Multitasking and Real-Time Arduino System

Melvin Mancini, *Student of Computer and Automation Engineering from Marche Polytechnic University*,
Email: melvin.mancini@gmail.com

Abstract— **This article presents the main differences between a Real-Time/Multitasking and Non-Real-Time operating system running on a widespread microcontroller, Arduino. A simple hardware and software system has been implemented to evaluate the main differences in efficiency between two running OSs on Arduino: ERIKA (Enterprise Open-Source Real-time Kernel) and Arduino's default OS Singletask. The goal is to show the performance of the execution of processes when they are managed by OS Real time and not. Additionally, we will focus on the importance of systems that must handle critical tasks which need to be run with predefined deadlines.**

I. Introduction

In recent years, Arduino is the most widely used platform for rapid prototyping. It consists of a programmable electronic board and an integrated development tool (IDE) running on personal computer. Arduino boards are able to read inputs such as light on a sensor, a finger on a button and turn it into an output, for example activating a motor, turning on an LED. Arduino provides a framework that allows the user to develop a fully-working application without knowing any details of the underlying hardware, in contrast with most of embedded firmware development environments. The language used to implement Arduino applications is a language similar to C++, so the programming knowledge required is basic. In addiction, there is a huge number of free third party libraries and code examples that allows a quick interaction with external devices. In spite of its simplicity and effectiveness, Arduino does not support concurrency and a program is limited to a single block of instructions cyclically repeated. The instructions written in a particular section of the program are cyclically repeated sequentially. This means that it is not possible run processes in parallel (Multitasking) or activities in Real-Time mode. In order to run periodic processes with deadlines, it is necessary to implement a sequential program that, at each cycle, controls what activity to be performed.
To overcome the limitation explained above, a special version of the Arduino framework has been used, called **ARTe (Arduino Real-Time Extension)**. *ARTe*, which introduces a support for implementing concurrent real-time applications in the standard Arduino framework with a minimal impact on the original programming model [5]. In addition to the classical Arduino programming model, consisting in a single main loop containing the code to be executed [1] [2], ARTe allows the user to specify a number of different loops, each to be executed with a given desired period [5]. An experiments is presented to evaluate the impact of the implemented solution in terms of both memory footprint, runtime overhead and CPU load on a concrete case-study.

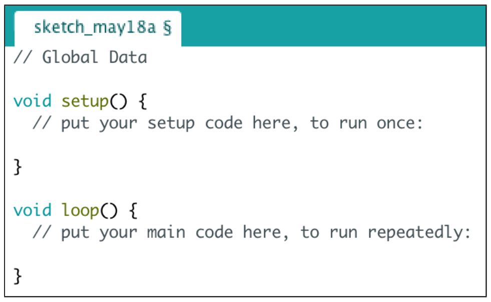
Paper structure. The remainder of this paper is organized as follows. Section II discusses the background including the **Arduino Framework** and a brief description of the RTOS (Real-Time Operating System) **ERIKA Enterprice**. Section III describes the design, system architecture and the internal structure of **ARTe**. Section IV presents the work done, the comparison between the various approaches analyzed and the performance evaluation obtained. Finally, Section V states about conclusions and future work.

II. Background

This section presents the basic features of Arduino and a general description of ARTe's structure and its functioning.

A. Arduino Framework

The **Arduino framework** provides a set of electronic boards (with one or more microcontrollers) and an integrated Development environment (IDE) used to develop the user Application and program the board [1]. There are several Arduino electronic boards with different hardware features and many I/O devices to integrate. For this instance, it was used the most popular Arduino board called **Arduino UNO**. It is equipped with an ATmega328P microcontroller running at 16 MHz, which offers 14 digital I/O pins and 6 analog inputs [2]. In order to develop the applications that will be loaded into the boards, Arduino provides a simple and intuitive software development tool. The **Arduino Integrated Development Environment**, or **Arduino Software (IDE)**, contains a text editor for writing code, a message area, a text console, a toolbar with buttons for common functions and a series of menus [1] [2]. The programming language to use is quite similar to C/C++ and the structure of the code is simple and straightforward.
The programming model structure consists of two basic functions, as reported in Figure 1. Function *setup()* contains the code that must be executed when Arduino is powered on, while *loop()* contains the code that must be cyclically repeated forever. These two functions, together with other *global data structures*, are part of the main file, denoted as *sketch*, where Arduino applications are developed. There are many standard libraries available that help facilitate development for users. Additionally, there are many libraries written by other developers, which allow you to implement specific features. As stated in the introduction, the Arduino framework does not provide a Multitasking and Real-Time programming structure. Once the *sketch* is loaded into the board, the *loop()* block can be considered the only running process [3]. The instructions inside the *loop()* are executed sequentially and cyclically.



```
// Global Data

void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

Fig. 1. Programming model structure Arduino

B. ERIKA Enterprise

ERIKA Enterprise (ERIKA for short) is an open-source real-time kernel that allows achieving high predictable timing behavior with a very small run-time overhead and memory footprint (in the order of one kilobyte). ERIKA is an OSEK/VDX [4] certified RTOS that uses innovative programming features to support time sensitive applications on a wide range of microcontrollers and multi-core platforms. In addition to the OSEK/VDX standard features, ERIKA provides other custom conformance classes, such as fixedpriority scheduling with preemption thresholds, deadline scheduling, through the Earliest Deadline First (EDF) algorithm and the Stack Resource Policy (SRP) [11], resource reservations (FRSH), and hierarchical scheduling (HR) [8]. ERIKA supports both periodic and aperiodic tasks under fixed and dynamic priorities and includes mutex primitives for guaranteeing bounded blocking on critical sections.
ERIKA also provides two types of interrupt handling mechanisms: a fast one (also referred to as Type 1) for short and urgent I/O operations, returning to the application without calling the scheduler, and a safe one (also referred to as Type 2) that calls the scheduler at the end of the service routine, meant to be used for the interaction with kernel objects (e.g., for activating a task).
As specified by the OSEK/VDX standard [4], in ERIKA all the RTOS objects like tasks, alarms and semaphores are static; that is, all the RTOS configurations are predefined at compile time and cannot be changed at run-time. The choice of using a static approach is crucial for containing both footprint and run-time overhead, obtaining a tailored RTOS image that is optimized for a specific application-dependent kernel configuration [5].
In ERIKA, the objects composing a particular application are specified in OIL (OSEK Implementation Language) and stored in proper configuration files. The ERIKA development environment also includes RT-Druid, which is a tool in charge of processing the OIL configuration to generate the specific ERIKA code for the requested kernel configuration.

III. The ARTe Architecture

This section presents the features and internal structure of *ARTe*, describing the flow of system operation.
ARTe is a framework that extends the Arduino framework that provides Multitasking and Real-time support. *ARTe* maintains the simplicity and linearity of the Arduino development framework (same syntax and similar programming structure), adding the functionality of multitasking and real-time execution.

A. ARTe design

There are several Real-Time and Multiprocessing operating systems that can be run on Arduino boards. There are many difficulties encountered in loading an OS Real-Time into a microcontroller. Often, it is necessary modify available hardware components or correct some low-level software instructions. Real-time OS applications are difficult to implement and advanced programming knowledge is required. The purpose of *ARTe* is to easily implement applications running in Real-Time mode on Arduino boards. The user does not have to worry about physically loading a real-time OS into the Arduino microcontroller. ***ARTe* introduces multitasking support and allows running multiple concurrent tasks in addition to the single execution cycle provided by the standard Arduino framework. *ARTe* shares the same Arduino philosophy, maintaining the same code syntax and a similar programming structure pattern**. In general, by implementing programs written in Arduino code, it is possible to have scheduled tasks according to a Real-Time algorithm. Tasks that need to be executed periodically and with deadlines are executed on Arduino boards, but managed according to the OS ERIKA Enterprise. The Arduino sketch is converted to a program that can be run on the ERIKA operating system. If the code compilation process (Arduino side and Erika side) is successful, it is possible load the source on the Arduino board and get a Multitasking and Real-Time hardware and software system [5]. ARTe has been conceived according to the following design objectives:

- *Simplicity*: *ARTe* allows to integrate a multitasking and Real-Time support in Arduino, while maintaining the simplicity of implementation (same code syntax and similar programming structure) offered by the Arduino framework [1] [3] [5].
- *Integration with standard Arduino Libraries*: The large number of libraries implemented for Arduino is the real strength of its diffusion [1]. Libraries facilitate user development, helping them in programming [2]. To this purpose, *ARTe* has been conceived to enable the use of all existing Arduino libraries inside a multiprogrammed application.
- *Real-Time multitasking support*: The Arduino platform was born with the purpose of providing the ability to easily build systems that interact with the outside environment through sensors, actuators and I/O devises. For this reason, any delay introduced in the computational activities may affect the overall system performance. Bounding the execution delays in all the concurrent activities programmed by the user is therefore crucial for ensuring a desired system performance.
- *Efficiency*: To preserve the performance of the Arduino computing platforms, *ARTe* has been designed to have a minimal impact on resource usage, in terms of both footprint and run-time overhead [5].

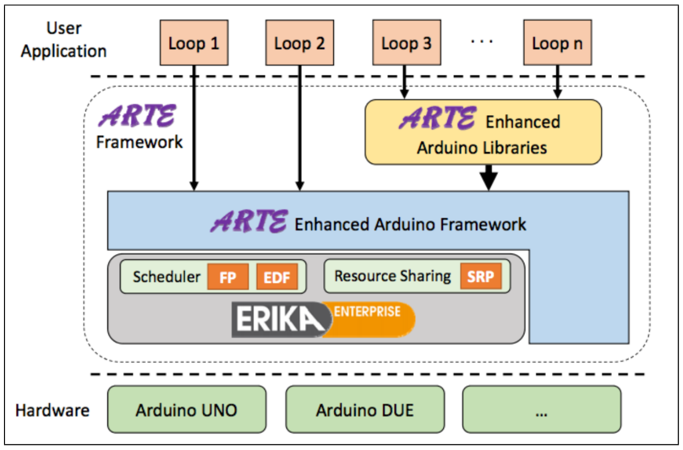
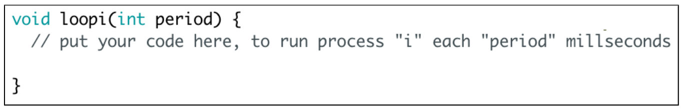


Fig. 2. General ARTe architecture



```
void loopi(int period) {
  // put your code here, to run process "i" each "period" millseconds

}
```

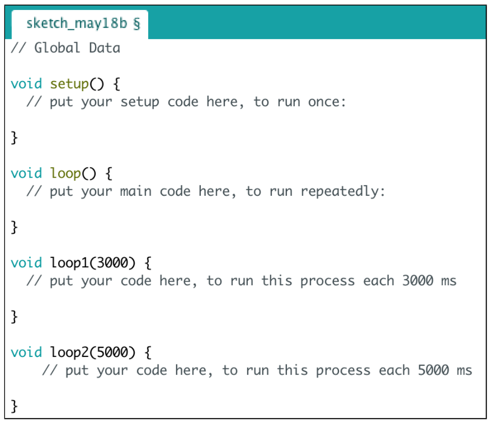
Fig. 3. How to write the loops

B. System description

The general architecture of the *ARTe framework* is illustrated in Figure 2. *ARTe framework* is inserted between the user space and the Arduino hardware board. Internally, in addition to the Arduino framework, it contains ERIKA's source code. It is important to note how *ARTe* maintains the main features of the framework Arduino, offering the ability to use all available libraries [5]. In addition to the single loop present in the standard Arduino approach, the user can specify **n concurrent loops**. Each loop identifies a process that can handle an Arduino activity such as reading a sensor value or controlling an actuator. The code inside each loop is converted to a task that can be executed on the RTOS ERIKA. At the bottom of the *ARTe* architecture there are the Arduino hardware platforms. Today *ARTe* supports the most popular platforms [5]: Arduino UNO and Arduino DUE.

C. ARTe programming model structure

As explained in the introduction, the *ARTe* programming model is as similar as possible to the original Arduino programming model. Each periodic loop defined by the user is specified as shown in 3, where i = 1, 2, 3, . . . and period represents the time interval (in milliseconds) with which the loop is executed. As in the original Arduino programming model, the *setup()* function is also available under *ARTe* with the same syntax and semantics. Similarly, the original *loop()* function can also be used under *ARTe*, offering the programmer the possibility to execute background activities when no other pending loops are running. A sketch created with the *ARTe framework* is structured as in Figure 4.



```
// Global Data

void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}

void loop1(3000) {
  // put your code here, to run this process each 3000 ms

}

void loop2(5000) {
    // put your code here, to run this process each 5000 ms

}
```

Fig. 4. Example of sketch with ARTe framework

D. ARTe build process

ARTe build process is illustrated in Figure 5. It explains how the user code (i.e. the *sketch*) is processed to obtain a binary executable file which constitutes a multitasking ERIKA application. In addition to the standard compilation phase of the Arduino framework (which is implemented inside the Arduino IDE), *ARTe* extends it with two processing phases: **ARTe pre-processing**, which processes the sketch before the original Arduino processing and the **ARTe post-processing**, invoked after the original Arduino processing. Subsequently, the compilation phases of *ARTe* will be described and analyzed. At the end of the build process, Multitasking and Real-time applications can be obtained that can be run on Arduino devices [5].

*1) **ARTe pre-processing**:* During this phase, the sketch is processed to extract the structure of the application and automatically generate the ERIKA configuration supporting the execution of the user application. For each identified loop, an ERIKA task configuration is generated in an OIL file and then associated to the code inside the loop. The remaining part of the ERIKA configuration consists in an OIL section that specifies the underlying hardware platform, which is selected from a set of predefined OIL templates [5].

*2) **Arduino processing**:* This phase is the standard stage of the framework [1] [2] [3]. The process is required to create compiler-compatible code. In particular, the original sketch (in .pde or .ino formats) is converted to a standard .cpp file (i.e., C++ code). Please refer to the official Arduino documentation for additional details on this phase [1][2][3].

*3) **ARTe post-processing**:* This phase is responsible for transforming the sketch into an ERIKA application and modify the .cpp file produced in the previous step to make it compiler-compatible. Specifically, each ARTE loop declaration is transformed into an OSEK compliant task declaration, in the form TASK(loopi). Also, since Arduino sketches are written in C++, while Erika is written in C, the ERIKA code has to be wrapped into an extern ”C” declaration to avoid errors when the code is linked together. At this point, the sketch is ready to be compiled, but it still requires additions to make it fully functional [5].

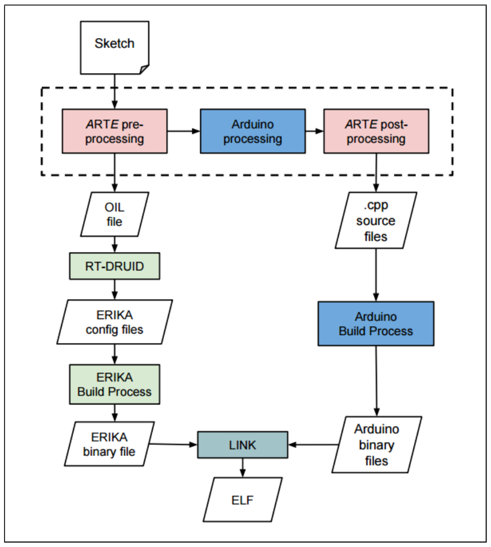


Fig. 5. ARTe build process

4) ***Linking****:* As described in the pre-processing step, the *ARTe pre-processing* phase produces as output the ERIKA configuration consisting in an OIL file. This file is given as input to the RT-DRUID tool, which generates the specific files of ERIKA describing its configuration. At this time, the ERIKA build process is executed to obtain the RTOS binary. On the other hand, the user code is built from the standard Arduino generation process, improved to have ERIKA C header visibility to get the files of the user application. Finally, the LINK phase puts together the ERIKA binaries with the object files deriving from the Arduino build process, generating the ELF binary file ready to be loaded into the microcontroller [5].

IV. Related work

This section presents the architecture of the implemented system and a description of the application development. A system performance comparison will be performed when its main assets are managed by different methodologies. An evaluation of the efficiency of applications implemented with different approaches will be presented.

A. The purpose

The work done has the purpose of presenting a real case study and showing the hardware and software system implemented. The built system simulates a technological structure consisting of several sensors that are usually installed on a Formula 1 car. The aim of the work is not to implement a real structure that needs to be installed. Therefore, the instrumentation and sensors used are not characterized by high precision and accuracy proprieties. The goal is to create a system that manages several activities and evaluate its performance when the principal tasks are executed according to two different approaches:

- *Singletasking and Non-Real-Time* using the standard Arduino system
- *Multitasking and Real-Time* using *RTOS ERIKA* implemented by *ARTe framework* for Arduino platforms.

A comparison will be made between software implementation using the *Arduino standard framework* and the use of the *ARTe framework*. The purpose is to compare and evaluate system performance when its processes are executed according to two different software methodologies.

B. The scenario

As described above, the scenario is to simulate a system that is usually installed on a Formula 1 car. In particular, the system must handle 3 major tasks that need to be executed in real time. Each one concerns a critical activity that a system must be able to handle. Individual activities have a different level of importance. In order of priority, the individual tasks are listed below:

1) The most important task is to give an alarm when an obstacle is detected below 10 cm. This is to alert the pilot of the risk of a dangerous contact.
2) The second task is to flash a LED. According to the official regulation, each car must have a flashing component that will turn on and off at a periodicity.
3) Finally, a car temperature acquisition system that has to send an alarm signal when a centigrade grade is exceeded. This is to avoid overheating of the engine or fuel and avoid possible fires.

C. System Architecture

The hardware architecture of the system is illustrated in Figure 6. The electronic board used is an **ARDUINO UNO**. The structure consists of 3 **LED** diodes with their resistors, **LM35 temperature sensor** and **HR-SC04 ultrasonic proximity sensor**. Each component physically performs the critical tasks that the system must handle. Depending on their purpose, the sensors have input or output capabilities. The signals captured by the devices are forwarded to Arduino, which processes them through the implemented software application. The software structure created consists of 3 processes which, according to their priority, are executed periodically with their deadlines. Each task takes care of a single critical activity. Next, the individual processes and the operation of the various sensors will be described.

1) First task: The most important task is the one that manages the obstacle detection activity. The implemented software procedure (Figure 7) allows to capture the input values returned by the *HC-SR04 sensor* and then it converts them to a distance. The principle of operation of the proximity sensor is based on the flying time of an acoustic signal. The device operates in transmission mode (TX) and in reception mode (RX) [9]. During transmission, the component sends a sound signal in one direction and then switches to RX mode waiting for the return signal. The input value acquired by Arduino is the time needed by the sound wave to travel and to return if it rebounds on an obstacle. The bottom of the device is about 36

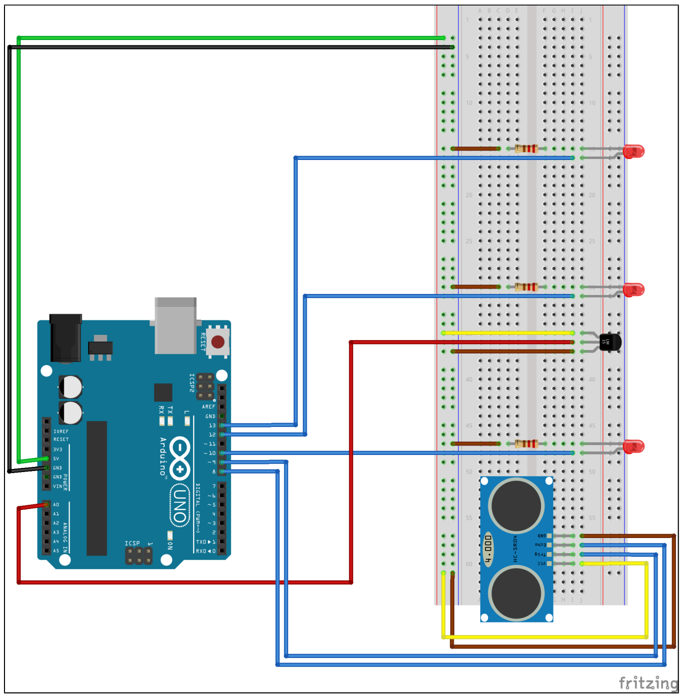


Fig. 6. System hardware architecture

centimeters [9]. Distance control is the most critical activity and therefore the highest priority is assigned to the process. High priority equals a short run time execution (3 seconds) or higher running frequency.

2) Second task: The second activity is to *flash* the *LED* for a period of time. A process executed every 7 seconds manages the LED. The software procedure (Figure 8) cyclically executes a series of instructions that turn on the LED for 500 milliseconds and then turn it off for another 500 milliseconds.

3) Third task: The last activity is temperature measurement. The *LM35 sensor* returns an analog value of the measured temperature. The software reads the analog value and performs sampling on 1024 values. This procedure (Figure 9) allows you to convert the input to the respective measurement in degrees Celsius. For more precision, 3 subsequent measurements (each 500 milliseconds at a time) are performed and then the arithmetic mean is calculated. From the datasheet [10] the minimum measurement is -50C while the maximum is 150C. These values depend on the applied voltage and operating conditions. The activity for temperature detection is less critical than the other. For this reason, the task is assigned a longer run time (11 seconds) and then it will be executed with less frequency.

D. Software Applications

This section will introduce applications developed according to different methodologies. Both applications perform the tasks described in the previous section, but each manages those processes in a different way. The code of each application is too extensive to be submitted and for this reason the main parts of the code will be illustrated.

```
void loop1(3000) {
  // Send a HIGH pulse to the trigger pin
  digitalWrite(trigger, HIGH);
  // I leave it to the HIGH value for 10 microseconds
  delayMicroseconds(10);
  // I carry it back to the LOW state
  digitalWrite(trigger, LOW);

  // I get the number of microseconds for which the echo PIN is left in
  // HIGH state
  duration = pulseIn(echo, HIGH);

  /* Sound velocity is 340 meters per second, or 29 microseconds per cent.
    Our impulse travels back and forth, so to calculate the distance
   Between the sensor and our obstacle we need to do: */
  distance = duration / 29 / 2;

  // Turn on the led if there is an obstacle at a distance of less than 10 cm
  if (distance < 10) {
    digitalWrite(led_HC, HIGH);
  }
  else {
    digitalWrite(led_HC, LOW);
  }
  // Print on the serial buffer
  Serial.print("duration : ");
  Serial.print(duration);
  Serial.print(" - distance : ");
  Serial.println(distance);
}
```

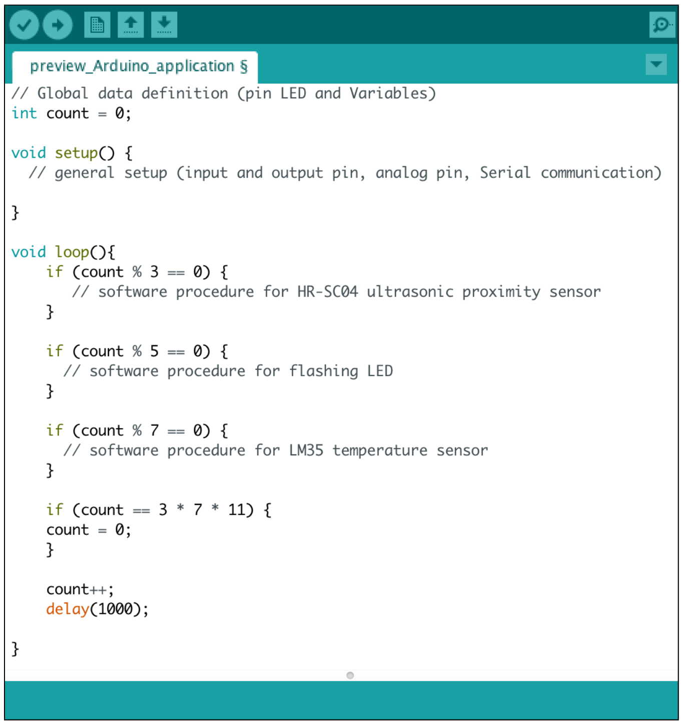
Fig. 7. Software procedure for HR-SC04 ultrasonic proximity sensor

```
void loop2(7000) {
  int i;
  Serial.println("Pulsed LED");
  // Code for the flashing of the led
  for (i = 0; i < 5; i++)
  {
    digitalWrite(led_pulsed, HIGH);
    delay(500);
    digitalWrite(led_pulsed, LOW);
    delay(500);
  }
}
```

Fig. 8. Software procedure for flashing LED

```
void loop3(11000) {
  float sum_temp = 0;
  float average_temp = 0;
  /* Acquire 3 values from the sensor every 500 milliseconds.
     Through an appropriate conversion I get the measure in degrees Celsius.
     Calculate the average of subsequent measurements.*/
  for (int i = 0; i < 3; i++){
    delay(500);
    temp_C = (1.1 * analogRead(LM35_analogPin) * 100.0) / 1024;
    sum_temp += temp_C;
  }
  average_temp = sum_temp / 3;
  // If the temperature is greater than 23 degrees, turn on the led
  if (average_temp > 23) {
    digitalWrite(led_temperature, HIGH);
  }
  else {
    digitalWrite(led_temperature, LOW);
  }
  Serial.print("Temperature : ");
  Serial.println(average_temp);
}
```

Fig. 9. Software procedure for LM35 temperature sensor

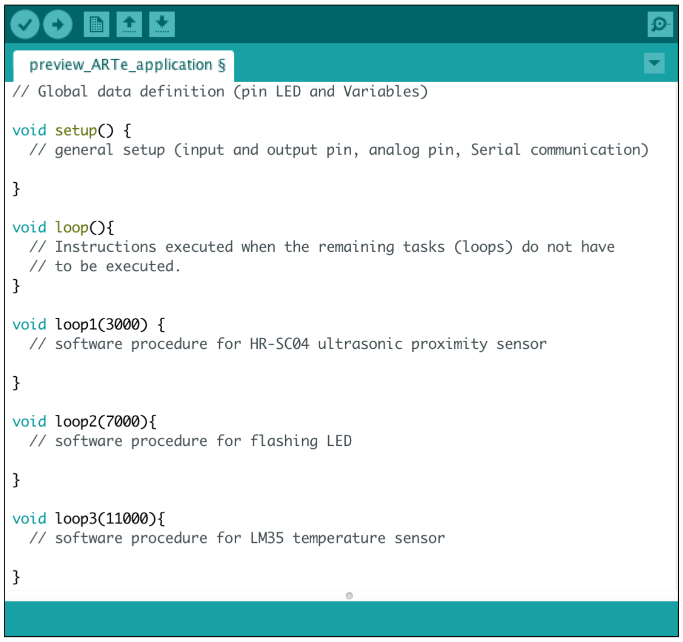


```
// Global data definition (pin LED and Variables)
int count = 0;

void setup() {
  // general setup (input and output pin, analog pin, Serial communication)

}

void loop(){
    if (count % 3 == 0) {
       // software procedure for HR-SC04 ultrasonic proximity sensor
    }

    if (count % 5 == 0) {
      // software procedure for flashing LED
    }

    if (count % 7 == 0) {
      // software procedure for LM35 temperature sensor
    }

    if (count == 3 * 7 * 11) {
    count = 0;
    }

    count++;
    delay(1000);

}
```

Fig. 10. Application developed with the standard Arduino framework according to a Singletask and Non-Real-Time methodology

1) Application developed with Arduino Framework: Figure 10 shows the application code developed using the *standard Arduino frawmework*. In this case, a Singletask and Non-Real-Time methodology was used. The heart of the program is within the *loop()* function. As explained in the previous sections, the Arduino standard framework provides a programming model structure. The instructions inside the *loop()* block are run cyclically and the task to be executed is selected according to the value of a counter variable (*count*). Each software procedure that implemented the critical activities are executed based on the value of the *count* variable. If *count* is divisible by 3 then the task code for the obstacle detection (via the ultrasonic sensor HC-SR04) is executed. If *count* is divisible by 7, the process is executed to blink the led. Finally, if *count* is divisible by 11 then the temperature detection task (via the LM35 sensor) is executed. When *count* is a multiple of 231 (3*7*11) then it is reset. The sketch syntax is controlled through the Arduino compilation process and then it is loaded into the *ARDUINO UNO* board.

2) Application developed with ARTe framework: Figure 11 illustrates the source code of the application developed through the *ARTe framework*. The methodology used is Multitasking and Real-time. The programming model structure of *ARTe framework* was described in the previous sections. Critical activities are structured in processes. Each tasks is implemented as a software procedure. The code that allows you to interact with the various devices is written inside each *loopi(int period)* block. Loops are organized in an orderly manner according to their priority. In summary:



```
// Global data definition (pin LED and Variables)

void setup() {
  // general setup (input and output pin, analog pin, Serial communication)

}

void loop(){
  // Instructions executed when the remaining tasks (loops) do not have
  // to be executed.
}

void loop1(3000) {
  // software procedure for HR-SC04 ultrasonic proximity sensor

}

void loop2(7000){
  // software procedure for flashing LED

}

void loop3(11000){
  // software procedure for LM35 temperature sensor

}
```

Fig. 11. Application developed with ARTe framework according to Multitasking and Real-Time approach

- *loop1(3000)* is the task to detect obstacles and it runs every 3 seconds (3000 milliseconds),
- *loop2(7000)* is the process for flashing the LED every 7 seconds,
- *loop3(11000)* is the software procedure for managing the temperature system and it has a deadline of 11 seconds.

The sketch thus implemented is first compiled (following the steps described in the ARTe build process) and subsequently uploaded to the *ARDUINO UNO* board.

E. Comparison and evaluations

ARTe framework is designed as multitasking support that allows you to run multiple concurrent tasks. *ARTe* shares the simplicity and linearity of the *Arduino standard framework*, keeping the same code syntax and a similar programming model structure. The difference between the *Arduino standard framework* and the *ARTe framework* is that in the first case, all processes are included in the *loop()*, whereas in the second case *n activities* can be organized in *n concurrent tasks* executed with periodicity. The implementation of each process is enclosed in its code section and it is possible to specify the execution frequency (expressed in time period). Indeed, the parameter indicated in the brackets of the loop is the period (in milliseconds) at which it has to be executed.
The main difference between the applications is the compilation and execution mode. The sketch implemented through *Arduino framework* is compiled with the standard Arduino compiler. The selection of the process to be performed is chosen according to the value of the *count* variable and according to its divisibility. The single Arduino *loop()* contains a delay instruction that is responsible to define the time granularity of the *loop*. Although such a system can still be easily

handled with the original *Arduino framework*, the situation can get worse with more complex applications, requiring a much higher programming effort to emulate a multithread behavior [5]. The code implemented through *ARTe framework* is compiled according to the process described in the previous sections (section *ARTe build process*).The sketch is compiled according to the standard Arduino process and through the RTOS ERIKA compiler. The result is a multitasking and Real-Time application and its processes are executed through the *ARDUINO UNO* board but managed according to a real-time execution algorithm. Another great advantage of the *ARTe* multithread support is that loops are preemptive, with a context switch time resulted to be lower than 10 microseconds on the Arduino UNO platform [5]. This feature is really important when the application includes small loops running with short periods together with time consuming loops with a long period. This situation, cannot be easily implemented using the classical Arduino programming model. The performance of the application developed by *ARTe* is better than the implementation of the software through the *standard Arduino framework*. The Real-Time and Multithreading application is more responsive and have reduced response times. Tasks are executed within their deadlines and critical critical processes can override tasks that have less urgency to run.
A simple example demonstrates the efficiency and effectiveness of the Multitasking and Real-Time application. Consider the case where *loop2(7000)* is running and then the LED, which simulates the car light, is blinking. The system behaves differently when it introduces an obstacle of less than 10 centimeters from the proximity sensor. If the singletask application is running on the *ARDUINO UNO* board, the warning LED of *HC-SR04 sensor* will be activated only after the process that manages the LED blinks is completed. If the application implemented with *ARTe* is running, the alarm LED is activated during the LED flashing. In single *loop()*, process priority is not respected and the system contains significant delays. Thanks to *ARTe* the *loop1(3000)*, which identifies the highest priority process, is executed while running *loop2(7000)*. This means that **the system respects execution order based on process priorities**. Additionally, **multithreading support introduced by *ARTe* allows concurrent execution of tasks**.

V. Conclusion

This paper presents the architecture and main features of *ARTe*, the Arduino framework extension that introduces multitasking support and execution of processes in Real-Time mode. In addition, a system comparison was made between different software methodologies. System performance was performed by analyzing the execution of different applications. Comparison confirmed the effectiveness of the system when its processes are running in Multitasking and Real-Time modes. The system generates excessive delays and poor responsiveness when its activities are handled according to the *standard Arduino framework*. The code written through *ARTe framework* allows you to develop efficient software applications. All critical activities are managed and the corresponding tasks are performed effectively and concurrently. The system is very responsive and has short response times.
ARTe is a development tool, alternative to the standard Arduino framework, which allows you to develop Multithreading and Real-Time software applications for Arduino hardware platforms. The main advantage that *ARTe* introduces is **Multitasking and Real-Time support** that the *standard Arduino IDE* does not have. ***ARTe* shares the same Arduino philosophy, maintaining the same code syntax and a similar programming model structure**. Its use is particularly useful when developing applications for systems whose activities are critical and their implementation must be guaranteed over time.

References

[1] *Arduino Introduction*: Official Arduino Site where there are all information about software and hardware platform. https://www.arduino.cc/en/Guide/Introduction

[2] *Arduino Playground*: a wiki where all the users of Arduino can contribute and benefit from their collective research. http://playground.arduino.cc/Main/LM35HigherResolution

[3] Arduino Scheduler library. https://www.arduino.cc/en/Reference/Scheduler

[4] *OSEK*: OSEK/VDX Operating System Specification 2.2.1. OSEK Group. http://www.osek-vdx.org, 2003

[5] Pasquale Buonocunto, Alessandro Biondi, Marco Pagani, Mauro Marinoni, Giorgio Buttazzo, *ARTE: Arduino Real-Time Extension for Programming Multitasking Applications*, Scuola Superiore SantAnna, Pisa, Italy, November 2016.

[6] *The Real-Time Systems Laboratory (ReTiS Lab)*: one of the worlds leading research teams in the area of embedded real-time systems, time critical scheduling algorithms, advanced operating systems and adaptive resource management. http://retis.sssup.it/?q=arte

[7] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, Third Edition. Springer, New York, 2011.

[8] A. Biondi, G. Buttazzo, and M. Bertogna. Supporting componentbased development in partitioned multiprocessor real-time systems. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems (ECRTS 2015)*, Lund, Sweden, July 8-10, 2015.

[9] ElecFreaks, *Datasheet HC-SR04 ultrasonic proximity sensor*: Description of HC-SR04 measuring device characteristics. http://www.micropik.com/PDF/HCSR04.pdf

[10] Texas Instruments *Datasheet LM35 temperature sensor*: Description of LM35 measuring device characteristics. http://www.ti.com/lit/ds/symlink/lm35.pdf

[11] T. P. Baker. *Stack-based scheduling for realtime processes. Real-Time Systems*, 3(1):6799, April 1991.

Melvin Mancini received the B.S. degree in Computer and Automation Engineering from Marche Polytechnic University. He is currently pursing the Master's degree in Computer and Automation Engineering from Marche Polytechnic University in Ancona, Italy.