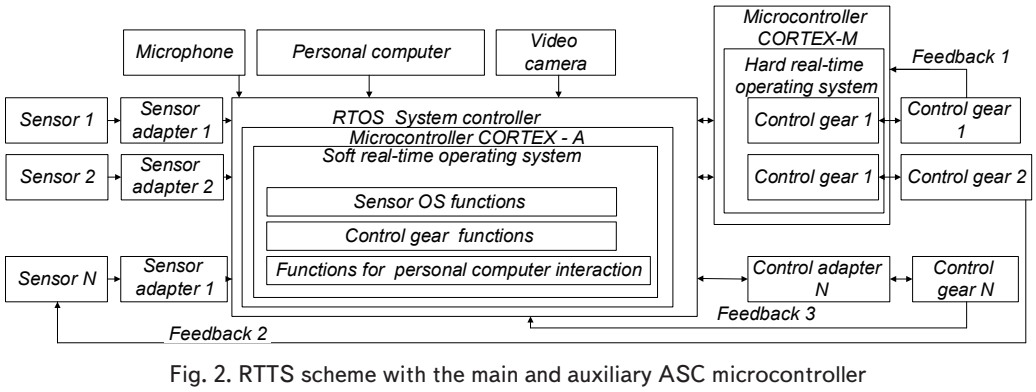

Fig. 2. RTTS scheme with the main and auxiliary ASC microcontroller

The hardware (material part) of ASC can utilize low-cost Cortex-M, such as the STM32F40x or STM32F10x series. MCs from other manufacturers are also possible. These MCs demonstrate the high performance speed and are less costly even compared to 16- or 8-bit MCs. At the same time, the practice of programming the Cortex-M microcontrollers in the C language is much easier than that of 16- or 8-bit MCs. The RTTS software in Fig. 2 includes the main controller's software and the high-speed ASC software. The developed typical State software patterns are designed exactly for ASC.

As previously stated, the cost of the Cortex-M MCs is approaching the cost of low-power microcontrollers. Therefore, the ASC hardware can be built on the basis of the Cortex-M architecture MCs. The software for ASC can be built on the basis of the State template or based on a multitasking RTOS according to the classification given in [1, 2]. The Full Featured RTOS operating systems [1, 2] based on the Cortex-M MCs are not used to build ASC due to unreasonable cost increases.

In the development of the patterns, the standard 32-bit microcontroller STM32F103 was used as the ASC hardware. The clock frequency of the microcontroller was 72 MHz. Power voltage was 3.3 V. The indoor temperature during testing was within 20–25 °C. The more precise characteristics of the microcontroller are given in [18].

The software development method is to use the Keil environment, the CMSIS typical libraries, and to evaluate the resulting solutions at debugging.

Although STM32F103 is not the most powerful MC, the software tested on STM32F103 can be easily adapted to similar and improved MCs. The software can be standardized for the Cortex-M architecture MCs and for other vendors through the CMSIS libraries.

4. 2. A mathematical model for formalizing the State pattern

There are several options for describing the model of the finite state machine. The basic description of the finite state machine chosen for the development of a State design pattern was the variant proposed in [19] in the form of sets and recurrent formulae. This option is more appropriate to describe nearly implemented software.

The finite state machine is set by the following recurrent formulae:

$$\left.\begin{aligned} z_{i+1} &= f(x_{i+1}, z_i, y_i);\\ y_{i+1} &= \phi(x_{i+1}, z_i, y_i),\end{aligned}\right\} \quad (1)$$

where the sequence x_0, x_1, x_2,..., x_n, using the terminology from [19], forms the input word and can be considered as a set **X**; the sequence of values z_0, z_1, z_2,..., z_k, using the terminology from [19], forms the word of states and can be considered as a set of **Z** states; the sequence of values y_0, y_1, y_2,..., y_l, using the terminology from [19], forms an output word and can be considered as a set of outputs **Y**. The pair (x_0, y_0) is termed the original state for the finite state machine. If functions (1) are not dependent on **Y** [19], (1) is transformed to the form

$$\left.\begin{aligned} z_{i+1} &= f(x_{i+1}, z_i);\\ y_{i+1} &= \phi(x_{i+1}, z_i).\end{aligned}\right\} \quad (2)$$

To compare other variants of the finite state machines, we shall write (1) and (2) in the form of a classic representation as a totality

$$\left(\mathbf{X}, \mathbf{Y}, \mathbf{Z}, f(x_{i+1}, z_i), \varphi(x_{i+1}, z_i), \mathbf{Z}_0\right). \quad (3)$$

Thus, (3) includes all the classic elements of the mathematical model of a finite state machine. A given formula provides an opportunity to implement a mathematical model in the form of a State software pattern for ASC in a procedural paradigm in the classic version of the Polled Loop Systems or based on the microcontroller Interrupt Driven system according to the classification given in [1, 2].

5. Results of the development of State software pattern for a controller

5. 1. Software implementation of the typical State software pattern in the form of the software for ASC in the procedural paradigm

The first step implies the software that represents a typical State software pattern for the ACS based on Cortex-M-based in the procedural paradigm of the classic variant of the polled loop in real time. A special feature is that it is maximally adapted to the mathematical model of a finite state machine (3).

Next, we show the result of the development of a header file for the software, which demonstrates the presence of all three sets in formula (3). A given file demonstrates exclusively the idea of the developed State software pattern in the procedural paradigm, in the variant of Polled Loop Systems in accordance with the classification of RTOS given in [1, 2].

The pattern can be implemented in the peripheral interruption function or a system timer of the Cortex-M series, that is, in the microcontroller Interrupt Driven variant or in a polled loop variant [1, 2]. A given solution makes it possible to reach the maximum speed of computations for ASC as it does not contain additional software delays, caused by the operation of the RTOS core elements.

To improve the readability of the obtained results, the typical formatting of the C programming language was slightly modified. During the development, we identified a global CurrentState variable. It stores the current value of the ASC state. Thus, below is the classical software implementation of mathematical model (3). It is implemented in the procedural variant. The developed header file that demonstrates the idea of a typical State software pattern for ASC based on Cortex-M in the procedural paradigm is shown further.

```
#include "stm32f10x.h"
#include "stm32f10x_tim.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_rcc.h"
#define _KEYPAD_NO_PRESSED 0xFF

enum Z_States CurrentState;
enum X_InputSignal InputsSig;

enum Z_States
    {// state from keyboard
    STATE_S = '*',STATE_RES = '#',STATE_0 = '0',STATE_9 = '9',
    STATE_8 = '8',STATE_7 = '7',STATE_6 = '6',STATE_5 = '5',
    STATE_4 = '4', STATE_3 = '3', STATE_2 = '2',STATE_1 = '1'};
```

```
enum X_InputSignal
{// input signals from keyboard
    SIG_STATE_S = '*',SIG_STATE_RES = '#',SIG_STATE_0 = '0',
    SIG_STATE_9 = '9',SIG_STATE_8 = '8',SIG_STATE_7 = '7',
    SIG_STATE_6 = '6',SIG_STATE_5 = '5',SIG_STATE_4 = '4',
    SIG_STATE_3 = '3',SIG_STATE_2 = '2',SIG_STATE_1 = '1', };

struct Y_OutSignal
{// output signal
    char (*ptr_state_0)(void);
    char (*ptr_state_1)(void);
    char (*ptr_state_2)(void);
} OutSignal= {handle_state_0,handle_state_1, handle_state_2};

char handle_state_0(void); char (*ptr_state_0)(void);
char handle_state_1(void); char (*ptr_state_1)(void);
char handle_state_2(void); char (*ptr_state_2)(void);

void init_state(enum Z_States INIT_STATE,enum X_InputSignal _0_INP);
uint32_t _FSM_simple(void);
void _FSM_switch(void);
enum X_InputSignal input_from_external(void);
```

In the header file, we use the enumerator enum Z_State to describe the set of states, forming the finite set of the internal ASC states, which corresponds to (3). If necessary, a developer can increase this number to the required value.

The enumerator enum X_InputSignal is also used to describe the set of input signals, which corresponds to the set of inputs (3).

To form the set of output signals, the structure struct Y_OutSignal is used, whose elements are the indicators on the function, which corresponds to the set of output signals (3).

The functions void FSM_simple(void) and void FSM_switch(void) correspond to the transition functions of the mathematical model of a finite state machine (3).

If necessary, a developer can increase the number of functions and states to the required value. Development practice has shown that the use of enumerators is convenient for a relatively small number of states. The implementation file and the test run variant file can be built independently using a well-known model solution.

Thus, a given header file demonstrates the developed typical State software pattern for an ASC based on Cortex-M in the procedural paradigm in the variant of a polled loop system. The solution is maximally close to the mathematical model of the finite state machine.

5. 2. Software implementation of the typical State pattern for the Cortex-M microcontroller in the form of a linked list

Developed in chapter 5. 1, the typical State software pattern for a Cortex-M-based ACS in the procedural paradigm in the classic variant of the polled loop systems has one drawback. It can occur at a large number of states in the form of a long polled cycle in a state-busting algorithm, such as a switch operator. This makes it more difficult to read the code at a large number of states and to maintain it. Next, a code is offered that represents the result of the development of the State pattern in the C programming language. A special feature of the solution is a link between the pattern and mathematical model (3) and the elimination of the specified flaw by using the mechanism of a linked list (the typical construct of the C language).

In the developed solution for Cortex-M, a linked list is used that eliminates the long polling cycle in a brute force algorithm and implements the functions of transition and output (2).

The source code demonstrates only the very idea of using a linked list in the development of a typical State pattern in the form of software for ASC based on Cortex-M. Therefore, the software is as simplified as possible. In the developed example, the transition from state to state is carried out in sync, without a random input signal. This is done with the aim of explaining the idea of a typical State pattern in the form of software for ASC based on Cortex-M in the form of a linked list (the typical construct of the C language).

The example includes a modification of the model formatting of the C language to improve the readability of the software solution. The software was tested for the microcontroller stm32f103.

As one can see from the header file below, the code contains all the elements of the finite state machine (3). In this variant, the sets are implemented as structures, which slightly increases the volume of the source code. The relationship between the solution obtained in a given header file and mathematical model (3) is detailed in chapter 5. 1, with minor changes.

```
#include “stm32f10x.h”
#include “stm32f10x_tim.h”
#include “stm32f10x_gpio.h”
#include “stm32f10x_rcc.h”

void init_state_structsPK(void);
void init_state_structsPK_better(void);
void init_state_structs(void);

void _FSM_linked(void);
void FSM_to_step_PK(void);

enum Z_State
{// state
    STATE_S = ‘*’,STATE_RES = ‘#’,STATE_0 = ‘0’,STATE_9 = ‘9’,
    STATE_8 = ‘8’,STATE_7 = ‘7’,STATE_6 = ‘6’,STATE_5 = ‘5’,
    STATE_4 = ‘4’, STATE_3 = ‘3’, STATE_2 = ‘2’,STATE_1 = ‘1’
};

enum X_InputSignal
{// input signals from keyboard
    SIG_STATE_S = ‘*’,SIG_STATE_RES = ‘#’,SIG_STATE_0 = ‘0’,
    SIG_STATE_9 = ‘9’,SIG_STATE_8 = ‘8’,SIG_STATE_7 = ‘7’,
    SIG_STATE_6 = ‘6’,SIG_STATE_5 = ‘5’,SIG_STATE_4 = ‘4’,
    SIG_STATE_3 = ‘3’,SIG_STATE_2 = ‘2’,SIG_STATE_1 = ‘1’,
// add signals
    SIG_NO_ST_0 = ‘/’,SIG_NO_ST_1 = ‘|’,SIG_NO_ST_2 = ‘+’,
};
```

```
typedef struct Z_State_struct
    {// state from keyboard
    int STATE_S; int STATE_RES; int STATE_0; int STATE_9;
    int STATE_8; int STATE_7; int STATE_6; int STATE_5 ;
    int STATE_4; int STATE_3; int STATE_2; int STATE_1 ;
}z_state;

typedef struct X_InputSignal_struct
    {// input signals from keyboard
    int SIG_STATE_S; int SIG_STATE_RES; int SIG_STATE_0; int SIG_STATE_9;
    int SIG_STATE_8; int SIG_STATE_7; int SIG_STATE_6; int SIG_STATE_5 ;
    int SIG_STATE_4; int SIG_STATE_3; int SIG_STATE_2; int SIG_STATE_1 ;
}x_state;

// Y_OutSignal
    void handle_state_0(void); //void (*ptr_state_0)(void);
    void handle_state_1(void); //void (*ptr_state_1)(void);
    void handle_state_2(void); //void (*ptr_state_2)(void);
    void handle_state_3(void); //void (*ptr_state_3)(void);
    void handle_state_4(void); //void (*ptr_state_4)(void);
    void handle_state_5(void); //void (*ptr_state_5)(void);
```

The basic idea behind the development of a model State pattern in the form of software for ASC based on Cortex-M in the form of a linked list is shown later in the implementation file. The feature that distinguishes a given solution from the classic pattern with an infinite polling cycle is that the structure state_infoPK contains all the elements of (3) and is part of a linked list. The linked list makes it possible to navigate it without the cumbersome switch design.

Next, the source code includes a function to initiate the entire linked list init_state_structsPK, which creates the sets of states and all elements of the mathematical model of the finite state machine (3). A given solution is demonstrative and requires further refinement for registration in the form of a library for ASC.

Transfer and output functions are designed as a FSM_to_step_PK() function, which makes it possible to navigate the list without using a simple busting algorithm, which significantly reduces the source code of the demo software.

The software example given in the source code below shows the State pattern testing using a linked list and demonstrates that one does not need a polling cycle to switch the state. The file contains an entry point to the software. It is designed solely to demonstrate the idea and the feasibility of implementing a typical State pattern for ASC based on Cortex-M in the form of a linked list (the typical construct of the C programming language). The overall implementation of the template is maximally adapted to the mathematical model of the finite state machine (3). Pattern testing in a given example is performed at launch based on the polled loop system in accordance with the RTOS classification [1, 2].

The pattern can be tested in the peripheral interruption processing function or the Cortex-M series MC system timer based on the microcontroller Interrupt Driven architecture [1, 2].

```
#include "FSM.h"
#include "tm1637.h"
#include "delay.h"
#include <stdio.h> /* printf, scanf, NULL */
#include <stdlib.h> /* malloc, free, rand */
uint32_t CurrentStateInit ='0';

const struct Z_State_struct Z_States={'*','#', '0', '9', '8', '7','6','5', '4','3', '2', '1'};
const struct X_InputSignal_struct X_InputList={'*','#', '0', '9', '8', '7','6','5', '4','3'};

int Z_CurrentState;
int X_CurrentInput;
enum Z_State CurrentState;
enum X_InputSignal InputsSig;

void Y_handle_state_0(void)
{TM1637_clearDisplay(); TM1637_display(0,Z_States.STATE_0);}
void Y_handle_state_1(void)
{TM1637_clearDisplay(); TM1637_display(0,Z_States.STATE_1);}
void  Y_handle_state_2(void)  {TM1637_clearDisplay();  TM1637_display(0,2);}
void  Y_handle_state_3(void)  {TM1637_clearDisplay();  TM1637_display(1,3);}
void  Y_handle_state_4(void)  {TM1637_clearDisplay();  TM1637_display(1,4);}
void  Y_handle_state_5(void)  {TM1637_clearDisplay();  TM1637_display(2,5);}
void  Y_handle_state_6(void)  {TM1637_clearDisplay();  TM1637_display(2,6);}
void  Y_handle_state_7(void)  {TM1637_clearDisplay();  TM1637_display(2,7);}
void  Y_handle_state_8(void)  {TM1637_clearDisplay();  TM1637_display(3,8);}
void  Y_handle_state_9(void)  {TM1637_clearDisplay();  TM1637_display(3,9);}
    .....

typedef struct state_infoPK
    { int val;
    struct Z_State_struct z_state;// enums is not suitable there
    struct X_InputSignal_struct x_state;
    void (*Y_handle)(void);
    struct state_infoPK *next;
    } my_node_t;

my_node_t *head = NULL;

void FSM_to_step_PK(void)
    {
    my_node_t * current = head;
```

```
while (current != NULL) {
    current->Y_handle();
    current = current->next;
    delay_ms(250); }}

void init_state_structsPK(void)
    {
    head = (my_node_t*)malloc(sizeof(my_node_t));
if (head == NULL){while(1);}
    head->val = 0;
    head->Y_handle=Y_handle_state_0;

    head->next = (my_node_t *) malloc(sizeof(my_node_t));
    head->next->val = 1;
    head->next->Y_handle=Y_handle_state_1;

    head->next->next = (my_node_t *) malloc(sizeof(my_node_t));
    head->next->next->val = 2;
    head->next->next->Y_handle=Y_handle_state_2;

    head->next->next->next = (my_node_t *) malloc(sizeof(my_node_t));
    head->next->next->next->val = 3;
    head->next->next->next->Y_handle=Y_handle_state_3;
    }

void init_state_structsPK_better(void)
    {
    head = (my_node_t*)malloc(sizeof(my_node_t));
    int size_str = sizeof(head);
if (head == NULL){while(1);}
    (*head).val = 0;
    (*head).Y_handle=Y_handle_state_0;

    (*(head)).next = (my_node_t *) malloc(sizeof(my_node_t));
    (*(head+size_str)).val = 1;
    (*(head+size_str)).Y_handle=Y_handle_state_1; }
```

The STM32F10 microcontroller was used for testing. The software was developed in the Keil environment.

```
#include "delay.h"
#include "FSM.h"
#include "tm1637.h"
int main()
    {
    TM1637_init();
    TM1637_brightness(BRIGHTEST);
    TM1637_clearDisplay();
    init_state_structsPK();
    FSM_to_step_PK();
while(1)
    {
    delay_ms(250);
    FSM_to_step_PK();}}
```

5. 3. Results of estimating the speed of the developed patterns compared to the real-time operating system

As previously stated, the typical State software patterns for Cortex-M to implement the ASC software make it possible to reach a faster software performance speed compared to RTOS-based solutions. This is because the patterns do not contain additional software delays, caused by the work of the RTOS core elements. To confirm this, the comparison was performed between the software speed of the developed pattern and the RTOS-based solutions.

For the purpose of comparison, a procedure of speed estimation was devised based on the calculation of clocks under the mode of step-by-step debugging of MC in the Keil programming environment.

The first feature of the procedure is the units to estimate the time of the software operation. The software operation time was evaluated using the capabilities of the Keil programming environment. In this case, we measured not the time of operation between the steps of the software but the number of clocks recorded in the Cycle Count register (DWT_CYCCNT) register. Hereafter, we term this value the number of cycles. A given value can be determined in a step-by-step debugging in the Keil programming environment. Such a capability of the programming environment is demonstrated in Fig. 3, where the arrow shows the value of the States counter, which records the number of cycles during step-by-step debugging.

This approach provides an opportunity to summarize the results of the evaluation of the software speed on the Cortex-M MCs from other manufacturers. Since the Cortex-M MCs may have different clock rates, the time estimate obtained for a single MC cannot be summarized on MCs from different manufacturers. And the estimate of the number of cycles makes it possible to compare the speed for different software solutions and summarize it for the Cortex-M MCs by different manufacturers supporting the CMSIS libraries in the Keil environment.

The next feature of the procedure is the choice of RTOS. The real-time CMSIS-RTOS Variant 1.03 operating system was chosen for comparison. It belongs to a multi-tasking-core-based RTOS according to the classification given in [1, 2]. For the ASC technical solution, the full-featured RTOS, according to the classification given in [1, 2], is not intended to be used. That is why we chose the CMSIS-RTOS Variant 1.03. This choice is also due to the fact that the developer of a given RTOS is the Keil developer. This allows us to argue that the RTOS works properly, it has optimal characteristics and is adapted to the Cortex-M architecture MC for most manufacturers. This makes it possible to extend the results for other multi-tasking RTOS according to the classification given in [1, 2].

The third feature of the procedure is to select the location of the software's stop points during the step-by-step run and measure the number of cycles recorded in the Cycle Count (DWT_CYCCNT) register to compare different software solutions. The main element of the comparison for RTOS-based software and that based on the developed pattern is the estimate of the number of cycles. Moreover, the RTOS measured the number of cycles between exiting one function of one thread and entering another function of another thread when switching threads.

In the developed pattern, we measured the number of cycles between exiting one function of one state and entering another function of another state as one moves from one

state to another. In both cases, an empty __NOP() operator is employed to properly assess the function.

Such an approach makes it possible to most accurately assess the difference in the operation time of practical software solutions based on RTOS and that on the basis of the developed pattern.

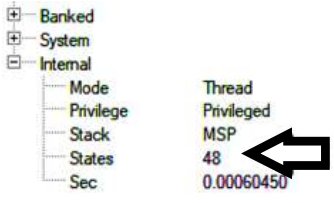

Fig. 3. States register in the Keil programming environment (Registers work window)

There was no statistical estimate of the number of cycles (States, Fig. 3) because in multiple experiments the States' value for the same stop points did not change. This is the peculiarity of a hard time mode RTOS.

This is followed by an example of a procedure to determine the number of cycles between exiting one function of one thread and entering another function of a different thread when one switches threads for CMSIS-RTOS Variant 1.03.

At the initial stage, we selected the 32-bit microcontroller STM32F103 of the Cortex-M architecture. To implement the procedure, we used a demo source code, shown below. It represents the demonstration of the parallel execution of the two threads.

In the first step, stop points were set in the function **void** test_thread1(void const *argument) and in **void** test_thread1(void const *argument).

```
#include "STM32F10x.h"
#include <cmsis_os.h>
#include "STM32F10x_rcc.h"
osThreadId test_ID1; //Identifiers of threads 1 and 2
osThreadId test_ID2;

void test_thread1(void const *argument)
    {__NOP(); }//Function that is implemented in thread 1
void test_thread2(void const *argument)
    {__NOP();}//Function that is implemented in thread 2

osThreadDef(test_thread2, osPriorityNormal,1, 0); //Macros settings
osThreadDef(test_thread1, osPriorityNormal,1, 0);

int main(void) {
    //MC periphery configuration (not shown)
    osKernelInitialize(); //Initializing the CMSIS-RTOS Variant 1.03
    // Creation of threads 1 and 2
    test_ID2 = osThreadCreate(osThread(test_thread2), NULL);
    test_ID1 = osThreadCreate(osThread(test_thread1), NULL);
    osKernelStart(); //Launch the core CMSIS-RTOS Variant 1.03
while (1) { ; }
    }
```

The second step was to determine the number of States in the step-by-step debugging. The number of States was determined between the stop points in the first function

void test_thread1(void const *argument)

and in the second function

void test_thread1(void const *argument)

in the process of thread switching.

The third step summarized the results obtained, which are given in Table 1. They are shown in relative cycles (States). Statistical treatment of the results was not carried out because during repeated repetitions of debugging, more than 50 times, the same number of cycles were produced. A given solution can be generalized for other Cortex-M architecture MCs by other manufacturers. The Sec field (Fig. 3) was used for approximate time assessment.

The cycle counter value was similarly compared when switching between the states of the developed patterns, namely:

– software implementation of the typical State software pattern for ASC based on Cortex-M in the procedural paradigm in the classic variant of the polled loop systems, line 2, Table 1;

– software implementation of the typical State pattern for the Cortex-M microcontroller in the form of a linked list, line 3, Table 1. Both patterns are brought to the mathematical model of the finite state machine in accordance with the developed solution.

The results of comparing the relative speed of the software based on RTOS and that based on the developed patterns are summarized in Table 1.

The values given in column 3 are closer to development practice because they show the actual number of cycles in the course of exiting one function (state or thread) and entering another function (state or thread). A given value in RTOS is much higher than the value of switching between the states. Because the periphery was set up the same in all three cases at debugging, one of the main sources of delay is the switch time between these functions. Therefore, a given procedure makes it possible to correctly compare the software speed.

Table 1

The results of comparing the relative speed of software solutions

No.	Software solution	The number of States at debugging	Estimated time of delay (approximate)
1.	Real-time operating system CMSIS-RTOS Variant 1.03	564	7×10^{-5} s
2.	Software implementation of the typical State pattern in the form of software for ASC in the procedural paradigm	42	5×10^{-6} s
3.	Software implementation of the typical State pattern for a Cortex-M microcontroller in the form of a linked list	59	7×10^{-6} s

6. Discussion of the developed State software patterns

The reported solutions provide an opportunity to implement the software for ASC based on a Cortex-M MC in

the procedural paradigm in the State pattern variant. As it follows from Table 1, the number of cycles when switching between the states of the obtained patterns is about 10 times less than the number of switches between the RTOS threads in CMSIS-RTOS Variant 1.03. The testing was carried out at the same functionality and hardware base. Thus, the switch rate between the states of the resulting patterns is about 10 times greater than the switch rate between the RTOS threads in CMSIS-RTOS Variant 1.03. The measurements are given in relative units and make it possible to generalize a given solution for Cortex-M MCs from different manufacturers.

The peculiarity of the first solution of the State software pattern for use in ASC is that the software template is maximally adapted to the mathematical model of the finite state machine (3). This allows mathematical analysis.

The feature of the second solution is to use a linked list when implementing the finite state machine. As one can see from Table 1, the number of cycles (States) in the variant with a linked list increases slightly, while the potential readability of the code and its brevity significantly improves.

An additional focus of this study is to refine the State patterns for the typical libraries and actual software solutions as the proposed variant is a demonstration of the technology. In addition, it is necessary to investigate other potential possibilities to increase the speed of the software related to the hardware features of Cortex-M MCs.

The resulting solutions should be used in the following cases:

– the limitations of using the built-in memory of a microcontroller, for example, the core of an operating system takes up Read-Only Memory (ROM) and Random-Access Memory (RAM);

– the speed of patterns performance should be greater than it is allowed by an RTOS;

– the patterns should be used with a relatively large number of sensors and actuators under the ACS control in the RTTS structure.

7. Conclusions

1. A State software pattern has been developed for use in ASC. Unlike well-known solutions, it is built on the basis of the libraries of the Cortex M architecture microcontrollers and is maximally adapted to the mathematical model of the finite state machine. This makes the resulting solution easy to adapt to a wide range of Cortex-M MCs from different manufacturers, ensuring code unification and simplifying the maintenance process. The maximum adaptability to a mathematical model allows using mathematical modeling methods during development. Applying a given pattern provides faster speed of the software solution than the solutions based on RTOS.

2. A drawback in the developed State software pattern for use in ASC has been revealed. It can occur at a large number of states in the software in the form of a long polling cycle in the brute force algorithm, such as using a switch operator. This requires a description of the entire functionality of the software in the busting operator, increasing the length of the programming code.

To address this shortcoming, a State pattern has been developed for ACS based on Cortex-M in the form of a linked list (the typical construct of the C language). A given pattern also ensures a faster software performance compared to those based on RTOS; it has been built on the basis of the Cortex M architecture microcontroller libraries. In this case, the pattern makes it possible to use the transition over the elements of the linked list instead of busting states, which significantly reduces the volume of the source code as the number of states increases.

The pattern has been maximally adapted to the mathematical model of the finite state machine. The adaptation to the model of the finite state machine implies that the pattern includes all elements of the mathematical model of the finite state machine, namely: a set of internal states, the sets of input signals, output signals, input functions and output functions. These elements have appropriate designations in the source code and can be used for mathematical analysis.

3. The developed State patterns for use in ASC provide a higher speed compared to the RTOS-based software.

To objectively evaluate the developed patterns compared to solutions based on RTOS, a procedure has been developed and applied that makes it possible to assess the performance speed of different software and is based on the calculation of the number of cycles recorded in the Cycle Count (DWT_CYCCNT) register in the programming environment. Our analysis has shown that the number of cycles at switching between the state functions of the developed patterns is about 10 times less than the number of cycles when switching between the thread functions in RTOS. Consequently, the developed patterns make it possible to increase the speed of ASC operation by about 10 times, which is a positive result of the current study. The increase in speed is due to the lack of computational costs in the patterns that occur in RTOS and are related to the operation of its core.

When developing the source code for patterns, we have demonstrated solely the basic idea of using a linked list for implementing the pattern of a finite state machine. Therefore, it is necessary to develop a software implementation option in the form of software infrastructure and a full-fledged library for practical use in ASC.

References

1. Real Time Operating Systems Lecture (2001). MIT. Available at: http://web.mit.edu/16.070/www/year2001/RTOS27.pdf
2. Real Time Operating Systems. Part II (2001). MIT. Available at: http://web.mit.edu/16.070/www/year2001/RTOS28.pdf
3. Saini, P., Bansal, A., Sharma, A. (2015). Time Critical Multitasking For Multicore Microcontroller Using Xmos® Kit. International Journal of Embedded Systems and Applications, 5 (1), 01–18. doi: https://doi.org/10.5121/ijesa.2015.5101
4. Sadgrove, M. (2011). Microcontroller interrupts for flexible control of time critical tasks in experiments with laser cooled atoms. Available at: https://arxiv.org/pdf/1104.0064.pdf
5. Execution time analysis. Rapita Systems. Available at: https://www.rapitasystems.com/products/features/execution-time-analysis
6. Chen, Z., Chen, J., Zhou, S. (2019). Embedded electronic scale measuring system based on STM32 single chip microcomputer. 2019 Chinese Automation Congress (CAC). doi: https://doi.org/10.1109/cac48633.2019.8997317

7. Bessmertnyy, R. S., Katin, P. Y. (2019). Use of high-performance microcontroller for improving economic efficiency of jem production. Standartyzatsiya. Sertyfikatsiya. Yakist, 3 (115), 69–77.
8. Zhu, W., Wang, Z., Zhang, Z. (2020). Renovation of Automation System Based on Industrial Internet of Things: A Case Study of a Sewage Treatment Plant. Sensors, 20 (8), 2175. doi: https://doi.org/10.3390/s20082175
9. Kasthuri Arachchi, S. P., Shih, T. K., Hakim, N. L. (2020). Modelling a Spatial-Motion Deep Learning Framework to Classify Dynamic Patterns of Videos. Applied Sciences, 10 (4), 1479. doi: https://doi.org/10.3390/app10041479
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J., Booch, G. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Published by Addison-Wesley Professional, 416. Available at: http://www.uml.org.cn/c++/pdf/DesignPatterns.pdf
11. Katin, P. (2017). Development of variant of software architecture implementation for low-power general purpose microcontrollers by finite state machines. EUREKA: Physics and Engineering, 3, 49–54. doi: https://doi.org/10.21303/2461-4262.2017.00361
12. Solodovnikov, A. (2016). Developing method for assessing functional complexity of software information system. EUREKA: Physics and Engineering, 5, 3–9. doi: https://doi.org/10.21303/2461-4262.2016.00157
13. Dietrich, C., Hoffmann, M., Lohmann, D. (2015). Back to the Roots: Implementing the RTOS as a Specialized State Machine. The 11th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications, 7–12. Available at: https://people.mpi-sws.org/~bbb/events/ospert15/pdf/ospert15-p7.pdf
14. Beynon, W. M. (1980). On the structure of free finite state machines. Theoretical Computer Science, 11 (2), 167–180. doi: https://doi.org/10.1016/0304-3975(80)90044-4
15. Adamczyk, P. The Anthology of the Finite State Machine Design Patterns. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.95.838&rep=rep1&type=pdf
16. Andresen, K., Møller-Pedersen, B., Runde, R. K. (2015). Combined Modelling and Programming Support for Composite States and Extensible State Machines. Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development. doi: https://doi.org/10.5220/0005237302310238
17. Prasanna, Ch. S. L., Venkateswara Rao, M. (2012). Implementation of a Scalable μC/OS-II Based Multitasking Monitoring System. International Journal of Computer Science And Technology, 3 (2), 86–89. Available at: http://ijcst.com/vol32/1/prasanna.pdf
18. RM0008 Reference manual STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx and STM32F107xx advanced Arm®-based 32-bit MCUs. Available at: https://www.st.com/resource/en/reference_manual/cd00171190-stm32f101xx-stm32f102xx-stm32f103xx-stm32f105xx-and-stm32f107xx-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf
19. Bloh, A. Sh. (1975). Graf shemy i ih primenenie. Minsk: Visheyshaya shkola, 294.