MINISTRY OF SCIENCE AND HIGHER EDUCATION OF THE RUSSIAN FEDERATION

SAMARA NATIONAL RESEARCH UNIVERSITY
(SAMARA UNIVERSITY)

CMSIS-RTOS FOR CORTEX-M3 MICROCONTROLLERS

Рекомендовано редакционно-издательским советом федерального государственного автономного образовательного учреждения высшего образования «Самарский национальный исследовательский университет имени академика С.П. Королева» в качестве методических указаний для студентов Самарского университета, обучающихся по основным образовательным программам высшего образования по направлениям подготовки 11.04.01 Радиотехника, 24.04.01 Ракетные комплексы и космонавтика

Compilers: *I.A. Kudryavtsev,*
D.V. Kornilin,
O.O. Myakinin

SAMARA
SAMARA UNIVERSITY PUBLISHING HOUSE
2020

УДК 004.382.7(075)+811.111(075)
ББК 32.973.26я7+811.2Англ я7

Compilers: ***I.A. Kudryavtsev, D.V. Kornilin, O.O. Myakinin***

Reviewer: Dr. M.P. Kalaev, Cand. of Tech. Sc.

CMSIS-RTOS for CORTEX-M3 microcontrollers: laboratory training guide / Compilers: *I.A. Kudryavtsev, D.V. Kornilin, O.O. Myakinin*. – Samara: Samara University Publishing House, 2020. – 16 p.

The guide is focused on the development of RTOS-based software for microcontrollers with the Cortex-M3 core. Main techniques of development and debugging with CMSIS-RTOS are demonstrated.

The guidelines are intended for the students of 11.04.01 Radioengineering, 24.04.01 Space Vehicle Systems and Cosmonautics. Laboratory training with CMSIS-RTOS may be performed within the courses «Digital Devices and Microprocessors», «Fundamentals of microprocessor systems and programming microcontrollers».

The guide is prepared at the Department of Laser and Biotechnical Systems.

УДК 004.382.7(075)+811.111(075)
ББК 32.973.26я7+811.2Англ я7

CONTENT

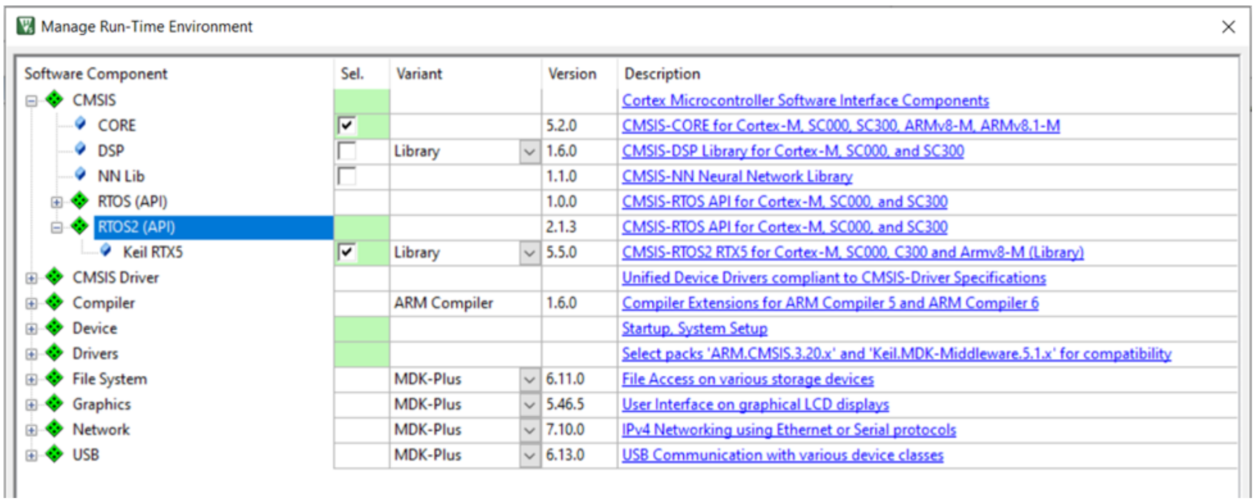
Introduction

Operating system is a popular instrument, used to improve functionality of MCU-based digital devices. Main advantages, provided by operating systems, include easy portability between various platforms, executing several tasks in parallel, easier implementation of trivial functions etc. Real Time Operating Systems (RTOS) are popular because of their compact cores and opportunity to ensure uninterruptible execution of a certain thread, if needed. It is recommended to study [1] for better understanding RTOS fundamentals.

This guide is focused on CMSIS-RTOS v.2, intended for CORTEX architectures. Main functionality of this RTOS will be investigated using CORTEX-M3 MCU (K1986WE92QI) in Keil μVision environment. It is recommended to get aware of CORTEX-M3 core prior to this laboratory training, using [2, 3]. This guide does not pretend to be a manual or reference guide to RTOS, thus all the details of functional implementation should be investigated, for example, in [4].

1 Creating RTOS-based project

CMSIS-RTOS needs few operations for creating a project, using its functionality. Create an empty project, as described in [2] then add RTOS core as shown in fig. 1.


Manage Run-Time Environment

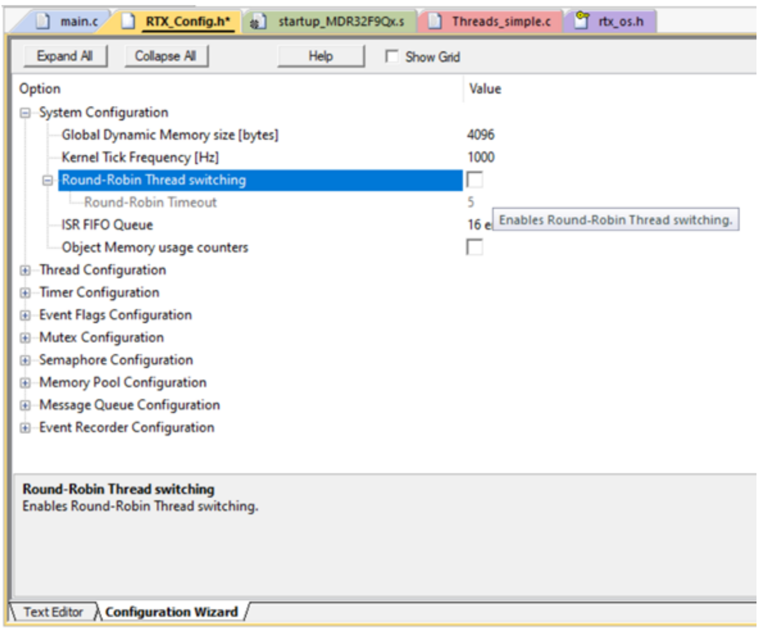
Software Component	Sel.	Variant	Version	Description
CMSIS	🟩			Cortex Microcontroller Software Interface Components
CORE	✓		5.2.0	CMSIS-CORE for Cortex-M, SC000, SC300, ARMv8-M, ARMv8.1-M
DSP	☐	Library	1.6.0	CMSIS-DSP Library for Cortex-M, SC000, and SC300
NN Lib	☐		1.1.0	CMSIS-NN Neural Network Library
RTOS (API)			1.0.0	CMSIS-RTOS API for Cortex-M, SC000, and SC300
RTOS2 (API)	🟩		2.1.3	CMSIS-RTOS API for Cortex-M, SC000, and SC300
Keil RTX5	✓	Library	5.5.0	CMSIS-RTOS2 RTX5 for Cortex-M, SC000, C300 and Armv8-M (Library)
CMSIS Driver				Unified Device Drivers compliant to CMSIS-Driver Specifications
Compiler		ARM Compiler	1.6.0	Compiler Extensions for ARM Compiler 5 and ARM Compiler 6
Device	🟩			Startup, System Setup
Drivers	🟩			Select packs 'ARM.CMSIS.3.20.x' and 'Keil.MDK-Middleware.5.1.x' for compatibility
File System		MDK-Plus	6.11.0	File Access on various storage devices
Graphics		MDK-Plus	5.46.5	User Interface on graphical LCD displays
Network		MDK-Plus	7.10.0	IPv4 Networking using Ethernet or Serial protocols
USB		MDK-Plus	6.13.0	USB Communication with various device classes

Fig. 1. Adding RTOS2 support to a project

In RTOS projects all the user functionality is concentrated in threads, thus **main** function usually performs only initialization of RTOS, core, peripheral modules and user structures. RTOS is initialized by the function **osStatus_t**

osKernelInitialize (void). Then user threads are created by calling **osThreadId_t osThreadNew (osThreadFunc_t func, void * argument, const osThreadAttr_t * attr).** When everything is prepared, it is necessary to start scheduler by calling **osStatus_t osKernelStart (void).** This line will be the last, executed in **main** function. All necessary prototypes are declared in rtx_os.h file, which is necessary to add by **#include <rtx_os.h> line.**

RTOS configuration can be easily done, using **RTX_config.h**. Note that you can edit this file in text form or as a wizard, as shown in fig.2. For our first experiment, do not alter any settings, except unchecking Round-Robin Thread Switching. This Round-Robin switching provides preemptive behavior, discussed below. In our experiment we will use evaluation board, described in [3], with its LCD, used for visualization. All necessary details of using LCD are described in [3], we need only text output in the LCD rows, performed by the function **void LCD_PutString(const char* string, uint8_t y).** In our experiment, we create eight threads, writing into separate lines of the LCD and observe their behavior.



Option	Value
System Configuration	
Global Dynamic Memory size [bytes]	4096
Kernel Tick Frequency [Hz]	1000
Round-Robin Thread switching	☐
Round-Robin Timeout	5
ISR FIFO Queue	16 e
Object Memory usage counters	☐
Thread Configuration	
Timer Configuration	
Event Flags Configuration	
Mutex Configuration	
Semaphore Configuration	
Memory Pool Configuration	
Message Queue Configuration	
Event Recorder Configuration	

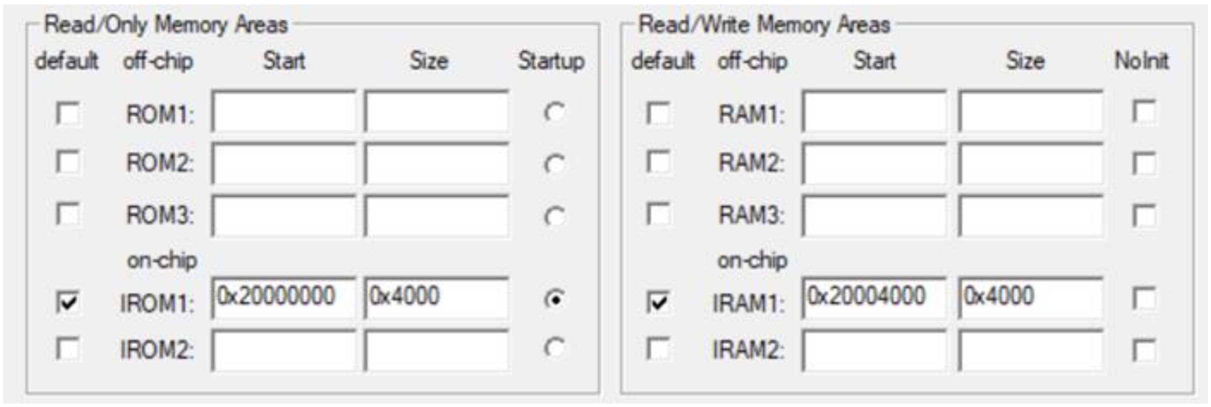
Enables Round-Robin Thread switching.

Round-Robin Thread switching
Enables Round-Robin Thread switching.

Fig. 2. RTOS configuration

Create the file **main.h** and copy there the code from Appendix A, then create another file **threads.c** and copy there the code from Appendix B. You can see that the functions are identical except the line row, where the text is written and the number of task. Calling **osDelay(1);** at the end of the loop is necessary to yield the control to scheduler, however, other ways are possible.

Configure the project for the placement in RAM, as shown in fig.3, build it and run as described in [2]. You can see that all the threads are running synchronously and the numbers, written at LCD, are changing almost simultaneously.



Read/Only Memory Areas

default	off-chip	Start	Size	Startup
☐	ROM1:			○
☐	ROM2:			○
☐	ROM3:			○
	on-chip			
☑	IROM1:	0x20000000	0x4000	●
☐	IROM2:			○

Read/Write Memory Areas

default	off-chip	Start	Size	NoInit
☐	RAM1:			☐
☐	RAM2:			☐
☐	RAM3:			☐
	on-chip			
☑	IRAM1:	0x20004000	0x4000	☐
☐	IRAM2:			☐

Fig. 3. Memory configuration for RAM-based experiment

In this case, the functionality of the threads does not require much CPU time and all the threads are having the same priority, which is not the case for all the tasks. In real conditions, we often have threads, taking various amount of CPU time and maybe having various priorities.

2 Preemptive behavior

Let’s imitate various requirements of CPU time in two threads. Add **for (int i=0; i<1000000; i++);** into the threads One and Two just before the line, writing in LCD. You can see that despite of the slowing only two threads, all other threads became visibly slower. The reason is that the all the threads have to wait when slow ones yield the control. In this case, preemptive behavior can improve the situation. In Round-Robin Thread Switching a thread is suspended by the scheduler after the timeout is expired, regardless of the completion of the operation and the control is transferred to another waiting thread. In this case all the threads get CPU time in accordance with their priority level, thus in case of equal priorities CPU time is shared equally. Put Round-Robin Thread Switching in RTOS settings in checked state (fig. 2), compile the project and run the program. We can see that the threads, where we did not insert any delays, work quickly. However, the LCD works with some glitches. The reason is in thread switching, when writing to LCD by **LCD_PutString** has not completed.

3 Resource sharing

In case like the one, we have now, when the threads should share the resources (Operation of writing to LCD has to be completed before the scheduler switch the threads). There are several solutions: mutexes, critical sections, semaphores and flags.

4 Mutexes

Mutex is a binary semaphore, which can be in one of two states, displaying the availability of a re-source. A thread, trying to access a resource, should try to ac-quire a mutex. If the resource is not available, the thread is suspended, until its re-lease by a current owner.

Declare a mutex by inserting **osMutexId_t MutexId;** into declarative part and create it by **MutexId = osMutexNew(NULL);** in **main.c.** Then insert into thread functions **osMutexAcquire(MutexId,osWaitForever);** prior to calling **LCD_PutString** and release mutex by calling **osMutexRelease(MutexId);** just after it. To make **MutexId** visible in the file **threads.c** it is necessary to declare it in this file with **extern** attribute. You should correct all the thread functions. Run the project and observe changes. Change Round-Robin settings and study, how it affects the performance.

5 Critical sections

Critical sections are not explicitly declared in CMSIS-RTOS, however one can prevent switching by the scheduler with the help of **int32_t osKernelLock (void).** Insert **osKernelLock()** and **osKernelUnlock()** instead of **osMutexAcquire(MutexId,osWaitForever);** and **osMutexRelease(MutexId);** and com-pare with mutex usage.

6 Semaphores

Semaphore is used, when one need to provide restricted access to a resource for several threads (in-stead of atomic access by the only thread, provided by mutex-es). In our experiment, we study the be-havior of a semaphore, limiting access to LCD by four threads. Declare the semaphore (**osSemaphoreId_t osSemaphoreId;**) and create it, calling **osSemaphoreId = osSemaphoreNew (4,4,NULL).**

Insert calls to **osSemaphoreAcquire(osSemaphoreId,osWaitForever);** before **while(1)** into all thread functions. Do not forget to declare your semaphore ID in the file **threads.c** with **extern** attribute. Then insert into any four functions after the call to **osKernelUnlock()** the following block:

```
if(Count>400)
  {
  osSemaphoreRelease(osSemaphoreId);
  return;
  }
```

These four threads will have completed after variable Count reaches 400, then waiting functions can acquire the semaphore. Investigate this behavior.

7 Flags

There are two types of flags in CMSIS-RTOS: event flags and thread flags. You can see more about these objects in [4]. We will use event flags to inform thread about an event (completion of ADC con-version). Details of ADC behavior and settings were described in [3], thus we do not discuss them again here. Copy the code, configuring ADC, from Appendix C to the thread function **Thread_One** (before **while(1)**). Now this thread will wait for event flag, which is set in the interrupt handler. Copy the code of the handler from Appendix C into **main.c** file.

Declare the event (**osEventFlagsId_t event;**) and create it, calling **event = osEventFlagsNew(NULL);** in the **main.c** file. Investigate the code and check data displaying, carefully changing the voltage with the help of TRIM. The code, used in this experiment may require more code memory, than used before. Select appropriate settings for memory configuration, as shown in fig. 3. Data manipulations, executed by threads, can require more space, than allocated by default. You can change this setting, creating thread, as shown below or directly changing settings in **ThreadAttr** structure.

```
ThreadId[0] = osThreadNew(Thread_One,NULL,&(osThreadAttr_t)
{.stack_size=400});
```

The process of selection is not straightforward, but you can see some evaluations, using menu View/Watch Windows/RTX RTOS. It is recommended to have no more than 80% of stack usage.

8 Message Queues

Message queues are described in [4]. They provide a buffer of FIFO type, which can be used for message exchange. We may investigate message queues, basing on the previous experiment. Substitute events by queues using the following lines:

1. Declaration - `osMessageQueueId_t MsgQueue;`
2. Creation - `MsgQueue = osMessageQueueNew(4,4,NULL);`
3. Writing - `uint16_t digit = MDR_ADC->ADC1_RESULT & ADC_RESULT_Msk;`
   `osMessageQueuePut(MsgQueue, &digit, osPriorityNormal, 0);`
4. Reading - `osMessageQueueGet(MsgQueue, &data, NULL, 0U);`

data should be a doubleword (**uint32_t**) placeholder. Make the changes and investigate the code be-havior.

9 Thread priorities

Thread priorities can be set within a range from osPriorityLow to osPriorityRealtime with some additional grades, as described in [4]. Changing this value during thread creation phase, similar to stack size, as shown above, investigate RTOS behavior and make conclusions.

References

1. Cooling, J. Real-time Operating Systems: Book 1. – The Theory (The engineering of real-time embedded systems) / J. Cooling. – Lindentree Associates, 2013.

2. Kudryavtsev, I.A. Software development for Cortex-M3 in Keil µVision: Guide / I.A. Kudryavtsev, D.V. Kornilin, O.O. Myakinin. – Samara: Samara National Research University, 2020. – 23 p.

3.Kudryavtsev, I.A. Studing of peripherals of Cortex-M3 in Keil μVision: Guide / I.A. Kudryavtsev, D.V. Kornilin, O.O. Myakinin. – Samara: Samara National Research University, 2020. – 25 p.

4. CMSIS-RTOS2 Documentation. – URL: https://www.keil.com/pack/doc/CMSIS/RTOS2/html/index.html (accessed: 10/15/2019).

Appendix A

```
#include <rtx_os.h>
#include "lcd.h"

void Thread_One(void *argument);
void Thread_Two(void *argument);
void Thread_Three(void *argument);
void Thread_Four(void *argument);
void Thread_Five(void *argument);
void Thread_Six(void *argument);
void Thread_Seven(void *argument);
void Thread_Eight(void *argument);

static osThreadAttr_t ThreadAttr[8];
osThreadId_t ThreadId[8];

int main()
{
  LCD_Init();
  osKernelInitialize();

ThreadId[0] = osThreadNew(Thread_One,NULL,&ThreadAttr[0]);
ThreadId[1] = osThreadNew(Thread_Two,NULL,&ThreadAttr[1]);
ThreadId[2] = osThreadNew(Thread_Three,NULL,&ThreadAttr[2]);
ThreadId[3] = osThreadNew(Thread_Four,NULL,&ThreadAttr[3]);
ThreadId[4] = osThreadNew(Thread_Five,NULL,&ThreadAttr[4]);
ThreadId[5] = osThreadNew(Thread_Six,NULL,&ThreadAttr[5]);
ThreadId[6] = osThreadNew(Thread_Seven,NULL,&ThreadAttr[6]);
ThreadId[7] = osThreadNew(Thread_Eight,NULL,&ThreadAttr[7]);

osKernelStart();
}
```

Appendix B

```
#include <rtx_os.h>
#include "lcd.h"

void Thread_One(void *argument)
{
int Count=0;
char Buffer[32];
while (1)
  {
  sprintf(Buffer,"Task 0, Count=%d", Count++);
  LCD_PutString(Buffer,0);
  osDelay(1);
  }
}

void Thread_Two(void *argument)
{
int Count=0;
char Buffer[32];
while (1)
  {
  sprintf(Buffer,"Task 1, Count=%d", Count++);
  LCD_PutString(Buffer,1);
  osDelay(1);
  }
}

void Thread_Three(void *argument)
{
int Count=0;
char Buffer[32];
while (1)
  {
  sprintf(Buffer,"Task 2, Count=%d", Count++);
  LCD_PutString(Buffer,2);
  osDelay(1);
  }
}
```

```
void Thread_Four(void *argument)
{
int Count=0;
char Buffer[32];
while (1)
  {
  sprintf(Buffer,"Task 3, Count=%d", Count++);
  LCD_PutString(Buffer,3);
  osDelay(1);
  }
}



void Thread_Five(void *argument)
{
int Count=0;
char Buffer[32];
while (1)
  {
  sprintf(Buffer,"Task 4, Count=%d", Count++);
  LCD_PutString(Buffer,4);
  osDelay(1);
  }
}

void Thread_Six(void *argument)
{
int Count=0;
char Buffer[32];
while (1)
  {
  sprintf(Buffer,"Task 5, Count=%d", Count++);
  LCD_PutString(Buffer,5);
  osDelay(1);
  }
}

void Thread_Seven(void *argument)
{
int Count=0;
char Buffer[32];
while (1)
  {
  sprintf(Buffer,"Task 6, Count=%d", Count++);
  LCD_PutString(Buffer,6);
  osDelay(1);
  }
}

void Thread_Eight(void *argument)
{
int Count=0;
char Buffer[32];
while (1)
  {
  sprintf(Buffer,"Task 7, Count=%d", Count++);
  LCD_PutString(Buffer,7);
  osDelay(1);
  }
}
```

Appendix C

```
   void Thread_One(void *argument)
   {
   char Buffer[32];
   ADC_InitTypeDef sADC;
   ADCx_InitTypeDef sADCx;

   RST_CLK_PCLKcmd (RST_CLK_PCLK_ADC | RST_CLK_PCLK_PORTD, 
ENABLE);
   PORT_InitTypeDef Nastroyka;
   Nastroyka.PORT_Pin = PORT_Pin_7;
   Nastroyka.PORT_OE = PORT_OE_IN;
   Nastroyka.PORT_MODE = PORT_MODE_ANALOG;
   PORT_Init (MDR_PORTD, &Nastroyka);

   ADC_DeInit();
   ADC_StructInit(&sADC);
   sADC.ADC_SynchronousMode= ADC_SyncMode_Independent;
   ADCx_StructInit (&sADCx);
   sADCx.ADC_ClockSource= ADC_CLOCK_SOURCE_CPU;
   sADCx.ADC_SamplingMode= ADC_SAMPLING_MODE_SINGLE_CONV;
   sADCx.ADC_ChannelNumber= ADC_CH_ADC7;
   sADCx.ADC_Channels= 0;
   sADCx.ADC_VRefSource= ADC_VREF_SOURCE_INTERNAL;
   sADCx.ADC_IntVRefSource= ADC_INT_VREF_SOURCE_INEXACT;
   sADCx.ADC_Prescaler= ADC_CLK_div_None;
   MDR_ADC->ADC1_STATUS = (1 << ADC_STATUS_ECOIF_IE_Pos);
   NVIC_SetPriority(ADC_IRQn, 1);
   NVIC_EnableIRQ(ADC_IRQn);

   ADC1_Init (&sADCx);
   ADC1_Cmd (ENABLE);

   while (1)
     {
     ADC1_Start();
      osEventFlagsWait(event, 0x00000001, osFlagsWaitAny, 
osWaitForever);
     sprintf(Buffer,"Task 0, Result=0x%0X", MDR_ADC->ADC1_RESULT & 
ADC_RESULT_Msk);
     osKernelLock();
     LCD_PutString(Buffer,0);
     osKernelUnlock();
     osDelay(1);
     }
   }

   // Interrupt handler
   void ADC_IRQHandler(void)
   {
   osEventFlagsSet(event, 0x00000001U);
   }
```

Методические материалы

CMSIS-RTOS FOR CORTEX-M3 MICROCONTROLLERS

Методические указания к лабораторной работе

Составители: ***Кудрявцев Илья Александрович,***
Корнилин Дмитрий Владимирович,
Мякинин Олег Олегович

Редактор А.В. Ярославцева
Компьютерная верстка А.В. Ярославцевой

Подписано в печать 30.12.2020. Формат 60×84 1/16.
Бумага офсетная. Печ. л. 1,0.
Тираж 25 экз. Заказ . Арт. – 28(Р3М)/2020.

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА» (САМАРСКИЙ УНИВЕРСИТЕТ)
443086, Самара, Московское шоссе, 34.

Издательство Самарского университета.
443086, Самара, Московское шоссе, 34.