7th International Scientific Conference
Technics and Informatics in Education
Faculty of Technical Sciences, Čačak, Serbia, 25-27th May 2018
Session 3: Engineering Education and Practice
UDC: 004.42

Arduino Platform Capabilities in Multitasking Environment

Dragana Mitrović[1*], Siniša Ranđić[1]
[1]University of Kragujevac, Faculty of Technical Sciences Čačak, Serbia
[*] dragana.mitrovic.94@gmail.com

Abstract: *Arduino platforms are one of the most popular bases for the development of embedded devices. The ability to independently design an interface with the environment, gives the Arduino modules characteristics of an "open hardware" device. At the same time, the existence of the Arduino IDE development environment enables easy and stable development of software. The structure of the Arduino program does not provide direct support for the operation of such devices in the multitasking environment. This problem has been partially overcome by the development of libraries, such as, for example, a library that allows the use of the FreeRTOS concept with the Arduino device. The paper presents the elements of realization of multitasking in the Arduino system operations with the support of the FreeRTOS system concept.*

Keywords: Arduino concept; multitasking; real – time operating systems; integrated development environment.

1. INTRODUCTION

In recent years, there have been significant changes in the development of computer hardware. Especially in the so-called embedded systems. In addition to the classic development systems, designers also have devices that can be called open-source hardware systems. Such devices have a predefined processor-memory structure. Regarding the implementation of the input/output subsystem, designers have at their disposal a number of analog and digital inputs that can be used in accordance with the specific requirements of the application. Due to the standardization of computer devices communications with the environment, such devices also have predefined communication interfaces such as UART, SPI, I^{2}C, etc. One of the most famous devices of this type is the Arduino module family [1].

The Arduino concept, as a program for students, was designed at the Interaction Design Institute Ivrea in Ivrea, Italy [2]. The goal was to provide a cost-effective and easy way to design devices that connect with the environment through various sensors and actuators. The Arduino concept is a typical representative of the device with open hardware. Most Arduino modules are based on Atmel 8-bit AVR microcontrollers (ATmega8, ATmega168, ATmega328, ATmega1280, ATmega2560). The microcontrollers that are built into Arduino modules have a boot loader that makes it easy to upload the developed program into a program flash memory. Each Arduino module has a number of digital and analog I/O pins. As a rule, a subset of digital I/O pins can be used to generate PWM (Pulse Width Modulated) signals.

Arduino programs can be written in any programming language for which there are compilers who can generate a machine code for the desired processor. Since Arduino modules are based on Atmel microcontrollers for the program development, the appropriate development environments, AVR Studio and Atmel Studio, can be used. The Arduino project offers an integrated development environment that has a simple mechanism to compile and upload programs into Arduino module. The Arduino IDE development environment is based on the Wiring IDE platform and the programming language Processing [4].

The great success that the Arduino concept has achieved has led to the fact that a large number of development systems, which are being designed and implemented today, has the same interface to the environment as in Arduino. Thus, the I/O interface of the Arduino UNO module has almost become standard.

When defining the Arduino concept and developing the appropriate IDE software, no direct support is provided for working in the multitasking environment. With this in mind, one of the important challenges, in the further development and application of the Arduino concept, was finding the possibility that devices based on the Arduino module work in multitasking mode. The aim of this paper is to point out the possibility of developing programs for Arduino based devices,

which will be able to work in a multitasking environment.

The main efforts are focused on adding libraries with functions designed to support multitasking elements. Special attention is given to the work in the multitasking environment based on the FreeRTOS operating system.

2. ARDUINO PROGRAM CONCEPT

Developing a program for Arduino module has been made very easy, by including necessary libraries and writing functions that use them, maintaining simple and short source code. Because programming in Arduino is based on manipulation of different sensors and actuators, the manufacturers that develop them mostly write libraries. Those who do not have previous experience in programming can just call libraries inside of Arduino code, without needing to understand how they work. On the other hand, more experienced programmers can change Arduino libraries or make new ones, following simple set of rules made by Arduino team.

Arduino libraries and programs are written in C and C++ programming language and the editor that is used for programming Arduino modules is cross-platform Arduino IDE software. Structure of simple program for Arduino is based on two functions, *setup()* and *loop()*. After including libraries and defining variables that will be used in the program, the setup function is created for initializing and setting initial values. This function is only run once, when program is first started. The second function, loop, allows program to change and respond by looping constantly functions described in it. Usually in the end of loop function, *delay()* function is called to postpone the next running of functions inside loop. Figure 1 shows the layout of the Arduino program's initial form (new sketch).

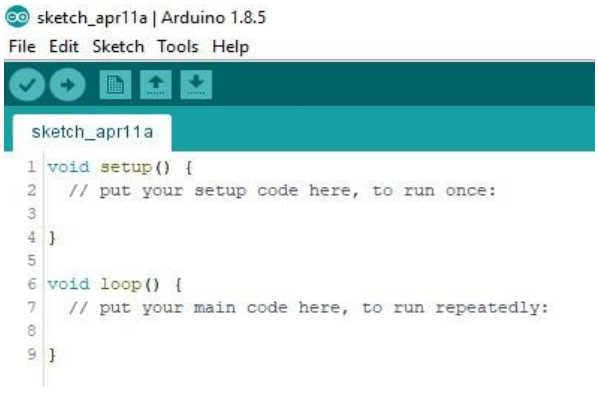

Figure 1. *A view of new Arduino sketch*

This structure of Arduino program allows execution of special tasks only on the principle of batch processing. This means that each task is defined as a separate function, which is then called within the main program loop. A specific function (task) will only be executed again when all the next functions in the loop are executed. The total time of one passage through the program loop depends on the number of functions that are executed and the execution time of each function. The appearance of the corresponding Arduino program is shown in Figure 2.

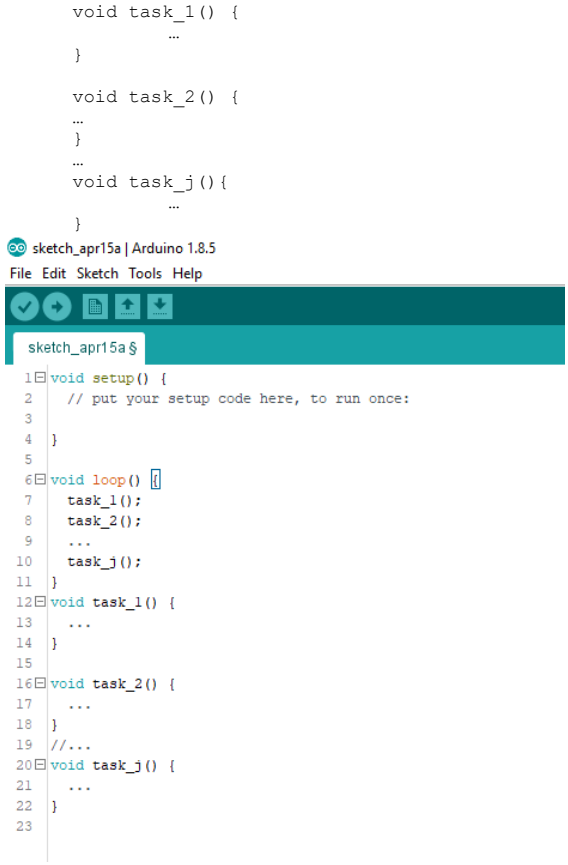

Figure 2. *Arduino program with multifunction environment*

The problem can arise if the number of tasks that are executed is large. In this case, the time interval between the two consecutive activations of the same task can be long. The inability to realize multitasking, in the basic version of the Arduino program, imposed the need for alternative solutions. The most common solution is based on the inclusion of multitasking support by adding appropriate libraries.

Based on this, it can be concluded that the Arduino software environment does not support concurrency, and that the interval in which a particular task can be performed cannot be defined. One of the solutions is the use of the Scheduler Library, which provides support for multiple concurrency loops. Figure 3 shows a part of the code that illustrates multitasking in the Arduino program using the Scheduler library.

Within the code, a special attention has to be given to the command Scheduler.startLoop(Task1) and Scheduler.startLoop(Task2). By default, these commands start tasks Task1 and Task2, whose functions are later defined. A command string, in the basic loop, *loop()* defines tasks that can be tagged with Task0. When this command is

executed, the control is transferred to another task. In the case of using the Scheduler library, the transfer of control to the following tasks can also be accomplished by calling the *yield()* function, as done in the definition of Task2 task.

```
// Include Scheduler Library for managing multiple tasks
void setup() {
  ...
 // Add "Task1" and "Task2" to scheduling.
 // "Task" is always started by default.
  Scheduler.startLoop(Task1);
  Scheduler.startLoop(Task2);
}
 // Task0 ON/OFF LED width 1 second delay
void loop() {
 digitalWrite(LED0, HIGH);
 // When multiple tasks are running 'delay' passes control to
 // other tasks while waiting and guarantees they get executed.
  delay(1000);
  digitalWrite(LED0, LOW);
  delay(1000);
}
// Task1 blink LED with 0.1 second delay.
void Task1() {
  digitalWrite(LED1, HIGH);
  delay(100);
  digitalWrite(LED1, LOW);
  delay(100);
}

// Task2 accept commands from Serial port
// '0' turns OFF LED
// '1' turns ON LED
void loop3() {
  if (Serial.available()) {
    char c = Serial.read();
    if (c == '0') {
      digitalWrite(LED2, LOW);
      Serial.println("LED turned OFF!");
    }
    if (c == '1') {
      digitalWrite(LED2, HIGH);
      Serial.println("LED turned ON!");
    }
  }

 // We must call 'yield' at a regular basis to pass
 // control to other tasks.
  yield();
}
```

Figure 3. *Example of multitasking using the Scheduler library [5]*

In this case, there is no separate task scheduler, but the task management is accomplished by directing the function *yield()* or *delay()*. Also, the considered approach does not provide support for real time operation, periodic activities, and there is no possibility of preemption.

Given the availability and popularity of the Arduino platform and development environment, the designers have also tried to find other solutions for multitasking. As a possible solution, the use of the FreeRTOS operating system was imposed. FreeRTOS [6,7] is an operating system that is designed to support the work of embedded devices in real time. The program was developed with the aim of being small, simple and quick to execute. Therefore, it does not possess some of the advanced features such as device drivers, user accounts and networking, encountered within standard operating systems.

FreeRTOS support for multitasking with Arduino platforms is accomplished by using the appropriate library.

3. ARDUINO AND FREERTOS OPERATING SYSTEM

Operating Systems (OS) are computer programs that support basic computer operations, functions, and provide services to all programs running on it. Most OS allow multiple functions to be executed at once, otherwise known as multitasking. This is just an illusion and in reality, one processor core can run only one task at the time. Behind multitasking process there is a part of operating system called *scheduler*, which rapidly switches through each program and runs only one task at the time. There are different types of OS based on the rules by which schedulers execute tasks, for example, scheduler in Unix OS provides equal time of execution for every task [8].

Real Time Operating System (RTOS) is a type of OS with scheduler that provides *deterministic* pattern for executing tasks [9]. This scheduler is mostly used for embedded systems that often have real time requirements such as responding to events in within defined time window. This deterministic pattern is often achieved by allowing user to assign a priority to each task that has to be executed. One version of RTOS that can be run on microcontrollers is FreeRTOS with real-time scheduler [10].

For realization of the multitasking on Arduino platforms, it is necessary to install the appropriate FreeRTOS library. In this case, the FreeRTOS version was optimized for Arduino AVR devices. This library has compatibility with the Arduino environment with simultaneous access to FreeRTOS functions.

FreeRTOS is compatible with many different architectures and compilers, and each version has few demo applications to help new users. To start project with this library it is necessary to download FreeRTOS .zip file, that contains source code with some demo projects, and extract it. List of all supported demos can be found on www.freertos.org along with the official documentation that contains instruction for running and modifying the FreeRTOS library.

3.1 Installing and using FreeRTOS with Arduino systems

FreeRTOS can be installed on Arduino platform using Arduino IDE Library manager greater than version 1.6.8. After installing it, library is included through *Sketch->Include Library* menu. On Arduino Uno device, FreeRTOS takes about 7340 bytes of its flash memory. After passing these steps, the programmer can compile one of three demos provided with the library. In addition to the simple FreeRTOS functions it is possible to include and use many other functions, like *Semaphores* and similar functions.

To enable multitasking within the Arduino program, it is necessary to include the

Arduino_FreeRTOS library. After that, it is necessary to define tasks that will be competitively carried out. The tasks are created using the xTaskCreate() function [11]. By creating tasks, the scheduler starts automatically. Within the Arduino program, each task is defined by the corresponding function. The principles of multitasking implementation within the Arduino system are shown in Figure 4.

```
#include <Arduino_FreeRTOS.h>
// Define two tasks Task1 & Task2
void TaskBlink( void *pvParameters );
void TaskAnalogRead( void *pvParameters );
void setup() {
// initialize serial communication at 9600 bits per second:
  Serial.begin(9600);
}
// Now set up two tasks to run independently.
  xTaskCreate(
    Task1
    ,  (const portCHAR *)"Task1"   // A name just for humans
    ,  128  // Stack size
    ,  NULL
    ,  2  // Priority,
    ,  NULL );
  xTaskCreate(
     Task2
    ,  (const portCHAR *) "Task2"
    ,  128  // Stack size
    ,  NULL
    ,  1  // Priority
    ,  NULL );
// Now the task scheduler automatically started
}
void loop()
{
  // Empty. Things are done in Tasks.
}
/*---------------------- Tasks ---------------------*/
void Task1(void *pvParameters)  // This is a task.
{
  (void) pvParameters;
  for (;;) // A Task shall never return or exit.
  {
    // Instructions
  }
}
void Task2(void *pvParameters)  // This is a task.
{
  (void) pvParameters;
  for (;;)
  {
    // Instructions
  }
}
```

Figure 4. *The concept of multitasking implementation in the Arduino system using FreeRTOS*

In a multitasking environment, the new task is always ready for execution and is placed in the list of ready tasks. The position of the tasks in the list of prepared tasks is in principle defined by its importance in relation to other tasks. According to task priority, the scheduler selects the next task to execute. During the execution of a task, it can be completed or its assigned time can be expired, and therefore its execution must be interrupted. In this case, the task is returned to the list of ready tasks. If during execution, the task cannot obtain the desired resource, its execution is interrupted, and the task is placed in the list of tasks waiting for resources. On the other hand, if during the execution the task releases the resource, the tasks that are waiting for that resource are transferred to the list of ready tasks.

In the case of FreeRTOS a new tasks are put into the ready state. However, if there are no high priority tasks, the new task will immediately go into the running state. It should be noted that tasks could be created both before and after the scheduler starts. Figure 6 shows the prototype of the xTaskCreate() function.

```
#include "FreeRTOS.h"
#include "task.h"

BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
                        const char * const pcName,
                        unsigned short usStackDepth,
                        void *pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t *pxCreatedTask );
```

Figure 6. *xTaskCreate() function prototype*

The pvTaskCode parameter in the xTaskCreate () function is a pointer to the function that realizes the task. The pcName parameter is a symbolic task name. The stack size is defined by the usStackDepth parameter. In addition, when creating tasks, its priority and input and output parameters are defined.

Transferring control to another task in a ready state can be accomplished by calling the functions vTaskDelay() and taskYIELD(). If the task vTaskDelay() task is called in the running task, the task goes into a blocked state and remains in it a certain number of intervals, which are specified as the function parameter. If this parameter is zero, the task goes into a blocked state, and running becomes a ready task with the same priority. Figure 7 shows the prototype of the vTaskDelay() function.

```
#include "FreeRTOS.h"
#include "task.h"

void vTaskDelay( TickType_t xTicksToDelay );
```

Figure 7. *vTaskDelay() function prototype*

Calling vTaskDelay(0) functions is equivalent to calling the taskYIELD() function. This function cannot be called only within the running task, which means it cannot be called before the scheduler starts. If there is no task with the same priority, the control will return to the task within which the taskYIELD() function is called.

The Watchdog Timer at Arduino microcontroller level generates time intervals from 15ms to 500ms, necessary for multitasking. If the task is completed before the expiration of the allocated quantum of time, the control automatically returns to the Scheduler.

By including FreeRTOS library in the Arduino program, the availability of the program memory for the user functions is reduced. An empty Arduino program (sketch) takes 444 bytes, which is 1% of the program memory. When FeeRTOS library is turned on, the occupancy is 14506 bytes or 44% of the program memory.

4. EXAMPLE OF MULTITASKING ON ARDUINO PLATFORM

An example of three tasks can serve as an illustration of using the FreeRTOS operating

system to work with Arduino platforms. Tasks are intended for:

- To turn On/Off LEDs;
- Reading the analogue value;
- Reading the digital value.

Figure 5 shows the introductory part of the Arduino program, which includes declaring libraries to be used or declaring variables.

```
#include <Arduino_FreeRTOS.h>
int inPin = 7;
int val = 1;

// Define three tasks for Blink & AnalogRead & DigitalRead
void TaskBlink( void *pvParameters );
void TaskAnalogRead( void *pvParameters );
void TaskDigitalRead( void *pvParameters );
// The setup function runs once when you
// press reset or power the board
void setup() {
  // Initialize serial communication at 9600 bits per second:
  Serial.begin(9600);
  pinMode(inPin, INPUT);

  // Now set up three tasks to run independently.
  xTaskCreate(
    TaskBlink
    ,  (const portCHAR *)"Blink"   // A name just for humans
    ,  128  // This stack size can be checked & adjusted
            // by reading the Stack Highwater
    ,  NULL
    ,  3  // Priority, with 3 (configMAX_PRIORITIES - 1)
       // being the highest, and 0 being the lowest.
    ,  NULL );

  xTaskCreate(
    TaskAnalogRead
    ,  (const portCHAR *) "AnalogRead"
    ,  128  // Stack size
    ,  NULL
    ,  2  // Priority
    ,  NULL );

   xTaskCreate(
    TaskDigitalRead
    ,  (const portCHAR *) "DigitalRead"
    ,  128  // Stack size
    ,  NULL
    ,  1  // Priority
    ,  NULL );
  // Now the task scheduler, which takes over control of
  // scheduling individual tasks, is automatically started.
}
```

Figure 5. *Introductory part of Arduino program*

In the declaration part of the program, the inclusion of the library required for enabling multitasking (Arduino_FreeRTOS) was performed. Also, three tasks have been declared – TaskBlink, TaskAnalogRead, and TaskDigitalRead. Within the setup of the section besides the standard definition of serial communication and digital contact as an input, tasks were created as independent entities. For each of the tasks created, it is defined:

- Symbolic task name (Blink, AnalogRead, DigitalRead);
- The size of the stack assigned to the task;
- Task priority.

By executing a setup part of the program, the task scheduler starts automatically. The main program loop is empty, because all jobs are executed within the tasks. At the end of the program, the program codes of the functions that implement the tasks are given. In Figure 6, the Blink and DigitalRead Task codes are given.

```
void TaskBlink(void *pvParameters)  // This is a task.
{
  (void) pvParameters;

  // Initialize digital LED_BUILTIN on pin 13 as an output.
  pinMode(LED_BUILTIN, OUTPUT);

  for (;;) // A Task shall never return or exit.
  {
    Serial.println("----------Task 1-1");
    digitalWrite(LED_BUILTIN, HIGH);
      // Turn the LED on (HIGH is the voltage level)
    vTaskDelay( 1000 / portTICK_PERIOD_MS );
      // Wait for one second
    Serial.println("----------Task 1-2");
    digitalWrite(LED_BUILTIN, LOW);
      // Turn the LED off by making the voltage LOW
    vTaskDelay( 1000 / portTICK_PERIOD_MS ); // wait for one second
  }
}

void TaskDigitalRead(void *pvParameters)  // This is a task.
{
  (void) pvParameters;
  for (;;)
  {
    //Serial.println("----------Task 3");
      // Read the input on digital pin 7:
     val = digitalRead(inPin);
      // Print out the value you read:
    Serial.print("Digital value: ");
    Serial.println(val);
    vTaskDelay(40);
      // One tick delay (1ms) in between reads for stability
  }
}
```

Figure 6. *The program code of the DigitalRead task*

Figure 7 shows the appearance of the SerialMonitor, which reflects sequencing of tasks, which are defined within the Arduino program under consideration.

```
----------Task 1-1
Analog value: 572
Digital value: 0
Analog value: 573
Digital value: 0
----------Task 1-2
Analog value: 572
Digital value: 0
Analog value: 572
Digital value: 0
----------Task 1-1
```

Figure 7. *Display on Serial Monitor*

Task 1 is executed from two steps. The first step is to turn on the LED, after which the task is blocked, and the control transmits to Task 2, and then to Task 3. Blocked Task 1 takes a long enough time to repeat the Task 2 and Task 3 execution cycle. Task 1 takes a long time to repeat the execution. Task 2 and Task 3 cycle activate Task 1, i.e. its second step in which the LED turns off and goes back to the blocked state. This creates the conditions to reactivate Task 1 and Task 2, which are executed twice before the control takes over Task 1 again.

5. CONCLUSION

The paper presents an approach to overcoming the problem of the lack of adequate multitasking support for the Arduino program environment.

Similar problems are not rarity in the field of computer technology, so they can serve as an example of what everything should be considered when designing new systems. The popularity and the breadth of the application of the Arduino concept further influenced the significance of this problem.

Indication of this and similar problems, as well as the presentation of the ways of their overcoming, creates opportunities for the development of appropriate educational content, especially at the master studies level. This was precisely the motive for students of master studies in the field of computer engineering in the Intelligent Sensors course to deal with the problems of multitasking in the Arduino system. The goal was to get out of the standard framework for acquiring knowledge on the principle of ex cathedra and to move on to learning through practice. The results achieved during the research, familiarization with the problem of multitasking in the Arduino system and ways of solving them, point to the importance and possibilities of this approach to acquiring knowledge.

Experience in implementing multitasking on Arduino systems can serve in the implementation of a similar work environment on embedded systems based on microcontrollers. This has a special significance when implementing devices belonging to IoT (Internet of Things), in which the application of multitasking can be very important [12].

ACKNOWLEDGEMENTS

The paper presents the results of the research within the Intelligent Sensors course at the Master studies in the field of Computer Engineering at the University of Kragujevac, Faculty of Technical Sciences in Čačak. Material support to the research was realized through the TR32043 project funded by the Ministry of Education, Science, and Technological Development of the Republic of Serbia.

REFERENCES

[1] Banzi, M., Shiloh, M., "The Open Source Electronics Prototyping Platform", *Maker Media, Inc*., 2015
[2] Kushner, D., "The Making of Arduino", *IEEE Spectrum,* October 2011
[3] Torvalds, M., "Arduino programming: Step – by – step Guide to Mastering Arduino Hardware and Software", *Amazon Digital Services LLC*, 2017
[4] Shiffman, D., "Learning Processing: A Beginner's Guide to Programming Images, Animation and Interaction", *1st Edition, Morgan Kaufmann*, 2008
[5] https://playground.arduino.cc/Code/Scheduler
[6] Cooling, J., "Real – time Operating Systems", *Book 1 – The Theory, Independently published*, 2017
[7] Cooling, J., "Real – time Operating Systems", Book 2 – Practice: Using STM Cube, FreeRTOS and the STM32 Discovery Board (The engineering of real – time embedded systems), *Independently published*, 2017
[8] Gulati, M., Gulati, M., "UNIX", Siliconmedia, *Amazon Digital Services LLC*, 2016
[9] Wang, K. C., "Embedded and Real – Time Operating Systems", *1st Edition, Springer*, 2017
[10] Barry, R., "Mastering the FreeRTOS Real Time Kernel, A Hands-On Tutorial Guide", Pre-release 161204 Edition, *Real Time Engineers*, 2016
[11] "FreeRTOS Reference Manual: API Functions and Configuration Options", Version 10.0.0 Issue 1*, Amazon Web Services*, 2017
[12] Serpanos, D., Wolf, M., "Internet-of-Things Systems: Architectures, Algorithms, Methodologies", *1st Edition, Springer*, 2017