The Journal of Systems & Software 186 (2022) 111185

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

ARTe: Providing real-time multitasking to Arduino ☆

Francesco Restuccia [a,b,*], Marco Pagani [a], Agostino Mascitti [a], Michael Barrow [b], Mauro Marinoni [a], Alessandro Biondi [a], Giorgio Buttazzo [a], Ryan Kastner [b]

[a] Scuola Superiore Sant'Anna Pisa, Italy
[b] University of California San Diego, USA

ARTICLE INFO

Article history:
Received 24 September 2020
Received in revised form 20 November 2021
Accepted 6 December 2021
Available online 10 December 2021

Keywords:
Real-time
Multi-tasking
Arduino
Educational

ABSTRACT

In the last decade, thanks to its modular hardware and straightforward programming model, the Arduino ecosystem became a reference for learning the development of embedded systems by various users, ranging from amateurs and students to makers. However, while the latest released platforms are equipped with modern microcontrollers, the programming model is still tied to a single-threaded, legacy approach. This limits the exploitation of the underlying hardware platform and poses limitations in new application scenarios, such as IoT and UAVs.

This paper presents the Arduino real-time extension (ARTe), which seamlessly extends the Arduino programming model to enable the concurrent execution of multiple loops at different rates configurable by the programmer. This is obtained by embedding a low-footprint, real-time operating system in the Arduino framework. The adherence to the original programming model, together with the hidden support for managing the inherent complexity of concurrent software, allows expanding the applicability of the Arduino framework while ensuring a more efficient usage of the computational resources. Furthermore, the proposed approach allows a finer control of the latencies and the energy consumption. Experimental results show that such advantages are obtained at the cost of a small additional overhead and memory footprint. To highlight the benefits introduced by ARTe, the paper finally presents two case studies, one of such in which ARTe has been leveraged to rapidly prototype a mechanical ventilator for acute COVID-19 cases. We found that ARTe allowed our ventilator design to rapidly adapt to changes in the available components and to the evolving needs of Intensive Care Units (ICU) in the Americas.

1. Introduction

The Arduino project started in the early 2000s with the goal of providing a simple framework to support people with very limited programming skills in the development of embedded projects. This purpose was pursued by creating a user-friendly development environment based on a simple programming model. Then, the team started a company to design and produced a simple and low-cost board to create working prototypes interacting with the physical world. The growth of the project was fueled by the decision to adopt an open-source approach that aggregated a community keen on supporting the development, creating examples and tutorials, and providing libraries for a wide and expanding range of features and devices. Over the years, this approach proved to be successful and pushed the producers to develop more and more powerful boards, still compatible with the original programming model, as well as extension boards (namely shields) to expand hardware features and interoperability.

The programming model requires the application developer to provide a C file that only defines the `setup()` function, executed at the startup to initialize the system, and the `loop()` function, which executes endlessly. The simplicity of this solution significantly alleviates the learning phase and motivates the base of the massive number of projects using Arduino. However, in the last decade target applications became more and more complex, including multiple sensors and actuators, and required interaction with other systems through communication devices. These scenarios proved to be challenging to get along with the Arduino programming model. For instance, a typical solution adopted by Arduino users to deal with such a complexity consists in offloading the communication stack to a shield board that is generally more powerful than the microcontroller within the Arduino board. Delegating multi-rate sensors and communication devices

☆ Editor: Neil Ernst.

* Corresponding author.
E-mail addresses: frestuccia@ucsd.edu (F. Restuccia), marco.pagani@santannapisa.it (M. Pagani), agostino.mascitti@santannapisa.it (A. Mascitti), mbarrow@ucsd.edu (M. Barrow), mauro.marinoni@santannapisa.it (M. Marinoni), alessandro.biondi@santannapisa.it (A. Biondi), giorgio.buttazzo@santannapisa.it (G. Buttazzo), kastner@ucsd.edu (R. Kastner).

https://doi.org/10.1016/j.jss.2021.111185
0164-1212/

Listing 1: Implementation of functions executing at different rates making use of delay.

```
int count = 0;  // it counts the number of minor cycles
int T1 = 10;    // period (ms) for executing function1
int T2 = 25;    // period (ms) for executing function2
int T3 = 50;    // period (ms) for executing function3
int Tmin;       // GCD of the periods (minor cycle)
int Tmaj;       // lcm of the periods (major cycle)
int K1;         // counter value for triggering function1
int K2;         // counter value for triggering function2
int K3;         // counter value for triggering function3

Tmin = GCD(T1, T2, T3);  // minor cycle
Tmaj = lcm(T1, T2, T3);  // major cycle

K1 = T1/Tmin;      // number of minor cycles in T1
K2 = T2/Tmin;      // number of minor cycles in T2
K3 = T3/Tmin;      // number of minor cycles in T3
H = Tmaj/Tmin;     // number of minor cycles in Tmaj

while (!end) {
  if (count%K1 == 0) function1();
  if (count%K2 == 0) function2();
  if (count%K3 == 0) function3();
  count++;
  if (count == H) count = 0;
  delay(Tmin);     // suspend for a minor cycle
}
```

to external boards produces complex solutions that are also inefficient in terms of cost, weight, and power consumption. This problem is even more evident with modern Arduino platforms (e.g., the Arduino Due board), where the main microcontroller remains mostly underutilized.

1.1. Limits of the Arduino programming model

The Arduino programming model works fine for simple control systems where sensors have to be acquired at the same frequency, but becomes problematic when the control system requires actions that need to be triggered at different rates.

Consider, for example, a system equipped with an infrared sensor, an inertial sensor, and a communication transceiver, which need to be acquired with different periods, e.g., dictated by the sensor dynamics and the computation times required to process the corresponding data. Suppose that the infrared sensor has to be sampled every $T_1 = 10$ ms, the inertial sensor every $T_2 = 25$ ms, and the communication device every $T_3 = 50$ ms. The solution typically adopted by Arduino users in these situations is to program the main loop to execute with a period T_{min} (also called minor cycle) equal to the greatest common divisor (GCD) of the three periods and trigger the other activities every K_i executions of the main loop. In our example, we have:

$$T_{min} = 5\,ms; \quad K_1 = \frac{T_1}{T_{min}} = 2; \quad K_2 = \frac{T_2}{T_{min}} = 5; \quad K_3 = \frac{T_3}{T_{min}} = 10. \qquad (1)$$

A possible implementation of this approach is to use a counter that counts the number of minor cycles and calls the function that has to be executed with period T_i when the counter reaches the value $K_i = T_i/T_{min}$, as shown in Listing 1.

To avoid the counter overflow, the counter has to be reset when it reaches the value of the least common multiple (lcm) of the periods, also called major cycle. This solution works fine when the execution times of the three functions are negligible with respect to the periods (e.g., for the case of blinking LEDs). However, when the functions perform more complex computations

Listing 2: Implementation of functions executing at different rates making use of `millis()`.

```
#define N 3             // number of functions

int T[3] = {10, 25, 50}; // function periods (ms)
unsigned long time;    // current time (ms)
unsigned long prevtime; // previous time (ms)
int i;
void (*func_ptr[3])() = {function1, function2,
    function3};

time = millis();
for (i=0; i<N; i++) prevtime[i] = time - T[i];

while (1) {
  for (i=0; i<N; i++) {
    time = millis();
    if (time - prevtime[i] >= T[i]) {
      prevtime[i] = time;
      (*func_ptr[i])();
    }
  }
}
```

and their execution times are comparable with their periods, this approach does no longer guarantee a regular activation of the activities, since each function will delay the next one. Fig. 1 illustrates two examples of schedule: Fig. 1a shows the case in which computation times are negligible, whereas Fig. 1b shows the case in which they are not (namely, $C_1 = 2.5$ ms, $C_2 = 5$ ms, $C_3 = 10$ ms). The shadowed areas denote the delay at the end of the loop and the number above each area represents the value of the counter during that interval.

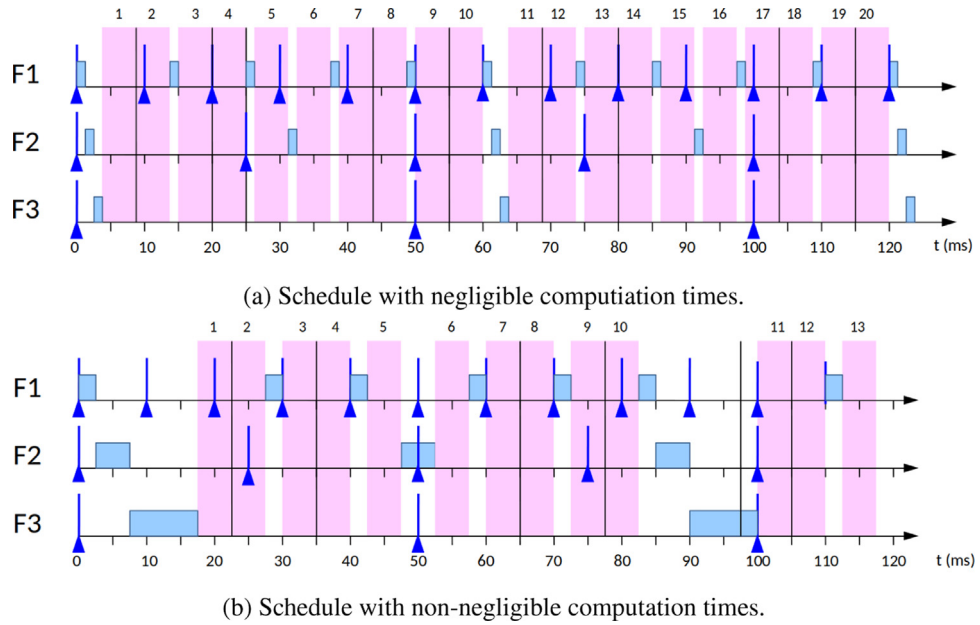
Note that, even in the case of negligible computation times, the functions tend to accumulate a delay that, after a number of executions, will cause a period skip. For instance, `function1` skips a period every six instances (see intervals [50 ,60] and [110, 120] in Fig. 1a).

In the case of non-negligible computation times (Fig. 1b), `function1` skips five periods out of eleven, and two of them are consecutive ([90, 100] and [100, 110]). Also, `function2` does not execute properly, since only one instance is executed in the interval [25, 75], equivalent to two full periods. It is worth observing that such a misbehavior is not due to an overload (in fact the overall processor utilization is $U = 2.5/10+5/25+10/50 = 0.65$, i.e., the 65%), but it is due to that specific implementation.

A better solution, sometimes used by Arduino developers, is to keep track of the activation times by the function `millis()`, which returns the number of milliseconds passed since the program started and activate each activity after a period is passed. An implementation following this approach is reported in Listing 2.

The implementation reported in Listing 2 reduces most of the delays introduced by the previous one, but it still does provide a general solution to the problem of executing functions at different rates. Fig. 2 illustrates two schedules produced for different computation times. Fig. 2a refers to the case in which computation times are $C_1 = 2.5$ ms, $C_2 = 5$ ms, $C_3 = 10$ ms, leading to a total processor utilization $U = 0.65$, whereas Fig. 2b refers to the case in which computation times are $C_1 = 5$ ms, $C_2 = 5$ ms, $C_3 = 15$ ms, leading to a total processor utilization $U = 1.0$.

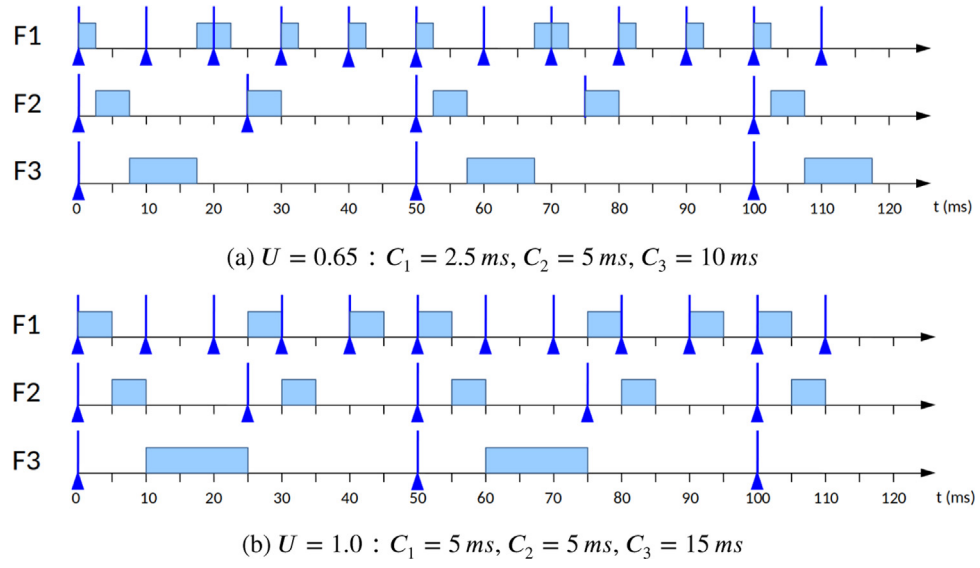
While the schedule in Fig. 2a respects all the specified periods, the one in Fig. 2b does not, since only three instances of function1 are executed out of five (in fact, the function is not executed in



(a) Schedule with negligible computiation times.

(b) Schedule with non-negligible computation times.

Fig. 1. Schedule obtained for the three functions under the solution illustrated in Listing 1, when their computation times are negligible (a) and when they are not (b). Triangles below the axes denote activation times and vertical bands represent minor-cycle suspension intervals (the number on top is the value of the counter).



(a) $U = 0.65 : C_1 = 2.5\,ms, C_2 = 5\,ms, C_3 = 10\,ms$

(b) $U = 1.0 : C_1 = 5\,ms, C_2 = 5\,ms, C_3 = 15\,ms$

Fig. 2. Schedule obtained for the three functions under the solution illustrated in Listing 2, for different values of the computation times.

the intervals [10, 20], [30, 40], [60, 70], [80, 90], and so on). The problem is that the functions are executed one after the other and cannot be *preempted* (i.e., temporarily interrupted to be later resumed) during their execution. For instance, in the schedule of Fig. 2b, at time $t = 10$ `function1` should be reactivated, but it cannot run since the sketch will call it in the next cycle, after the execution of `function3`, which completes at time $t = 25$, i.e., in the middle of the third period of `function1`. This type of problem can only be solved by handling the functions by a preemptive scheduler.

The solution presented in this paper extends the Arduino programming model by allowing the user to specify multiple loops, each with its own execution period, and by treating each loop as a concurrent thread scheduled by a preemptive scheduler provided by a real-time operating system. This is done by keeping the scheduler and the operating system transparent to the programmer, so that each loop can be developed by following the classical Arduino programming style. The software development of the overall application is actually simplified, since the user does not have to explicitly trigger the functions, as in the previous implementation shown above.

Listing 3 reports the code that implements the three functions according to the proposed extended programming model.

Paper structure. The rest of the paper is organized as follows: Section 2 presents an overview of the Arduino framework, the proposed extensions to provide multitasking, and the Erika Enterprise kernel that is leveraged in the proposed approach; Section 3

Listing 3: Implementation of functions executing at different rates using the ARTe programming model.

```
void setup() {
    <<setup code>>
}

void loop() {
    <<background code>>
}

void loop1(10) {
 function1();
}

void loop2(25) {
 function2();
}

void loop3(50) {
 function3();
}
```

Listing 4: Implementation of an application similar to the one proposed in Listing 3 using the FreeRTOS-Arduino programming model.

```
#include <Arduino_FreeRTOS.h>

void TaskLoop1(void *pvParameters);
void TaskLoop2(void *pvParameters);
void TaskLoop3(void *pvParameters);

void setup() {
  xTaskCreate(TaskLoop1,(const portCHAR *)"loop1",128,
      NULL,2,NULL);
  xTaskCreate(TaskLoop2,(const portCHAR *)"loop2",128,
      NULL,2,NULL);
  xTaskCreate(TaskLoop3,(const portCHAR *)"loop3",128,
      NULL,2,NULL);
}

void loop(){
  <<background code>>
}
void TaskLoop1(void *pvParameters) {
  (void) pvParameters;
  for(;;){
    function1();
  }
}
void TaskLoop2(void *pvParameters) {
  (void) pvParameters;
  for(;;){
    function2();
  }
}
void TaskLoop3(void *pvParameters) {
  (void) pvParameters;
  for(;;){
    function3();
  }
}
```

describes the proposed approach, while Section 4 highlights relevant implementation details. Experimental results and use cases are reported in Section 5, whereas Section 6 states our closing remarks.

Table 1
Features comparison of popular Arduino boards.

Arduino	Processor	Arch.	Freq.	SRAM	NV Memory
UNO	ATmega328	8 bit	16 MHz	2 KB	1 KB
DUE	Cortex M3	32 bit	84 MHz	96 KB	512 KB
Zero	Cortex M0+	32 bit	48 MHz	32 KB	256 KB

2. Background and state of the art

While the user experience for Arduino developers remained mostly unaltered along the years, the internals of the Arduino framework evolved to include new features and increase its modularity and portability. This section first presents the current status of the official Arduino framework and then provides an overview of some custom extensions proposed by other authors to support multitasking in Arduino. Finally, the section illustrates how the work proposed in this paper advances the state of the art and introduces the main features of the ERIKA Enterprise (Anon, 0000a)[1] *real-time operating system* (RTOS), which has been leveraged to realize the proposed solution.

2.1. The Arduino framework

Arduino is an open-source project based on easy-to-use embedded hardware and software. The Arduino ecosystem consists of a set of single-board microcontrollers and a software framework that comes with an *integrated development environment* (IDE). Over the years, many Arduino boards with different computational and I/O capabilities have been released. The first board has been the Arduino UNO, which is based on a Microchip ATmega microcontroller running at 16 MHz and offers 14 digital I/O pins and six analog input pins. The original Arduino UNO board has been later updated with different releases and is still supported nowadays. However, it offers limited computational capabilities and small amount of volatile (SRAM) and non-volatile (NV) memory. To overcome these limitations, the Arduino community released more advanced boards such as the Arduino DUE and Arduino Zero based on ARM Cortex microcontrollers. Both these boards dispose of more powerful computational and I/O capabilities with respect to Arduino UNO, and are hence better suited for more complex application scenarios. Table 1 reports a feature comparison of these three Arduino boards.

The software side of the Arduino project consists of a software framework, initially based on the Wiring project (Barragán, 2004), which includes an IDE in charge of the building process and the device flashing. A block diagram of the Arduino building process performed by the IDE is illustrated in Fig. 3. Basically, the building process can be divided into two phases: a pre-processing phase and a compilation-and-linking phase. During the pre-processing phase, the Arduino framework performs a few transformations like adding additional headers and generating prototypes for all the functions defined in the application source file, which is denoted by *sketch*. Then, the source files are passed to the compiler tool-chain to be compiled and linked with the Arduino libraries.

2.2. Related work

Several solutions are well known to support multitasking and provide solutions to simplify the development of applications on microcontrollers. An example is the work proposed by Rivas and Tijero (2019), which aims at simplifying the use of the Ada safety language for the development of applications running

[1] ERIKA Enterprise project website: https://www.erika-enterprise.com/.

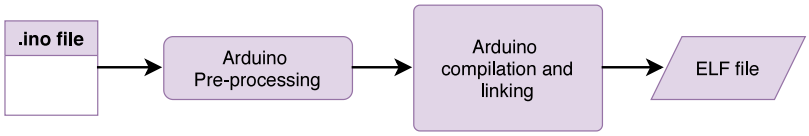


Fig. 3. Arduino building flow chart.

on small microcontroller devices. However, it is not straightforward to integrate some multitasking support in Arduino while *minimizing* the impact on its programming model and ensuring retro-compatibility. The Arduino programming model is based on the use of only two constructs: the `loop()` function and the `setup()` function. The former contains the code that is cyclically executed as long as the device is powered, while the latter is executed at the startup of the device. This simple programming model allows entry-level users to develop applications in Arduino without requiring specific skills in programming a microcontroller. On the other hand, this model is not suitable for multitasking as the user is limited to a single, sequential execution flow given by the code of the `loop()` function.

Several methods with different degrees of complexity have been proposed to fill this gap. The most straightforward ones do not require any third-party library or extension, and just allow defining the tasks as C functions to be executed in the standard Arduino `loop()` function. The scheduling logic is implemented with ad-hoc conditional statements and delay functions (such as `delay()` and `millis()`) to mimic a periodic activation of the tasks. In this paper, this basic approach is adopted as a baseline for comparison purposes: more details are provided in Section 5.

Other solutions were proposed by relying on the use of third-party libraries that implement multitasking in a way that is similar to the one just discussed, but offering a more friendly interface to the developer. Some examples of such libraries are the following:

- The Scheduler library (Anon, 0000b) allows registering multiple loops in the `setup` function that will cyclically be executed at run-time. There is no control in terms of periods and timing still relies on explicit delays, like in the original Arduino paradigm. In order to reduce the impact of loops with significant execution times, the library provides a `yield` function that the application programmer can explicitly use.
- SoftTimer (Kelemen, 0000) library enables multitasking by defining each task as a C++ object. The constructor of such objects takes as arguments the period of the task and a pointer to a callback function. The callback functions are meant to be executed in a periodic non-preemptable fashion according to the task period. Tasks are registered in the Arduino `setup()` function. This library originally prevented the use of the standard `loop()` function, hence breaking the original Arduino programming model. This limitation has been removed only recently.
- ArduinoThreads (Anon, 0000c) works similarly to the Scheduler library discussed above but implements a more complex set of constructs to manage the execution of tasks in a periodic fashion. This approach allows more flexibility but requires a significant impact in terms of knowledge and resulting application code.

Such a class of solutions suffer from the following drawbacks:

- *The programming model becomes more complex with respect to the original one*. Also, most of such extensions (with the exception of SoftTimer) do not provide explicit support for periodic activities. The user must explicitly specify initialization procedures and possible preemption points, with the result that code modifications are required to support multitasking.
- *Tasks are executed cooperatively*. Under this multitasking approach, the context switch is triggered by the running task, which voluntarily yields the processor to other tasks. This increases the latency variability and the complexity in guaranteeing response-time bounds for the tasks.
- *No protection against task overruns*. When a task instance runs longer than its period, the entire schedule can experience a domino effect, jeopardizing the whole application.

Many of such issues are originated by the fact that no operating system is adopted, hence forcing the user to implement some form of scheduling in the `loop()` function. This approach can be particularly error-prone, especially for users that are not familiar with scheduling techniques and concurrent programming.

FreeRTOS-Arduino (Barry, 0000) has been proposed to address these issues. It is based on a porting of the FreeRTOS kernel as an Arduino library and provides fixed-priority preemptive scheduling. Despite being a powerful solution, FreeRTOS-Arduino introduces several complications at the level of the programming model, as it requires the user to be confident with both the FreeRTOS API and concurrent programming, which are skills that typically exceed those of the general user of Arduino. A sample application developed following the Arduino-FreeRTOS programming model is reported in Listing 4.

Furthermore, FreeRTOS is not a static operating system, i.e., not all kernel code and data structures can be tailored to the application at compile time. Therefore, it is characterized by a larger footprint,[2] and memory and run-time overhead with respect to those that would be actually required to handle a certain set of tasks. This waste of resources can be a severe issue on resource-constrained platforms such as Arduino UNO and Arduino Zero.

Another solution based on an RTOS is Qduino (Cheng et al., 2015), which extends the Arduino framework with a custom API to allow implementing concurrent loops with support for mutual exclusion on shared resources. As for FreeRTOS-Arduino, Arduino users are required to acquire additional knowledge on real-time concurrent programming and the specific Qduino API. In addition, Qduino only supports Arduino platforms based on Intel x86 processors (e.g., Galileo, Arduino 101). Therefore, this solution is not compatible with the majority of Arduino boards.

Note also that the Arduino framework comes with a rich set of libraries that have been developed to work under singletasking, i.e., their internal state can be left inconsistent whenever they are suspended to execute other computational activities. Both FreeRTOS-Arduino and Qduino require explicitly managing mutual exclusion when using these libraries, hence complicating the programming model even for simple and typical operations that are present in many Arduino example sketches.

[2] For instance, the footprint of a simple blink application with FreeRTOS on Arduino UNO is 8264 bytes (25% of the available memory).

Like FreeRTOS-Arduino and Qduino, also ARTe (Buonocunto et al., 2016) relies on an RTOS to support multitasking, but differently from such previous approaches, it preserves the simplicity of the Arduino programming model. ARTe is characterized by a minimal impact in terms of footprint and run-time overhead. This was possible by building ARTe upon the ERIKA Enterprise (Anon, 0000a) RTOS that, among other choices (e.g., FreeRTOS Anon, 0000d and NuttX Anon, 0000e), resulted in an excellent candidate to efficiently use the scarce computational resources of Arduino platforms thanks to its *static* configuration at compile time. The preliminary version of ARTe presented in Buonocunto et al. (2016) did not provide support to handle shared variables and mutual exclusion. Furthermore, the work in Buonocunto et al. (2016) was conceived for older versions of the Arduino IDE, which were designed in a monolithic way; thus, it was not possible to customize the building process without modifying the IDE source code. Consequently, Buonocunto et al. (2016) was based on a customized version of the IDE, extensively modified to integrate a custom parser and a modified build process to support the RTOS. However, this approach originates considerable limitations in terms of portability and extensibility, as any new release of the Arduino IDE requires a porting of the extension presented in Buonocunto et al. (2016) while major updates of the IDE codebase may totally break compatibility. Thankfully, the most recent versions of the Arduino IDE include a more flexible build mechanism that allows developers and board manufacturers to customize the entire building process (Anon, 0000f). The mechanism works by exporting to the developer a set of *hooks* (i.e., code injection points) for each step of the building process.

2.3. This work

This paper presents a new version of ARTe (officially called ARTe v2 and referred to as just ARTe in the following) that introduces novel features to support multitasking and a radically different way to integrate multitasking in the novel Arduino IDE. In particular, such a new version of ARTe includes an automatic protection mechanism for global variables and a fine-grained mechanism that allows extending current Arduino libraries for thread safety. Both mechanisms are crucial for enabling the seamless integration of the large Arduino codebase within the ARTe multi-threaded environment. Furthermore, ARTe supports the most common Arduino platforms, i.e., Arduino UNO and Arduino Due.

Differently from Buonocunto et al. (2016), thanks to a new development available in recent versions of the Arduino IDE, ARTe does not require any modification to the official source code of the IDE and can easily be updated and maintained as a third-party plug-in module.

To keep the paper self-contained, a brief description of ERIKA Enterprise is reported next.

2.4. ERIKA enterprise

ERIKA Enterprise (Anon, 0000a) is an RTOS that offers a real-time scheduler and resource managers suited for developing highly-predictable applications on microcontrollers. ERIKA is characterized by a very small run-time overhead (in the order of a few microseconds) and a tiny memory footprint (a few kilobytes). ERIKA has been certified to conform to the OSEK/VDX (Anon, 0000g) standard and implements the OSEK/VDX API. Following the OSEK/VDX standard, all objects provided by ERIKA, such as tasks, alarms, and semaphores, must be statically defined alongside the application. That is, the RTOS configuration and all application objects must be known at compile-time and cannot be changed at run-time.

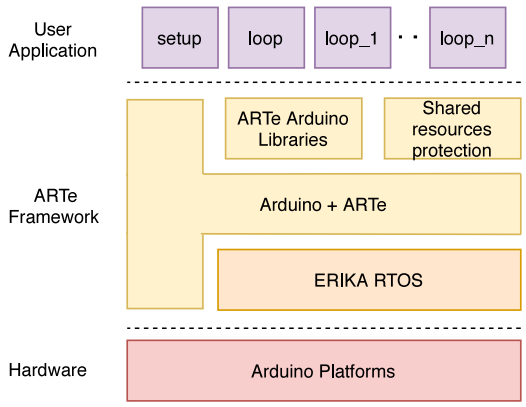


Fig. 4. General architecture of the ARTe framework.

This approach is crucial for ensuring a small run-time overhead and contain the memory footprint, as it allows producing tailored images of the RTOS that are optimized for a specific application. While designing an application, the programmer can define the RTOS objects and the kernel configuration by using the OSEK Implementation Language (OIL). OIL files are translated by RT-Druid, a tool included in the ERIKA developing environment, into a set of C source files that define the kernel configuration. Once the kernel configuration files are generated, the customized ERIKA kernel can be compiled together with the application code.

Aside from the standard OSEK/VDX features, ERIKA also provides additional conformance classes, such as fixed-priority scheduling with preemption thresholds (Wang and Saksena, 1999), Earliest Deadline First (EDF) scheduling algorithm (Liu and Layland, 1973), resource reservations (Marzario et al., 2004), and hierarchical scheduling (Bertogna et al., 1991; Biondi et al., 2015).

ERIKA provides two types of *interrupt service routines* (ISRs) for handling interrupts: (i) fast interrupts, called *Type 1 ISRs*, and (ii) lower-priority interrupts, called *Type 2 ISRs*. Type 1 ISRs are meant to be used for short and urgent I/O operations, returning to the application without calling kernel services (e.g., the scheduler). On the other hand, Type 2 ISRs can call selected kernel primitives and interact with the scheduler (e.g., activating a task), but introduce a larger latency than Type 1 ISRs.

3. Proposed approach

This section presents a general overview of the approach proposed in this paper, with a particular focus on the extensions to the original Arduino programming model and the integration within its toolchain.

3.1. General architecture

The general architecture of the ARTe framework is illustrated in Fig. 4. ARTe is designed to keep its programming model as similar as possible to the standard Arduino one, meaning that it supports the `setup` and `loop` functions with minimal modifications to their original behaviors.

The key difference with respect to the standard Arduino programming model is that ARTe allows the user to execute concurrent tasks (such as *loop*_1, *loop*_2, . . . , *loop_n* in Fig. 4), whose definitions are substantially similar to the standard Arduino `loop` function. Each task (i.e., each loop) is executed in a *periodic* fashion and its period can be specified as an argument of the

corresponding function. The ARTe programming model is detailed in Section 3.2.

Since the original Arduino programming model has not been originally designed to support multitasking, this feature may originate race conditions in accessing shared resources, such as global variables, or shared peripheral devices. To address this issue and safely support multitasking, ARTe comes with two protection mechanisms for shielding shared resources. These mechanisms can then be leveraged by concurrent tasks to synchronize with each other, e.g., to avoid leaving inconsistent a global data structure or the internal state of a device in the presence of a task preemption. In particular, to avoid requiring the user to explicitly synchronize the access to global variables under multitasking, ARTe comes with a transparent protection that automatically takes care of possible race conditions. This is accomplished by employing *static code analysis* and a form of *wait-free synchronization*: further details are provided in Section 3.4.

ARTe also allows tasks to use Arduino libraries (both the official ones and those provided by third-party contributors). However, as Arduino libraries are typically not designed to support multitasking, some modifications to them are required. To this purpose, ARTe offers a simple locking interface for shared resources protection, which is discussed in details in Section 3.5. A set of popular and essential Arduino libraries have already been modified and are distributed with the ARTe package.

Under the hood, ARTe adopts the ERIKA Enterprise RTOS. All the RTOS services and their configuration are completely masked to the ARTe user. ARTe already supports the Arduino UNO and Arduino DUE platforms.[3] The support for Arduino Zero is currently under development (no significant differences in the implementation are expected). Thanks to the modular implementation and the flexibility of ERIKA, limited efforts are envisaged to support other popular Arduino platforms.

3.2. ARTe programming model

As already mentioned in Section 2, the key design principle of ARTe is to preserve the simplicity of the Arduino programming model. As a first step towards matching this principle, both the `setup` and `loop` functions are preserved in ARTe. However, while the `setup` function maintains the original functionality, in ARTe the `loop` function hosts code that is executed in background, i.e., whenever there are no tasks ready to execute. Multitasking is supported by extending the semantic of the `loop` function. In ARTe, the user can define an arbitrary number of tasks, each specified by a function of the following format:

```
void loop<name>(int period) {
    // Here goes the task code
}
```

Specifically, all C functions whose name begin with the keyword `loop` and ends with other characters are interpreted as tasks. Examples of suitable names for such functions are `loopi`, `loop_42`, `loopfoo`. To maintain the simple and intuitive approach of Arduino, such functions have been designed to take one and only one argument that denotes the period, in milliseconds, with which the corresponding task must be executed. For instance, the following code defines a task in ARTe that prints “Hello World" in the serial console every 100 ms:

```
void loop_example(100) {
    Serial.println("Hello world");
}
```

[3] For Arduino UNO the newest version 3 of ERIKA is used, while for Arduino DUE ERIKA version 2 is adopted since Erika v3 does not still support the corresponding Cortex-M microcontrollers.

Listing 5: Example of periodic activities using standard Arduino programming model.

```
void setup() {
 << setup code >>*@void loop() static int ticks =
      0;const int interval = 10;if (ticks activity1();if
      (ticks activity2();if (ticks activity3();ticks +=
      interval;if (ticks == 210)ticks =
      0;delay(interval);
```

More complex examples are presented in the following sections.

3.3. Example of an ARTe application

With the traditional Arduino programming paradigm, applications consisting of concurrent periodic activities can be programmed using the (so-called) loop scheduling technique. Listing 5 shows a sample application that comprises three periodic tasks: `activity1`, `activity2`, and `activity3`, having periods of $T_1 = 10$ ms, $T_2 = 30$ ms, and $T_3 = 70$ ms, respectively. The delay value at the end of the Arduino loop defines the granularity of the periodic activations. The optimal delay value corresponds to the greatest common divisor of the tasks' periods (10 in this example). A time counter (`ticks` in the example) is used to detect when the tasks need to be activated. Note that the time counter can be reset only after the time at which the schedule repeats itself (known as the *hyperperiod*). The hyperperiod is equal to the least common multiple of the tasks' periods, which equals to 210 in the presented example.

As anticipated in the introduction of this paper, this approach works reasonably well when the application consists of tasks that have very short computation times, e.g., tasks for blinking a set of LEDs, but is unsuitable for complex applications that include tasks with short computation times and short periods along with tasks with long computation times and long periods.

ARTe solves these issues by relying on the multitasking support offered by ERIKA, which implements fixed-priority preemptive scheduling. Listing 6 shows how the application of Listing 5 can be implemented under ARTe. As it can be observed from the listing, compared to the baseline solution discussed above, ARTe provides a simpler, more concise, and less error-prone programming paradigm.

Listing 7 shows an excerpt of the OIL configuration file generated by the ARTe builder for the sample application reported in Listing 6, while Listing 8 presents the extended `setup` function. ARTe creates a variable of type COUNTER named TaskCounter,

Listing 6: Example of periodic activities using ARTe programming model.

```
void setup() {
 << setup code >>*@void loop() @*<< background code
      >>*@void loop1(10) activity1();void loop2(30)
      activity2();void loop3(70) activity3();
```

Listing 7: OIL configuration for the periodic activities examples.

```
CPU m3 {
 << ERIKA OS settings >>*@ TASK loop1 PRIORITY =
      3;@*<<Task settings>>*@ ;ALARM Alarmloop1 COUNTER
      = TaskCounter;ACTION = ACTIVATETASK  TASK =
      loop1;;;TASK loop2 PRIORITY = 2;@*<<Task
      settings>>*@ ;ALARM Alarmloop2 COUNTER =
      TaskCounter;ACTION = ACTIVATETASK  TASK =
      loop2;;;TASK loop3 PRIORITY = 1;@*<<Task
      settings>>*@ ;ALARM Alarmloop3 COUNTER =
      TaskCounter;ACTION = ACTIVATETASK  TASK =
      loop3;;;;
```

Listing 8: Generated setup function for the periodic activities examples.

```
void setup() {
 << setup code >>

    SetRelAlarm(Alarmloop1, ARTE_TASK_INIT_OFFSET, 10)
        ;
    SetRelAlarm(Alarmloop2, ARTE_TASK_INIT_OFFSET, 30)
        ;
    SetRelAlarm(Alarmloop3, ARTE_TASK_INIT_OFFSET, 70)
        ;
}
```

which is incremented by the ISR handling the hardware timer. The counter can be connected with multiple alarms (e.g., `Alarmloop1`), each one in charge of activating a task when the desired value (a multiple of the hardware timer period) is reached. This configuration is static and specified in the OIL file shown in Listing 7. Instead, the current value for each period can be varied; hence the periods are configured injecting the needed code in the `setup` function, as shown in Listing 8.

3.4. Sharing data among tasks

Global variables are one of the most practical solutions for implementing communications channels among concurrent tasks in a shared-memory environment. Furthermore, global variables are a natural choice for preserving the simplicity of the Arduino framework being in line with the intuitiveness of the typical programming style of Arduino users. However, in a concurrent environment, global variables must be accessed in mutual exclusion to avoid race conditions and preserve the program correctness. In a more traditional programming environment, a professional programmer is responsible for implementing a proper access to global data by employing adequate primitive objects, like mutexes or monitors. In the domain of real-time operating systems, several resource access protocols, such as the Priority Inheritance Protocol (PIP), the Priority Ceiling Protocol (PCP), and the Stack Resource Policy (SRP) have been designed to bound the blocking delays caused by concurrent resource accesses. However, correctly using these mechanisms requires expertise in concurrent programming, which may be beyond the typical Arduino user background. Therefore, to preserve the accessibility of the original Arduino framework, ARTe automatizes

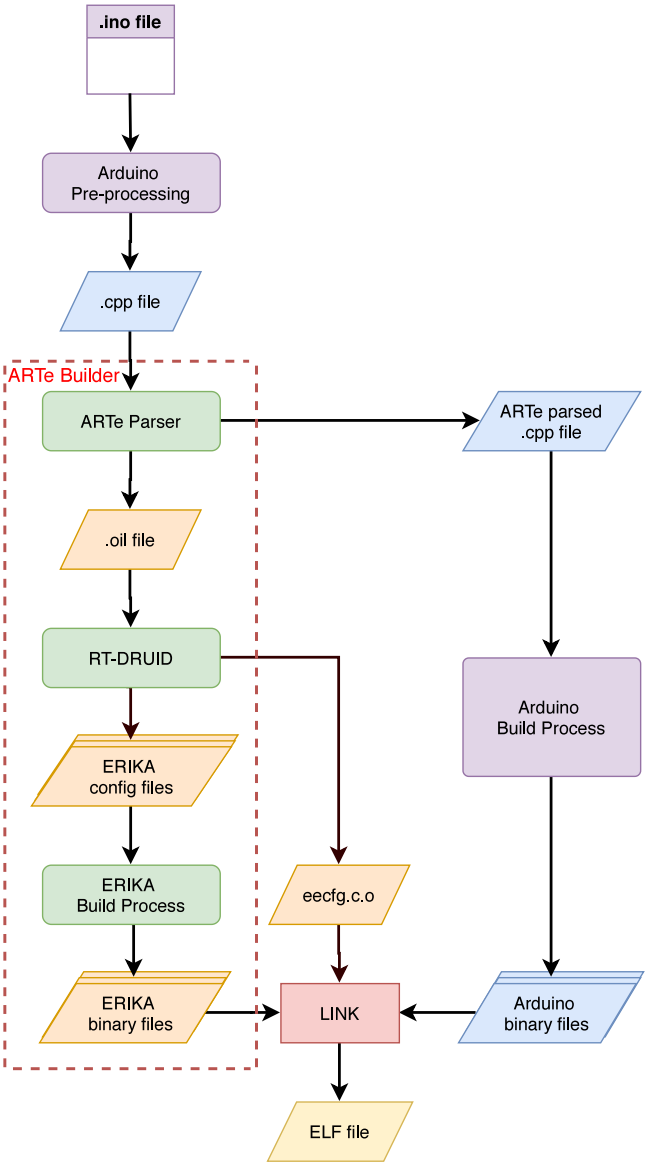


Fig. 5. Flow chart of the ARTe building process.

the handling of shared global variables. This approach enforces correctness and prevents errors originating from race conditions, which are notoriously challenging to be spotted.

The simplest way for implementing an automatic protection mechanism for global variables would be to protect all of them with a centralized lock that is taken at the beginning of each task and released at the end of the task. Unfortunately, such a simple approach reduces concurrency and may introduce unnecessary and long blocking times in high-priority tasks, which is undesirable for systems with timing constraints. ARTe protects global variables by utilizing a more evolved technique based on *local proxy variables*. During the compilation process, the ARTe parser, a component of ARTe that transforms the ARTe program into intermediate files that are used afterwards in the compilation (see Fig. 5), creates a table listing all global variables and then analyzes all tasks to detect the statements in which such global variables are accessed. If more than one task accesses a global variable, the builder defines a Mutex object associated with that global variable. Then, on each task that accesses that variable, the builder (i) replaces the global variable with a local proxy variable, and (ii) injects code fragments at the head and at the tail of the task's code

to synchronize the local proxy with the global variable. Such code fragments are referred to as synchronization *prolog* and *epilogue*, respectively. In practice, the synchronization prolog locks the mutex associated with the global variable, performs a copy of the global variable on the local proxy, and then releases the mutex. The synchronization epilogue locks the global variable mutex, copies back the value of the local proxy variable into the global variable, and then releases the mutex. If a task performs read-only access on a global variable, the ARTe builder injects only the head snippet. The builder ignores global constants. It is worth noting that this mechanism is entirely transparent to the user, it does not require changes to the programming paradigm, and is fully compatible with existing code. The approach is applied to all global variables to maintain a consistent behavior across different platforms (hardware-dependent optimizations for atomic variables are possible but currently not supported).

While this approach is simple and effective, it also has some limitations. Indeed, to synchronize a global variable with its local proxy, it is necessary to know the size of the variable at compile time. Therefore, the ARTe protection mechanism does not support dynamically allocated memory and data types accessed through pointers (e.g., linked lists). In fact, the ARTe protection mechanism is limited to standard C++ primitive data types and user-defined C-like passive data structures (PDS). However, since typical Arduino sketches do not make use of pointers and user-defined types, this limitation is expected to have a minimal impact. Please note that the scope of the ARTe protection mechanism is limited to the Sketch code. On the contrary, libraries developers are responsible for protecting library code against race conditions using the mutual exclusion support mechanism described in Section 3.5. Finally, it is worth remarking that the ARTe protection mechanism is not semantically equivalent to mutual exclusion. Indeed, the protection mechanism is intended to provide easy-to-use deadlock-free communication channels between tasks based on local proxy copies. Such local copies are initialized at the beginning of the task with the value of the corresponding global variable. Then, global variables are updated back at the end of the task with the values of local copies. Hence, whenever multiple tasks access the same global variable, the updates made by one task are not visible to the other tasks until it completes and the other tasks begin a new job. For this reason, it is recommended to use only one writer task for each global variable, i.e., each global variable implements a 1-to-N channel.

3.5. Adapting libraries for concurrent execution

The Arduino framework owes part of its popularity to the large codebase of proprietary and third-party libraries that relieve the programmer from the need of knowing the low-level details of each hardware peripheral. However, as stated before, the majority of such libraries are not thread-safe and thus cannot be directly integrated into ARTe. A simple solution for adapting existing Arduino libraries to the ARTe multi-threaded environment would be to execute each library call as a non-preemptive section. However, this approach would be unsuitable for real-time applications since some libraries include long busy waits that may jeopardize the timing performance of the entire application, as no other task would be capable of making progress in its execution during such busy waits.

To address this issue while preserving the simplicity of the Arduino framework, ARTe provides a fine-grained support mechanism that allows developers to extend libraries for a multi-threaded environment. The support mechanism provides the developers with a set of primitives, summarized in Listing 9, which can be used to define critical sections. The `arteLockRes()` and `arteUnlockRes()` primitives allow the programmer to define critical sections for protecting specific hardware resources. From the programmer perspective, these resources are available using an enumerated type as visible from Listing 10. On the ERIKA side, mutual exclusion is implemented using a predefined set of OSEK `RESOURCE` objects representing the basic set of I/O devices available on Arduino boards. Additional or custom peripheral devices that are not included in the set of predefined resources reported in Listing 10 can be protected using the global `arteLock()` and `arteUnlock()` primitives. These functions implement mutual exclusion with a global lock related to a special resource, named `RES_SCHEDULER`, which is part of the OSEK standard. When the calling task acquires such a resource, it becomes non-preemptive until it releases the resource. To avoid unbounded priority inversions, ERIKA makes use of the Immediate Priority Ceiling protocol (also known as Highest Locker Priority) (Buttazzo, 2011) while accessing these resources. Listing 11 shows how an existing library can be extended to support multitasking in ARTe by including critical sections using the primitives mentioned above.

Listing 9: ARTe locking primitives.

```
void arteLock(void);
void arteUnlock(void);

void arteLockRes(enum arteRes resource);
void arteUnlockRes(enum arteRes resource);

/* For testing purposes */
uint8_t arteEnabled(void);
uint8_t arteNestingLevel(void);
uint8_t arteNestingLevelRes(enum arteRes resource);
```

Listing 10: ARTe resources declaration.

```
enum arteRes {
 arteIO,
 arteADC,
 arteDAC,
 arteSPI,
 arteI2C,
 arteTIMER,
 arteUSART,
 arteSTREAM
};
```

Listing 11: Example of an Arduino library function extended using ARTe locking primitives.

```
int TwoWire::read(void) {
 int retval;

 /* *** ARTe - begin critical section *** */
 arteLockRes(arteI2C);

 if (rxBufferIndex < rxBufferLength)
  retval = rxBuffer[rxBufferIndex++];
 else
  retval = -1;

 /* *** ARTe - end critical section *** */
 arteUnlockRes(arteI2C);

  return retval;
}
```

Since ARTe follows a programming model based on periodic activities, there is no need to explicitly introduce delays in the code using functions such as millis(), micros(), delay(), and delayMicroseconds(). More importantly, the use of such functions is discouraged, since they could introduce delays longer than expected due to preemptions, depending on their internal implementation.

Finally, the support library provides a set of auxiliary functions that can be used by the developer for debugging and testing purposes.

The `arteLockNestingLevel()` and `arteLockNestingLevelRes()` functions return the nesting level of the critical section. The `arteEnabled()` function returns `true` if the ARTe framework is enabled or `false` if ARTe is disabled and the code is compiled using the regular Arduino environment.

4. ARTe implementation

This section describes the entire process employed by ARTe to produce the final (binary) executable of the application to be programmed on the platform starting from the application code. To avoid discussing several aspects of the Arduino framework that do not pertain to the contribution of this work, the presentation is mainly focused on the modifications that have been performed to the Arduino building process. Particular attention is taken at the code parsing stage provided by the ARTe parser.

The starting point for the ARTe building procedure remains the sketch file written by the user. ARTe just requires sketches written according to the programming model presented in the previous section with at least one periodic task defined. The trivial case composed only of the classic Arduino *loop()* is not handled by ARTe, since it would add the cost due to the ERIKA kernel to perform the same activities already provided by Arduino itself.

4.1. ARTe build chain

Fig. 5 shows the build flow for an ARTe application, which starts with the standard Arduino pre-processor that generates a C++ source file from the sketch (.ino file). A parsing stage (i.e., the ARTe parser) is first employed to process the C++ source file for the purpose of producing two outputs: another Arduino-compatible C++ source file and an OIL configuration file for the ERIKA kernel. These two files are then processed in parallel by the following stages (see Fig. 5). One branch involves the regular Arduino building process, whose documentation is available on the official Arduino website (Anon, 0000h). The other branch, called ARTe Builder and invoked through a pre-build hook recipe, is in charge of managing all the stages to obtain an instance of the ERIKA kernel tailored to the application under compilation. The main stages of the build flow are discussed in details next.

Arduino pre-processing. This is the first stage invoked by the Arduino Builder and is in charge of creating a unified C++ file that includes all the Arduino-related header files and the user code present in the sketch. Since the Arduino IDE gives no access to the sketch, ARTe has been designed to process the C++ file generated by this stage.

ARTe parser. This is the core component of ARTe. It has been developed in the Java programming language to simplify extensibility and relies on Doxygen (Anon, 0000i) as a parsing engine. The ARTe parser inputs the C++ file produced by the Arduino pre-processing stage and provides two outputs. The first one is the configuration file for the ERIKA RTOS (OIL file), which specifies a number of parameters such as the number of tasks and their setting (e.g., priorities), the number of shared resources and their affinity with the tasks, the timer to periodically activate the tasks, etc. This file is created starting from a template OIL file that contains the configuration parameters of ERIKA that pertain to the hardware platform. Finally, the OIL file is provided to the RT-DRUID configuration tool (distributed with the ERIKA RTOS) to generate tailored version of the RTOS as a linkable library. The second file is a processed version of the original sketch code augmented with the necessary calls to the ERIKA kernel. These system calls will be later resolved at linking time while combining the sketch object file with the ERIKA library. In this way, the ERIKA core and the Arduino code can be compiled through different compilation flows and combined together only in the end during the linking phase. This approach allows for maintaining a higher level of modularity, facilitating maintainability. Fig. 6 details the internal behavior of ARTe parser.

Arduino Build Process. In this stage, the Arduino Builder proceeds by compiling all the Arduino-related files needed by the application and produces a library. At the end of the build process, the Arduino binary files are provided to the linker together with ERIKA binary files.

RT-DRUID. When this stage is executed, the ERIKA configuration file produced by ARTe is ready to be analyzed by the RT-DRUID tool, which generates C code to configure ERIKA and a makefile to build the RTOS.

ERIKA build process. After the execution of RT-DRUID, it is possible to perform the compilation of ERIKA. This building process produces in output two libraries. The first one contains the architecture-independent code of the ERIKA kernel, while the second one includes the code needed to instantiate it on the specific platform and is different for each available platform.

Linking stage. The linking stage is realized by extending the standard linking procedure performed by Arduino, which has been modified to merge the ERIKA kernel produced by ARTe with the object files of the application. The result is an ELF binary file ready to be downloaded on the board via the standard Arduino tools.

4.2. Internals of the ARTe parser

This section details the internal steps performed by the ARTe parser.

C++ file split. The C++ file produced by the Arduino pre-processing stage contains a long inclusion of Arduino-related header files. ARTe does not need to analyze the code in these header files (and performing such an activity would slow down the build process). Therefore, as a preliminary step, ARTe extracts the user source code from the C++ file in input. The user code is saved in a temporary file and utilized as the input for the static analysis performed by Doxygen. The ARTe parser works on this temporary file and finally publishes the changes it applies in the original C++ file, hence replacing the user code generated by the Arduino pre-processing stage.

Doxygen Analysis. Doxygen analyzes the user code and provides an XML file that models the code structure and allows identifying all the functions and all the global variables. This XML file is imported in the Java environment using the Java Element interface (Anon, 0000j) to keep track of functions and global variables in proper data structures.

ERIKA task creation. All the ARTe loops are converted into ERIKA tasks, like in the example reported in Listings 12 and 13. All other functions defined in the sketch are not modified by ARTe. The background `loop` is executed as background activity, i.e., a task running with the lowest priority in the system.

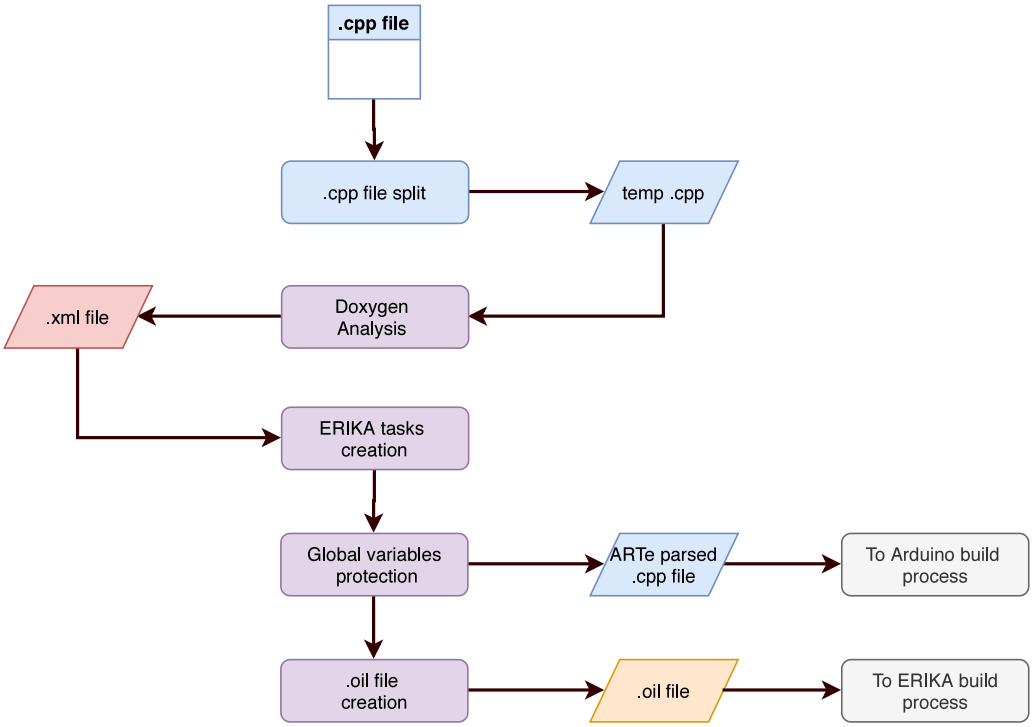


Fig. 6. Flow chart that describes the interval behavior of the ARTe parser.

ERIKA alarms, which control the periodic activation of tasks, are configured at the end of the Arduino `setup` function by injecting calls to the corresponding ERIKA primitives. In this way, tasks can start executing only after the user code in the `setup` function has completed its execution. The definitions of some ERIKA functions are also injected at the top of the temporary C++ file. Furthermore, some header files are included to access the ERIKA and ARTe API.

Global variables protection. Thanks to the Doxygen analysis, the ARTe parser is capable of detecting which global variables are accessed by more than one task. Such global variables are then protected with the mechanism discussed in Section 3.4, which also requires defining an ERIKA resource for each global variable in the OIL configuration of the RTOS. As stated in Section 3.4, the protection mechanism is restricted to primitive types and C-like passive data structures.

The functions of the ERIKA API that are used to implement mutual exclusion are:

- void `GetResource(ResourceType ResID)`, to lock the resource specified as a parameter; and
- void `ReleaseResource(ResourceType ResID)`, to unlock the resource specified as parameter.

More specifically, the protection of a global variable `var` requires performing the following actions:

1. The name of the global variable is changed from `var` to `__ARTE_GLOBAL_var__`;
2. A local variable `var` is defined in the local scope of each ARTe task that uses the global variable of interest;
3. A short critical section is placed at the beginning of the task body to safely perform the copy of the value of the global variable `__ARTE_GLOBAL_var__` into the local variable `var`;
4. If the task performs some modifications on the global variable, which can be detected via the Doxygen analysis, another short critical section is placed at the end of task body to update the global variable `__ARTE_GLOBAL_var__` with the content of the local variable `var`. Such an approach ensures that the updated value is correctly saved on the global variable when the task terminates the execution, hence publicizing the output produced by the task.

Oil file creation. The last step is the generation of the OIL configuration file, which follows the OSEK standard to configure the ERIKA RTOS. This step makes use of a template file, which is chosen by the ARTe builder within a library of templates as a function of the Arduino board selected by the user. The template contains all the architecture-dependent configurations and parameters. The final OIL file produced by this stage includes the declarations of all the global resources and tasks. Each task is assigned a priority based on the Rate Monotonic algorithm, i.e., the shorter the task period the higher the priority. ERIKA alarms are used to periodically activate the tasks. They are set together with the corresponding action to take when the alarm fires (i.e., the task to be executed). An example of how a task and the relative alarm are declared in the file is reported in Listing 8.

5. Experimental evaluation

This section presents a practical evaluation conducted to assess the performance of the ARTe framework.

As a first step, the scalability of ARTe with respect to the traditional approach is evaluated by comparing the memory footprint of the runtime support as a function of the number of loops. Following, two complete case-study applications have been developed to test the ARTe framework in a realistic scenario where multiple peripherals devices on the Arduino board are used under multitasking.

5.1. Memory footprint

Arduino boards are typically memory-constrained embedded platforms. Therefore, to assess the sustainability of the ARTe

Listing 12: Example of a sketch containing two ARTe loops sharing a global variable.

```
int global_var;

void loop_1(100)
{
  int local_var;

  local_var = global_var;
}

void loop_2(100)
{
  global_var++;
}
```

Listing 13: Corresponding code generated by the ARTe builder.

```
TASK (loop_1)
{
 //--------------------------------
 int global_var;
 GetResource(__ARTE_MUTEX_global_var__);
 memcpy(&global_var,&__ARTE_GLOBAL_global_var__,
      sizeof(int));
 ReleaseResource(__ARTE_MUTEX_global_var__);
 //--------------------------------

 int local_var;

 local_var = global_var;
}

TASK (loop_2)
{
 //--------------------------------
 int global_var;
 GetResource(__ARTE_MUTEX_global_var__);
 memcpy(&global_var,&__ARTE_GLOBAL_global_var__,
      sizeof(int));
 ReleaseResource(__ARTE_MUTEX_global_var__);
 //--------------------------------

 global_var++;

 //--------------------------------
 GetResource(__ARTE_MUTEX_global_var__);
 memcpy(&__ARTE_GLOBAL_global_var,&global_var__,
      sizeof(int));
 ReleaseResource(__ARTE_MUTEX_global_var__);
 //--------------------------------
}
```

framework, it is worth evaluating the impact of the runtime support in terms of memory consumption. To this end, a simple modular application consisting of up to 16 tasks (each performing a single GPIO operation only) has been programmed with both the standard Arduino framework using the loop scheduling technique, and then with the ARTe framework. Table 2 reports and compares the memory footprint of both implementations while varying the number of loops (in bytes and as percentage of the total amount of resources available on the board). These measurements have been collected when building an application for the Arduino Due platform.

Table 2
Evaluation of the memory footprint of ARTe vs. the standard Arduino programming model on an Arduino Due platform.

Number of loops	Footprint (bytes)	
	Arduino	ARTe
1	10708 (2.042%)	12556 (2.395%)
2	10732 (2.047%)	12608 (2.405%)
4	10788 (2.058%)	12696 (2.422%)
8	10900 (2.079%)	12880 (2.457%)
16	11124 (2.122%)	13248 (2.527%)

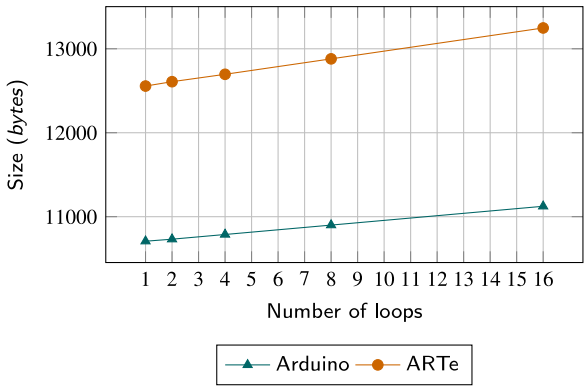


Fig. 7. Arduino and ARTe footprints on the Arduino Due platform with respect to the number of tasks.

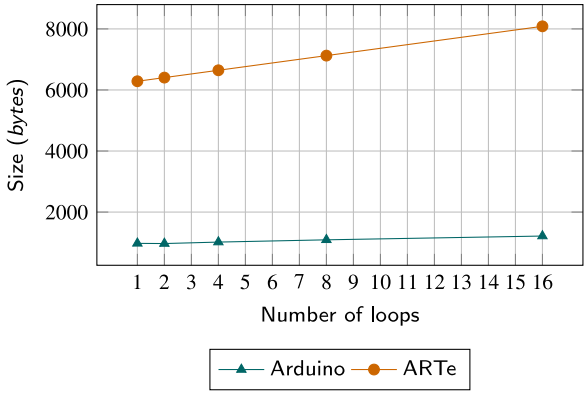


Fig. 8. Arduino and ARTe footprints on the Arduino UNO platform with respect to the number of tasks.

The results obtained for the case of a single task show that the ARTe runtime support, i.e., the ERIKA RTOS plus the support libraries for mutual exclusion, requires only 1848 bytes of additional memory in the worst case, corresponding to less than 0.4% of the available memory on an Arduino Due. It is also worth noting that the memory footprint scales almost linearly with the number of tasks with a rate of fewer than 50 bytes per task. Fig. 7 shows a graphical representation of this trend where the first value of the curve labeled Arduino is the size of the standard Arduino sketch, including only the *loop()* function.

The same experimental evaluation has been performed by using the Arduino UNO platform, and the results are reported in Table 3. In this setting, the ARTe runtime support demands about 8 KB with 16 periodic tasks (considering both program storage space and dynamic memory), while ARTe for Arduino Due demands 13 KB. This is caused by the different architectures and the implementations of the hardware-specific code. Also, remember that the two platforms rely on different versions of the

Table 3
Evaluation of ARTe memory footprints on the Arduino UNO platform. Since the Arduino UNO toolchain differentiates between *Program storage space* (first addend in sum) and *Global variables dynamic memory* (second addend in sum), this distinction is preserved in the table.

Number of loops	Footprint (bytes)	
	Arduino	ARTe
1	964 + 11 (2.98% + 0.54%)	5494 + 791 (17.03% + 38.62%)
2	958 + 11 (2.96% + 0.54%)	5552 + 853 (17.21% + 41.65%)
4	1006 + 11 (3.11% + 0.54%)	5668 + 977 (17.57% + 47.70%)
8	1080 + 11 (3.4% + 0.54%)	5900 + 1225 (18.29% + 59.81%)
16	1204 + 11 (3.73% + 0.54%)	6364 + 1721 (19.73% + 84.03%)

Table 4
Comparison of the worst-case profiled response times (pWCRT) for Arduino native and ARTe implementations of the case study application.

Loop	Period (ms)	pWCRT (ms)	
		ARTe	Loop sched
IMU	20	6.603	6.689
FIR	40	0.148	6.529
Servo	50	0.008	6.532
Web server	100	11.253	20.993
LED-1	1000	0.010	13.990
LED-2	2000	0.010	11.000
LED-3	3000	0.010	11.002

ERIKA kernel (ERIKA v2 for Arduino Due and ERIKA v3 for Arduino UNO).

As illustrated in Fig. 8 and detailed in Table 3, the footprint of an application increases with the number of tasks (i.e., ARTe loops). The test application is the same as the one considered in Fig. 7). Overall, ARTe tasks require a linearly increasing amount of memory, which is only slightly larger than the one required by the stock Arduino framework.

5.2. Case-study application 1

This section presents a case study developed as a test-bed to assess the effectiveness of the ARTe framework in a realistic application scenario. The hardware setup for the case study consists of an Arduino Due board equipped with an Ethernet shield, and connected to an inertial measurement unit (IMU) and a servo motor to realize radio-controlled applications. The Ethernet shield communicates with the Arduino board through the SPI bus while the IMU is connected to the board using the I2C bus. The servo motor is connected to a general-purpose output pin and is managed using a control signal generated by an MCU internal timer. The application cyclically (i) reads the angular displacement from the IMU sensor; (ii) filters the incoming displacement data using a FIR filter; (iii) actuates the servo motor to replicate the displacement angle; and (iv) updates a hosted web page that shows a graphical representation of the displacement angle.

Each of these activities is implemented as an ARTe loop. The software stack of the case study, which includes both the ARTe loops and the runtime support, is illustrated in Fig. 9.

The IMU loop periodically reads the displacement angle from the IMU using a software stack comprising the MPU6050 6-axis accelerometer/gyroscope, I2Cdev, and Wire libraries, and writes the acquired value to a global variable. The FIR loop reads such a variable and processes the samples coming from the IMU loop using a 40-tap low-pass filter, storing the result into another global variable. The servo loop reads such a variable, maps the displacement angle into the servo motor configuration space, and finally generates the control signal using the Servo library. The same variable is used by the webserver loop (WS loop) to notify the displacement of the IMU to the connected clients.

The webserver loop is a complex activity supporting up to four concurrent web clients. Internally, the webserver is built

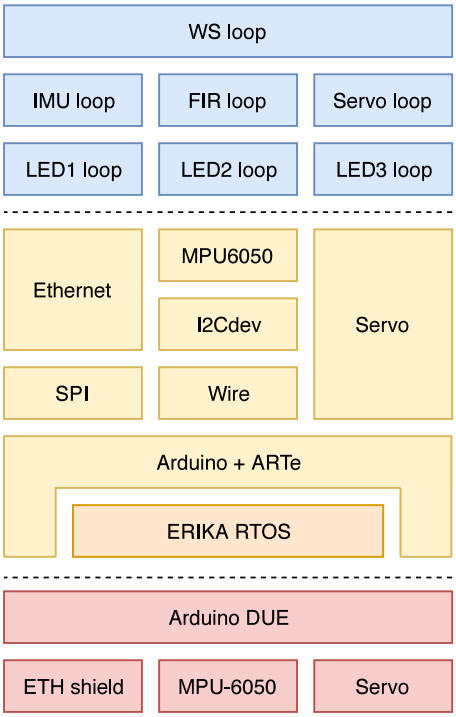


Fig. 9. Software stack for the case study application.

on top of the Ethernet library and uses the Server-Sent Events (SSE) technology, which is standardized as part of HTML5, to push updates to the clients using lightweight messages. Each iteration of the webserver loop (i) checks if a new client is available and then (ii) updates already connected clients using SSE messages. When a new connection request from a client is detected, the webserver replies by sending a web page that contains the JavaScript (JS) code used by the client to initiate the SSE data stream (through the EventSource JS interface) and visualize the data using an HTML5 Canvas element. Next, once the client sends the SSE events stream initialization request, the webserver registers the client as connected and acknowledges the request by starting sending SSE events.

The case-study application also includes three additional loops, each blinking a LED at a different rate, presenting to the user visual feedback of the running application. These loops also serve as overload indicators since they will be preempted by other tasks in case of overload, thus freezing the LEDs blinking activity.

The memory footprint of the Arduino native implementation is 52.8 kB bytes, while the memory footprint of the ARTe implementation is 55.2 kB bytes, corresponding to about 10% of the total flash memory in both cases. Table 4 describes the loops (tasks) that constitute the case study application and compares the longest response times (pWCRT) profiled for two different implementations. The first implementation has been built using

Table 5
Ventilator IO Hardware used in the case study. ARTe was used to test all IO peripherals, with the optimal configuration highlighted.

Part ID	Part Function	Quantity	Interface	Vendor	Part Serial
AF1	Airflow sensor	2	I2C	Honeywell	HAFUNH0300L4AXT
PS1	Pressure sensor	3	ADC (onboard)	Honeywell	150PAAB5
PS2	Pressure sensor	0/2	ADC (onboard)	Honeywell	015PAAB5
O21	O2 Sensor	0/2	ADC (external)	Maxtec	Max-23
O22	O2 Sensor	2	ADC (external)	Maxtec	Max-12C
O23	O2 Sensor	0/2	ADC (external)	Maxtec	Max-250ESF
PV1	Proportional Valve	0/2/4	PWM (12V drive)	Yong Chuang	YCLT21-35-1GBV-5B61B
PV2	Proportional Valve	0/2/4	PWM (12V drive)	Yong Chuang	YCLT21-2C-1GBV-5B61B
PV3	Proportional Valve	4	PWM (12V drive)	IQ valves	Tesla iQ
OI1	O2 Sensor IO	1	I2C	Texas Instruments	ADS1115
CM1	Control Modem	1	ADC (onboard)	C-19 Crisis Tech	TVCV19 Univ Controller
VD1	Valve driver	4	PWM (onboard)	C-19 Crisis Tech	TVCV19 AJak Controller
VD2	Valve driver	0/2/4	PWM (onboard)	Infineon Technologies	IRFZ44NPBF

the Arduino native approach described in Section 3.3, while the second implementation has been built using ARTe. In both cases, the response times have been measured during a 30-minute run with two clients connected to the webserver. It is worth observing how the preemptive scheduling available on ARTe can significantly improve the response times of high-priority loops, such as the FIR task. On the contrary, with the Arduino native approach, high-priority tasks (shorter period) can significantly be delayed by low-priority tasks (larger period).

5.3. Case-study application 2

This section presents a demonstration application of ARTe as part of a RMVS001 compliant makeshift ventilator platform. The platform was developed as treatment of last resort and has not been used to administer life support therapy.

During the COVID-SARS-2 pandemic of 2020, the global healthcare system has been under immense strain. Mechanical ventilators, which are used to treat acute COVID-19 cases, have been in short supply globally. Ventilating a COVID-19 patient requires safety-critical control of a sophisticated pneumatic system that drives the breathing cycle of a sedated patient using feedback from an array of pressure and airflow sensors. Existing designs have proven too costly for many healthcare systems or require parts that are not available locally where needed. This has created an urgent demand for the rapid development of safety-critical embedded systems adaptable for locally available sensors that can meet the performance requirements for life-saving ventilation treatments (Barrow et al., 2021).

Depending on vendor supplies, a wide range of suitable I/O peripherals could be integrated into a ventilator platform. However, it is difficult to adapt to available parts using conventional RTOS software stacks. Slow RTOS development cycles can become a bottleneck to meeting the needs of healthcare systems experiencing a crisis. Specialized driver software must be developed for each peripherals combination, limiting the possible configurations. By contrast, ARTe is particularly suited for the rapid development of ventilators. Community provided software drivers are readily available for every peripheral or are simple to add using templates and their companion tutorials (Bayle, 2013). By exploiting ARTe, it is possible to dramatically reduce the time needed to adapt to part shortages and create a hardware configuration that meets performance requirements and safety specifications.

Implementation details are presented for one of the possible ventilator designs whose custom set of I/O peripheral can quickly be modified according to hardware availability. The selected example configuration is highlighted in the table of ventilator parts evaluated using ARTe, shown in Table 5.

A block diagram of the example ventilator software stack is shown in Fig. 10.

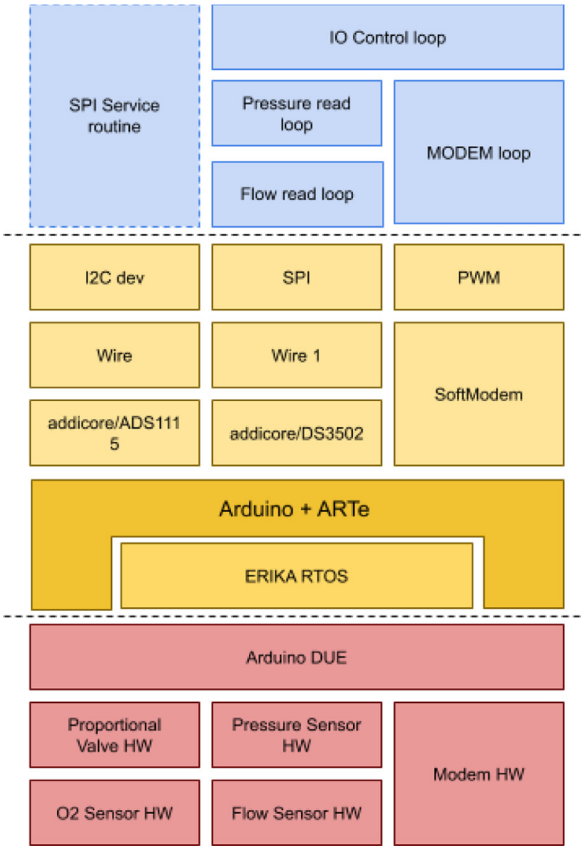


Fig. 10. Software stack for the example devices configuration of the ventilator.

Implementing ventilator I/O requires the coordination and scheduling of many peripheral management tasks. A high-level pseudo-code of the proposed ventilator application is given in Listing 14, illustrating how ARTe has been leveraged to achieve the goals of rapid development and timing predictability, crucial for patient safety. The application has two high-level activities to manage: the ventilation itself and a user interface to adjust the ventilation therapy. The mechanical ventilation is implemented as a PID closed control loop, whereby ARTe is used to set the discrete integration time step as precisely as possible to 20 ms (50 Hz). Our user interface is mostly offloaded to a secondary device and only a minimal set of PID variable setting and monitoring functionalities are implemented on the ARTe host board. The synchronous user interface task is implemented as an ARTe loop, called `loop_UI`, with a period of 50 ms for the most responsive user experience. The footprint of the application is 7.7

Listing 14: Pseudo-code of the ventilator application.

```
struct raw_data <t> {
    << ... >>
};

raw_data adcs;
raw_data pwms;
raw_data afs;
raw_data pss;

void setup() {
    init_adc0-1();
    init_modem();
    init_pwm0-4();
    init_af0-1();
    init_ps1-4();
}

void loop() {
    command = modem.rx();
    switch (command) {
        case read_sensor:
            modem.tx(JSON_buffer);
            break;
        case write_valves:
            pwms = modem.rx();
            break;
        << ... >>
    }
}

loop_pwm(20) {
    pwm.set(pwms);
}

loop_adc(20) {
    adcs = adc.read();
}

loop_UI(50) {
    JSON_buffer = data_to_JSON(adcs, pwms, afs, pss);
}

// Other tasks follow the same pattern
<< ... >>
```

KB, which correspond to 24% of the program storage space of an Arduino UNO platform. A description of the functionalities of the application reported in Listing 14 follows.

- `void setup()`, calls all transducer peripheral boot strapping functions such that they read *(monitor)* or write *(maintain)* global data structures, where:
  - ⋆ `void init_modem()` configures an analog modem peripheral used for remote control of the ventilator, including configuring an ISR triggered by the modem to allow for asynchronous remote control.
  - ⋆ `void init_adc0-1()` maps two ADC channels on a TWI ADC peripheral to a Oxygen level reading library that maintains the `adcs` buffer.
  - ⋆ `void init_pwm0-4()` initializes four on board PWM channels provided by the Arduino library and maps them to a gas valve control library that monitors the `pwms` buffer.
  - ⋆ `void init_af0-1()` maps two TWI bus mounted airflow sensors to an air flow reading library that maintains the `afs` buffer.
  - ⋆ `void init_ps1-4` initializes four on board ADC channels provided by the Arduino library and maps them to a gas pressure reading library that maintains the `pss` buffer.
- `void loop()`, polls the modem control library for ventilator commands. Commands are used to set parameters to the Proportional Integral Derivative (PID) respiratory control function and to interrogate the PID status to update the user on how the ventilator is performing.
  - ⋆ `modem.tx()` and `modem.rx()` calls a modem library function that sends data to and receives data from an external remote control software that updates the user interface displaying data.
- `void loop_pwm()`, maintains ventilator gas valve aperture settings as specified by the PID. This safety critical task uses an ARTe loop to guarantee that the valve aperture is maintained at a timely interval.
- `void loop_adc()`, monitors the O2 gas concentration reading at a fixed interval for PID. This timing critical task uses an ARTe loop to ensure stability of the PID integral that consumes this data.

The pseudo code illustrates how ARTe has provided the required benefit of timing predictability, shared task memory management, and task management along with the rapid development capability of the Arduino ecosystem. Specifically, *Timing predictability* leveraged in `loop_pwm` and `loop_adc` were essential for accurate respiratory control in the PID loop.

6. Conclusions

The Arduino framework has become a reference solution to learn the basic principles to program small embedded systems, quickly develop software interacting with real hardware, and fast prototype application to assess their advantages. However, the standard Arduino approach is limited to a single loop, thus limiting the exploitation of modern hardware platforms and the applicability to more complex fields, such as IoT. This paper presented ARTe, an extension to the Arduino framework able to seamlessly provide concurrent programming to the developer. ARTe is designed to handle transparently most of the issues arising from concurrency, leaving the application developer with a solution almost unaltered with respect to the original one. The internals of the extension has been explained to show the solutions applied to the various issues and its extensibility. A set of experiments has also been presented to show the simplicity and usability of the framework, evaluate the limited cost in terms of footprint, and show its applicability to a real-world use case. In particular, the mechanical ventilator use case highlights the rapid prototyping and extensibility advantages of the ARTe framework.

CRediT authorship contribution statement

Francesco Restuccia: Conceptualization, Methodology, Software, Writing – original draft. **Marco Pagani:** Investigation, Methodology, Software, Writing – original draft. **Agostino Mascitti:** Software, Writing – original draft. **Michael Barrow:** Software, Writing – original draft. **Mauro Marinoni:** Resources, Methodology, Supervision, Writing – review & editing. **Alessandro Biondi:** Methodology, Supervision, Writing – review & editing. **Giorgio Buttazzo:** Supervision, Writing – review & editing. **Ryan Kastner:** Resources, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The authors would like to thank Pasquale Buonocunto[4] for the vision and the fundamental contribution given to the first version of the ARTe framework.

References

Anon, 0000a. Erika enterprise RTOS kernel, http://erika.tuxfamily.org/drupal/.
Anon, 0000b. Arduino scheduler library, https://www.arduino.cc/en/Reference/Scheduler.
Anon, 0000c. ArduinoThreads, https://github.com/ivanseidel/ArduinoThread.
Anon, 0000d. The freertos kernel, https://www.freertos.org/.
Anon, 0000e. Nuttx real-time operating system, https://nuttx.apache.org/.
Anon, 0000f. Pre and post build hooks documentation, https://arduino.github.io/arduino-cli/latest/platform-specification/#pre-and-post-build-hooks-since-arduino-ide-165.
Anon, 0000g. OSEK/VDX Operating system specification 2.2.1., https://www.iso.org/standard/40079.html.
Anon, 0000h. The arduino official webpage, https://www.arduino.cc/.
Anon, 0000i. The doxygen parser official webpage, http://www.doxygen.nl/.
Anon, 0000j. The oracle documentation for the element interface, https://docs.oracle.com/javase/8/docs/api/javax/lang/model/element/Element.html.
Barragán, H., 2004. Wiring: Prototyping physical interaction design. Interaction Design Institute, Ivrea, Italy.
Barrow, M., Restuccia, F., Gobulukoglu, M., Rossi, E., Kastner, R., 2021. A remote control system for emergency ventilators during sars-cov-2. IEEE Embedded Systems Letters.
Barry, R., 0000. Arduino freertos, https://github.com/feilipu/Arduino_FreeRTOS_Library.
Bayle, J., 2013. C Programming for Arduino. Packt Publishing Ltd.
Bertogna, M., Fisher, N., Baruah, S., 1991. Resource-sharing servers for open environments. IEEE Trans. Ind. Inf. 5 (3), 202–220.
Biondi, A., Buttazzo, G., Bertogna, M., 2015. 2015. Supporting component-based development in partitioned multiprocessor real-time systems. in: proceedings of the 27th euromicro conference on real-time systems, ecrts 2015, lund, sweden.
Buonocunto, P., Biondi, A., Pagani, M., Marinoni, M., Buttazzo, G., 2016. ARTE: arduino real-time extension for programming multitasking applications. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing. ACM, pp. 1724–1731.
Buttazzo, G.C., 2011. Highest locker priority. In: Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. p. 212, Section 7.5.
Cheng, Z., Li, Y., West, R., 2015. Qduino: A multithreaded arduino system for embedded computing. In: 2015 IEEE Real-Time Systems Symposium. IEEE, pp. 261–272.
Kelemen, B., 0000. Softtimer pseudo multitasking solution, https://github.com/prampec/arduino-softtimer.
Liu, C., Layland, J., 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. J. Assoc. Comput. Mach. 20 (1), 46–61.
Marzario, L., Lipari, G., Balbastre, P., Crespo, A., 2004. IRIS: A New reclaiming algorithm for server-based real-time systems. In: Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium, Toronto, Canada.
Rivas, M.A., Tijero, H.P., 2019. Leveraging real-time and multitasking Ada capabilities to small microcontrollers. J. Syst. Archit. 94, 32–41.
Wang, Y., Saksena, M., 1999. Scheduling fixed-priority tasks with preemption threshold. In: Proc. of the 6th IEEE Int. Conference on Real-Time Computing Systems and Applications, RTCSA'99, Hong Kong, China.

[4] Formerly Scuola Superiore Sant'Anna, currently Buontech solutions Srl.