ARTE: Arduino Real-Time Extension for Programming Multitasking Applications

Pasquale Buonocunto, Alessandro Biondi, Marco Pagani, Mauro Marinoni, Giorgio Buttazzo
Scuola Superiore Sant'Anna, Pisa, Italy
Email: {p.buonocunto, alessandro.biondi, m.pagani, m.marinoni, giorgio.buttazzo}@sssup.it

***Abstract*—This paper presents an extension to the Arduino framework that introduces multitasking support and allows running multiple concurrent tasks in addition to the single execution cycle provided by the standard Arduino framework. The extension has been implemented through the ERIKA Enterprise open-source real-time kernel, while maintaining the simplicity of the programming paradigm typical of the Arduino framework. Furthermore, a support for resource sharing has also been integrated in the external Arduino libraries to guarantee mutual exclusion in such a multi-task environment.**

I. Introduction

In recent years, Arduino established as the most popular platform for rapid prototyping. It consists of a physical programmable embedded board (often referred to as the microcontroller) and an integrated development environment (IDE) that runs on a personal computer. Its widespread adoption is mainly related to the *simplicity* of the development and programming phases, that magnifies the Arduino user experience. In fact, 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. This is achieved through a minimal programming model, clean library APIs to access the hardware resources and the use of a simplified version of the C++ language, making easier to learn how to develop an Arduino application. In addition, there is a huge number of free third party libraries and code examples that allows a quick interaction with external devices, relieving the user from acquiring the knowledge for their usage. Unlike most others programmable boards, Arduino does not need external hardware (i.e., a programmer) to load the new code onto the board, since a USB cable is used for power supply, programming and communication. Finally, the Arduino board is designed with a common form factor that allows making connections to the microcontroller peripherals in a standard way.

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. Such a limitation prevents a full exploitation of the computing platform and in several situations forces the user to adopt tricky coding solutions to manage activities with different timing requirements within a single execution cycle.

Paper contributions. To overcome the limitation explained above, this paper presents the 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. Furthermore, a preliminary API support is proposed to addresses the problem of guaranteeing a mutually exclusive access to shared resources in such a multitasking environment. In addition to the classical Arduino programming model, consisting in a single main loop containing the code to be executed, ARTE allows the user to specify a number of different loops, each to be executed with a given desired period. A set of experiments is finally 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.

This work has been partially supported by Telecom Italia.

ARTE design goals. ARTE has been conceived according to the following design objectives:

- *Simplicity*: although different works [2], [12], [4] have been proposed to integrate a multitasking support in Arduino, we decided to pursue the "Arduino philosophy", thus making all the new programming features provided by ARTE *ease of use*. This has been achieved by designing the ARTE programming model *as similar as possible* to the original Arduino programming model, hence limiting the additional effort required to the user to implement concurrent applications.
- *Real-Time multitasking support*: Arduino is generally used to build embedded systems that interact with the environment through sensors, actuators and communication devices. For this reason, any delay introduced in the computational activities may affect the overall system performance [11]. Bounding the execution delays in all the concurrent activities programmed by the user is therefore crucial for ensuring a desired system performance.
- *Integration with standard Arduino Libraries*: The huge number of libraries provided with Arduino is one of the key strength points that determined its widespread use. To this purpose, ARTE has been conceived to enable the use of all existing Arduino libraries inside a multiprogrammed application. This has been achieved through a slight modification of the libraries to ensure data consistency under a multitasking execution.
- *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.

Paper structure. The remainder of this paper is organized as follows. Section II discusses the background including the related work. Section III presents our proposed solution. Section IV describes the internal structure of ARTE with details of its implementation. Section V reports an example of usage of ARTE, a case-study and some experimental results carried out to evaluate the performance of the proposed approach. Finally, Section VI states our conclusions and future work.

II. Background

A. Arduino Framework

The Arduino framework consists of a set of circuit boards (equipped with one or more microcontrollers) and an integrated development environment (IDE) used to develop the user application and program the board. Various Arduino boards have been released, having different computational capabilities and different I/O devices. For instance, the first board proposed, called Arduino UNO, is equipped with an ATmega328P microcontroller running at 16 MHz, which offers 14 digital I/O pins and 6 analog inputs. Another popular board is Arduino DUE, which outperforms Arduino UNO relying on a 32-bit Atmel SAM3X8E ARM Cortex-M3 microcontroller running at 84 MHz and offering a larger set of I/O pins.

The Arduino framework offers a very simple programming model structured in two basic functions, as reported in Listing 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. As stated in the introduction, such a programming model does not allow the user to specify a multitasking application.

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

void loop() {
  <code here>
}
```

Listing 1: Arduino programming model

To support users in writing applications, a lot of standard libraries are provided with the Arduino framework. Most of them are platform independent (making possible using the same API independently of the specific Arduino board) and have been designed to hide as much as possible all details related to the functionality offered by the used microncontroller. Being Arduino a mono-programming environment, all the libraries are designed not to be executed in concurrency.

B. Related Work

A few works have been proposed to support multitasking in the Arduino framework [2], [1], [3], all consisting in ad-hoc libraries not relying on a real-time operating system (RTOS) to manage concurrency. This approach mainly suffers from the following drawbacks:

- tasks are cyclically executed in a cooperative manner, making more difficult to achieve response time guarantees on application tasks;
- the resulting programming model is quite more complicated with respect to the original one offered by Arduino, requiring the user to specify initializations and explicit preemption points.

In addition, most of such extensions are not maintained and do not provide an explicit support for periodic activities.

Another proposed approach is FreeRTOS-Arduino [4], which is based on the FreeRTOS kernel, ported to be used as an Arduino library. Although this solution uses a RTOS to provide fixed-priority preemptive scheduling, it requires the user to be confident with concurrent programming and with the FreeRTOS API, which is far more complex than the standard Arduino programming model. Moreover, FreeRTOS is not a static RTOS, because not all kernel code and data structures can be tailored to the application at compile time. For this reason, the kernel is characterized by a larger footprint, memory, and runtime overhead due to dynamic objects management.

Qduino [12] is another solution that extends the Arduino framework with a real-time kernel and a new API for handling multiple concurrent control loops and mutual exclusion. However, Qduino supports only x86 platforms and aims at multicore CPUs, thus it is not compatible with the vast majority of Arduino boards, whose CPUs are much simpler and lack memory protection units. Furthermore, it requires the average Arduino user to acquire additional knowledge on real-time concurrent programming and the specific Qduino API. The Qduino API is quite similar to the one previously proposed by ARTE [10], but it requires the user to specify more parameters than the task period.

Like FreeRTOS-Arduino and Qduino, ARTE relies on an RTOS to implement concurrency, but unlike the previous approaches preserves the simplicity of the Arduino programming model, also providing a transparent integration of the large set of standard Arduino libraries. Moreover, ARTE supports the most common Arduino platforms (i.e., Arduino UNO and Arduino DUE).

To meet the design goals described in Section I, the underline RTOS should have a minimal impact in terms of footprint and runtime overhead and provide support for real-time task management. Among the existing open-source real-time kernels, ERIKA Enterprise [13] resulted to be the one that best fitted the requirements. Other open-source RTOSes, like FreeRTOS [5] and NuttX [6] could be used as well, but have been discarded for their larger footprints due to the dynamic management of their kernel objects. To provide the background needed to understand the ARTE approach, the next section briefly summarizes the features of the ERIKA kernel.

C. Erika Enterprise

ERIKA Enterprise (ERIKA for short) is an open-source real-time kernel [13] 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 [16] 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 fixed-priority scheduling with preemption thresholds [19], deadline scheduling, through the Earliest Deadline First (EDF) algorithm [14] and the Stack Resource Policy (SRP) [7], resource reservations (FRSH) [15], and hierarchical scheduling (HR) [8], [9]. 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, 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.

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 Approach

A. System description

The ARTE architecture proposed in this paper is illustrated in Figure 1. In ARTE, a user application exploits the ARTE framework to execute on the Arduino hardware platforms. In addition to the single loop present in the standard Arduino approach, the user can specify n *concurrent loops*. Loops can use the standard Arduino libraries, which are also enhanced to be executed on a multitasking environment, as presented in Section IV-C. Overall, the user application and the Arduino libraries rely on the Arduino framework, which has also been extended to support multiple concurrent loops by integrating it with the ERIKA Enterprise kernel (see Section IV-A). At the bottom of the ARTE architecture there are the Arduino hardware platforms. Today ARTE supports the most popular platforms: Arduino UNO and Arduino DUE.

B. The ARTE programming model

As explained in the introduction, the ARTE programming model has been designed to result as similar as possible to the original Arduino programming model. Each periodic loop defined by the user is specified as follows:

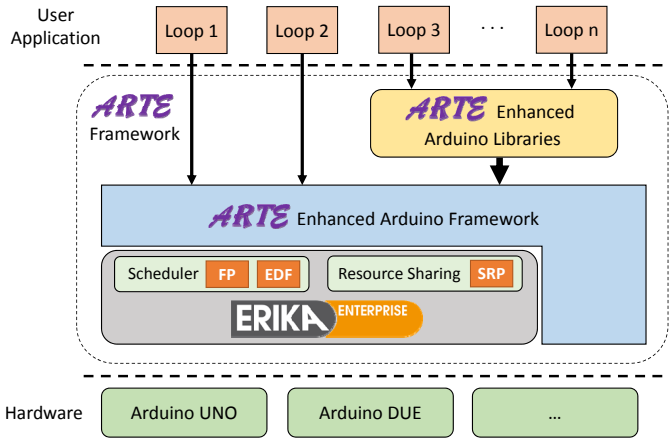

Figure 1: The ARTE architecture.

```
void loopi(int period) {
  <code here>
}
```

where $i = 1, 2, 3, \ldots$ 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.

IV. Internals

This section describes the internal structure of ARTE. The ARTE build process is first presented to explain how the user code (i.e., the sketch) is processed to obtain a multitasking ERIKA application and the binary executable file. Then the section illustrates how ARTE provides support for mutual exclusion inside the Arduino libraries.

A. ARTE build process

The whole ARTE build process flow is shown in Figure 2. The original Arduino framework includes a sketch processing phase, denoted as *Arduino processing*, which is implemented inside the Arduino IDE. The main part of the ARTE build process consists in extending the Arduino IDE with two additional processing phases (shown inside the dashed box): (i) *ARTE pre-processing*, which processes the sketch before the original Arduino processing, and (ii) *ARTE post-processing*, invoked after the original Arduino processing.

ARTE pre-processing. During this phase, the sketch is processed to extract the structure of the application, that is, the identification of the loops with their periods, in order to *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. In addition, the period of the loop is extracted and used to configure an *OSEK*

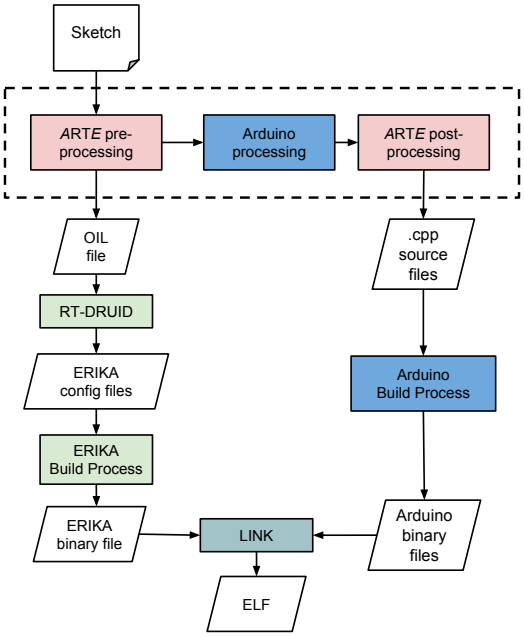

Figure 2: Build process

alarm, which is the OSEK standard mechanism conceived to trigger periodic activities. 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.

Arduino processing. This phase consists in the default Arduino transformation needed to produce a compiler-compatible code. In particular, the original sketch (in .pde or .ino formats) is converted to a standard .cpp file (i.e., C++ code); any additional files besides the main one are appended to it. Please refer to the official Arduino documentation for additional details on this phase.

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. In particular, all the ERIKA initialization functions are added in the `setup()` function (i.e., before any user-defined code is executed), and each OSEK alarm automatically generated in the ARTE pre-processing phase is activated. In this way, task activations will be completely transparent to the user.

Linking. As shown in Figure 2, 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. Note that, as described in Section II-C, this binary file is an RTOS image specifically configured for the user application needs that are automatically derived from the ARTE sketch. On the other side, the user code is built by means of the standard Arduino build process, enhanced to have the visibility of ERIKA C headers, so obtaining the object files of the user application. Finally, the LINK phase puts together the ERIKA binary with the object files that resulted from the Arduino build process, generating the final ELF binary file ready to be loaded into the microcontroller.

B. Mutual exclusion

Since the standard Arduino framework is designed to be single-threaded, not all of its code is thread-safe, including all the external third-party libraries. A simple way to keep them safe in a concurrent environment is to run all the tasks in non-preemptive way. This solution however, would introduce large blocking delays in the presence of loops with long execution times. On the other hand, a fully preemptive approach cannot guarantee data consistency on shared global data structures.

The typical solution adopted in modern RTOSs is to guarantee data consistency is to access global data structures through mutual exclusion semaphores (mutexes). The use of mutex semaphores, however, requires inserting specific lock/unlock primitives inside the code. In the context of real-time operating systems, several resource access protocols, as Priority Inheritance Protocol (PIP) [17], Priority Ceiling Protocol (PCP) [17], and Stack Resource Policy (SRP) [7] have been developed to bound the blocking delays caused by concurrent resource accesses.

An alternative solution is to adopt a limited preemptive approach [20], where preemption is disabled only inside specific regions of code (the critical sections). Note that fully preemptive and non-preemptive scheduling are two particular cases of such general approach.

A simple implementation of limited preemptive scheduling can be achieved by defining a single non-preemptive high-priority task, as done in MansOS [18]. This solution, however, is quite limiting because of the additional constraint on the programming model. In fact, all the critical sections must be inserted in the non-preemptive task, disallowing safe resource sharing among multiple tasks. This can be acceptable for a simple application scenario, like a small WSN application, but cannot be considered a general solution for a more complex embedded system application.

All the mechanisms discussed above are already implemented in ERIKA and can be used in ARTE. However, they require the user to insert specific RTOS primitives in the application code and have a deep understanding of the problems related to concurrency. Unfortunately, this is in contrast with the Arduino philosophy, which aims at *simplicity*.

To address mutual exclusion while maintaining the simplicity of the original Arduino framework, ARTE provides two primitives that allow first and third-party library developers to easily extend their library to guarantee safety in a multitask

```
void WiFiDrv::config(...)
{
  /* *** ARTE - begin critical section *** */
  arteLock();

  <SPI transaction>

  /* *** ARTE - end critical section *** */
  arteUnlock();
}
```

Listing 2: Modified Wifi shield library. File *wifi_drv.cpp*

environment. In addition, since the libraries support is crucial for using ARTE, all the Arduino standard libraries have been modified to be included as part of the ARTE framework.

The two primitives provided by ARTE to library developers are *arteLock()* and *arteUnlock()*, allowing the definition of critical sections. As an example, Listing 2 shows how a function of the WiFi shield library has been modified to include a critical section, protecting the transaction between Arduino and the WiFi shield over the SPI bus. In ARTE, mutual exclusion is implemented through a single shared resource, denoted as (*RES_SCHEDULER*), which is part of the OSEK standard. When the running task acquires a lock on that resource, the task becomes non-preemptive until such a resource is released.

Thanks to this extension, any Arduino library developer, even without any real-time system programming expertise, can easily extends his own library code to be safe for a multitask environment, without the need to declare and manage shared resources.

Since many Arduino libraries are built using a hierarchical structure (i.e., relying on other lower-level libraries), it is necessary to deal with nested critical sections. To cope with this problem, the ARTE mutual-exclusion support is designed in such a way that only the first *arteLock()* and the last *arteUnlock()* actually enters and exits the critical section. Furthermore, to avoid side effects, *arteUnlock()* does not have any effect if called from outside a critical section.

In order to be seamlessly integrated with the Arduino framwork, the API has been conceived to have no effect when the ARTE extension is disabled. In this way, the Arduino libraries, modified with the inclusion of critical sections, are still compatible with the standard Arduino framework when the ARTE extension is disabled. This choice facilitates the distribution of ARTE enhanced libraries as main-line distribution, having a common code for ARTE and non ARTE users.

C. Library implementation

As a default option, functions *arteLock()* and *arteUnlock()* are declared as weak aliases of the same "empty" function, thus implementing a null behavior. If ARTE extension is enabled, the weak symbols are overridden by strong symbols defined in the ERIKA image, whose source code is reported in Listing 3. In both functions, interrupt are disabled to avoid critical races on the variable `nesting_level_`, then the nesting level is checked and the OSEK API `GetResource` (or `ReleaseResource`) is used to disable preemption.

```
static const uint8_t max_nesting_level_ = 1;
static uint8_t nesting_level_ = 0;

void arteLock(void)
{
  EE_hal_disableIRQ();

  if (++nesting_level_ <= max_nesting_level_)
    GetResource(RES_SCHEDULER);

  EE_hal_enableIRQ();
}

void arteUnlock(void)
{
  EE_hal_disableIRQ();

  if (nesting_level_ > 0) {
    if (--nesting_level_ < max_nesting_level_)
      ReleaseResource(RES_SCHEDULER);
  }

  EE_hal_enableIRQ();
}

// For test purpose
uint8_t arteEnabled(void) { return 1; }
uint8_t lockNestingLevel(void) { return nesting_level_; }
```

Listing 3: ARTE primitives for mutual-exclusion.

V. Evaluation

This section presents some experiments carried out to evaluate the effectiveness of the proposed approach. An example of ARTE application is illustrated to compare it with a possible formulation that would be necessary using the classical Arduino programming model.

Furthermore, a complete case-study project has been developed with the purpose of testing the proposed extension in a "real world" scenario, where multiple extension devices and peripherals are attached to the Arduino board. This provides an interesting test bench for this work since the Arduino libraries, needed to control those additional devices, are required to work properly in a multitask environment.

A. Example

The selected example consists in a simple multi-rate led blinking application. In this example, the application is in charge of making three different leds blinking with different periods, equal to 3, 7, and 11 seconds, respectively.

Listing 4a shows the considered example implemented with the classical Arduino programming model. The single Arduino *loop()* contains a *delay* instruction that is responsible to define the time granularity of the loop. The argument passed to the delay function must be equal to the greatest common divisor (GCD) of the blinking periods (in this case 1 second). A variable *count* is used to keep track of the current multiple of the time granularity to determine which led has to blink.

On the other side, Listing 4b shows the same program formulated using the ARTE programming model. Using the proposed approach is it possible to specify three different loops, one for each led. The parameter indicated in the brackets of the loop is the period (in milliseconds) at which it has to be executed.

Although such a simple example can still be easily handled with the original Arduino framework, the situation can get

```
int led1 = 13;
int led2 = 14;
int led3 = 15;
int count = 0;

void loop() {
  if (count % 3 == 0)
    digitalToggle(led1);

  if (count % 7 == 0)
    digitalToggle(led2);

  if (count%11 == 0)
    digitalToggle(led3);

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

  count++;
  delay(1000);
}
```

(a) Arduino sketch

```
int led1 = 13;
int led2 = 14;
int led3 = 15;


void loop1(3000) {
  digitalToggle(led1);
}



void loop2(7000) {
  digitalToggle(led2);
}



void loop3(11000) {
  digitalToggle(led3);
}
```

(b) ARTE sketch

Listing 4: Example of multi-rate blinking leds sketches written using classic Arduino and ARTE.

worse with more complex applications, requiring a much higher programming effort to emulate a multithread behavior.

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 DUE platform. 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.

Listing 5 shows the OIL configuration generated by ARTE for the multi-rate blinking example. As the figure shows, an OSEK-task specification is provided for each loop defined in the ARTE sketch. In addition, an OSEK-alarm is associated to each task. Task priorities are implicitly assigned following the *rate-monotonic* policy (i.e., the lower the period, the higher the priority).

```
CPU m3 {
  OS EE {
    CPU_DATA = CORTEX_MX {
      MODEL = M3;
      APP_SRC = "ARTE-sketch.cpp";
      COMPILER_TYPE = GNU;
      MULTI_STACK = FALSE;
    };

    MCU_DATA = ATMEL_SAM3 {
      MODEL = SAM3xxx;
    };
    KERNEL_TYPE = BCC1;
  };

  COUNTER TaskCounter;

  TASK loop3 {
    PRIORITY = 0x03;
    SCHEDULE = FULL;
    STACK = SHARED;
  };

  ALARM Alarmloop3 {
    COUNTER = TaskCounter;
    ACTION = ACTIVATETASK { TASK = loop3; };
  };

  TASK loop2 {
    PRIORITY = 0x02;
    SCHEDULE = FULL;
    STACK = SHARED;
  };

  ALARM Alarmloop2 {
    COUNTER = TaskCounter;
    ACTION = ACTIVATETASK { TASK = loop2; };
  };

  TASK loop1 {
    PRIORITY = 0x01;
    SCHEDULE = FULL;
    STACK = SHARED;
  };

  ALARM Alarmloop1 {
    COUNTER = TaskCounter;
    ACTION = ACTIVATETASK { TASK = loop1; };
  };
};
```

Listing 5: The OIL configuration generated for the multi-rate led blinking example.

B. *Footprint*

Arduino boards are typically memory constrained platforms, therefore this section evaluates the impact of ARTE in terms of memory occupation.

Table 1 compares the memory footprint obtained when compiling an *empty* sketch with the classic Arduino framework, with the footprint of a sketch compiled with ARTE for a different number of empty loops. The reference platform for this evaluation is the Arduino DUE board. Note that, for the case of a single loop, only 1216 *bytes* of additional memory are required by ARTE, corresponding just to 0.23 percent of the total available memory. If further empty loops are declared, the memory footprint grows almost linearly with a rate of less than 50 bytes per loop. The plot in Figure 3 shows such a linear increase in memory footprint as a function of the number of loops.

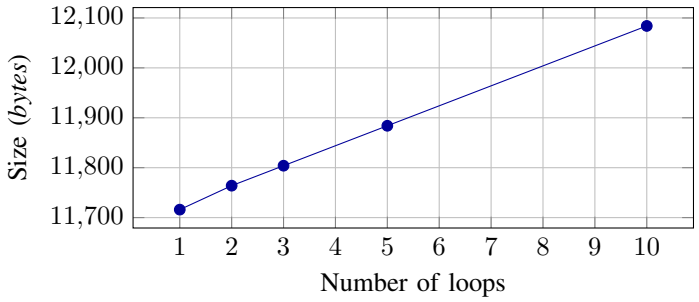

Figure 3: Footprint for an ARTE empty sketch.

Table 2 compares the footprints resulting from the two versions of the three-led example shown in Listing 4. In this case, the additional footprint of the ARTE version is only 1232 *bytes*.

Platform		Footprint	
		Size (bytes)	
Arduino		*Uno*	*Due*
		466	11,716
	Loops	Size (bytes)	
		Uno	*Due*
Arduino + ARTE	1	802	11,716
	2	822	11,764
	3	840	11,804
	5	878	11,884
	10	974	12,084

Table 1: ARTE and Arduino footprints for an empty Sketch.

Platform		Footprint
Arduino		Size (bytes)
		10,852
Arduino + ARTE	Loops	Size (bytes)
	3	12,084

Table 2: Footprint for the blinking LEDs demo.

C. A case-study

This section presents an evaluation of the ARTE approach on a more complex application developed on an *Arduino DUE* board. The system includes an inertial measurement unit (IMU), a servomotor and the Ethernet shield, an external hardware device that provides Ethernet connection to the Arduino board.

The goal of this application is to use a rotation angle measured by the inertial sensor to control the angular position of the servomotor, while hosting a web page that displays the values of the sensor and allows the user to enable or disable the actuation of the servo. To improve the responsiveness of the web user interface the orientation samples are streamed to the web browser though a *WebSocket* connection. Figure 4 shows an overview of the software and hardware layers involved in the demo.

From a software perspective, the application is structured into nine periodic tasks. Each task is defined through an ARTE loop. Three loops are dedicated to motion control: the first loop (*IMU Task*) periodically samples the orientation from the IMU; the second loop (*FIR Task*) performs a low-pass filtering on the collected samples trough a FIR filter; the third loop (*Servo Task*) maps the filtered orientation samples to the servo configuration space to control the servomotor.

The network functionality is realized by other three loops. The first loop (*WEB Task*) listens for HTTP requests and, once a request is received, it responds by sending a Web page containing

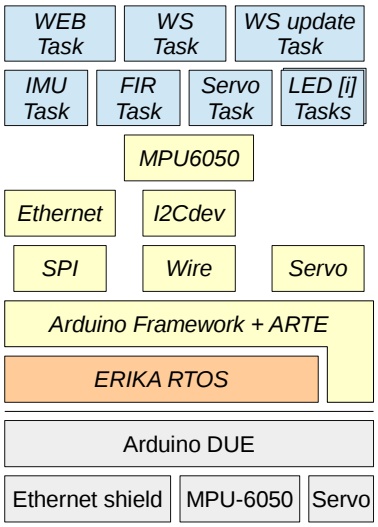

Figure 4: Case-study: software and hardware layers

the HTML elements and the JavaScript code that triggers the WebSocket connection. The request is then received and processed by the WebSocket loop (*WS Task*) that responds to the browser. Such a response concludes the handshake and allows establishing a WebSocket connection between the application and the browser. Once the WebSocket connection has been established, another task (*WS update Task*) periodically sends orientation samples trough the connection. The samples are received by the JavaScript code running on the browser and used to update an HTML5 element that dynamically shows the orientation to the user.

Finally, to provide the user with a visual feedback on the periodic behavior of the ARTE loops, the application includes three additional loops, each toggling a LED at a different rate.

Table 3 reports the tasks periods and the profiled worst-case execution times (pWCETs). As can be seen from Table 3, the worst-case processor load is around 89 percent. However, for every initialization phase occurring after each new HTTP connection, the processor load stabilizes around 25 percent.

Finally, the memory footprint of the whole application resulted to be equal to 104,032 *bytes*, occupying about 20 percent of the available flash memory on the SAM3 microcontroller of the Arduino DUE board.

Overall, this case study shows that ARTE is able to manage complex multitasking applications with a minimal runtime overhead and footprint.

VI. Conclusions

This paper presented a solution for integrating real-time multiprogramming capabilities in the Arduino framework in order to simplify the development of complex embedded systems characterized by multiple periodic tasks running at different rates. The integration has been implemented by exploiting the real-time features of the ERIKA Enterprise kernel, an OSEK-compliant open source kernel for small embedded platforms. The extended interface enables the user to easily specify multiple

Task	Period [ms]	pWCET [μs]
FIR	10	285
IMU	10	1804
Servo	10	11
Web Server	500	216193
WebSocket	500	116428
WebSocket update	50	1003
LED-1	1000	7
LED-2	2000	7
LED-3	3000	7

Table 3: Tasks periods and estimated worst-case execution times for the demo application.

loops to be executed at different rates while guaranteeing mutual exclusion among shared resources in a transparent way, thus preserving the simplicity of the classical Arduino framework.

Thanks to the proposed extension, the user can quickly and reliably exploit the full computational power of the platform without adopting tricky coding solutions, typically needed for managing activities with different timing requirements within a single execution cycle.

The impact of the implemented extension in terms of both memory footprint and runtime overhead has been evaluated and resulted to be affordable for most practical uses.

In the future, we plan to enhance ARTE by including a transparent mechanism to guarantee data consistency without requiring the user to specify critical sections, thus preserving the simplicity characterizing the Arduino programming model.

REFERENCES

[1] Arduino-compatible multi-threading library. https://github.com/jlamothe/mthread.

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

[3] Cooperative multithreading for microcontrollers, including arduino. https://code.google.com/p/threadkit.

[4] Freertos port for arduino. https://github.com/greiman/FreeRTOS-Arduino.

[5] Freertos: Quality rtos and embedded software. http://www.freertos.org.

[6] Nuttx real-time operating system. http://www.nuttx.org.

[7] T. P. Baker. Stack-based scheduling for realtime processes. *Real-Time Systems*, 3(1):67–99, April 1991.

[8] M. Bertogna, N. Fisher, and S. Baruah. Resource-sharing servers for open environments. *IEEE Transactions on Industrial Informatics*, 5(3):202–220, August 1991.

[9] A. Biondi, G. Buttazzo, and M. Bertogna. 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, July 8-10, 2015.

[10] P. Buonocunto, A. Biondi, and P. Lorefice. Real-time multitasking in arduino. In *WiP Session of the 2014 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*, Pisa, Italy, June 2014.

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

[12] Z. Cheng, Y. Li, and R. West. Qduino: A multithreaded arduino system for embedded computing. In *Proceedings of the 36th IEEE Real-Time Systems Symposium (RTSS 2015)*, San Antonio, Texas, December 2015.

[13] P. Gai, G. Lipari, L. Abeni, M. di Natale, and E. Bini. Architecture for a portable open source real-time kernel environment. In *Proceedings of the Second Real-Time Linux Workshop and Hand's on Real-Time Linux Tutorial*, November 2000.

[14] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[15] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo. 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, May 2004.

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

[17] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[18] G. Strazdins, A. Elsts, and L. Selavo. Mansos: Easy to use, portable and resource efficient operating system for networked embedded devices. In *Proc. of the 8th ACM Conference on Embedded Networked Sensor Systems*. ACM, 2010.

[19] Y. Wang and M. Saksena. 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, December 13-15, 1999.

[20] Y. Wu and M. Bertogna. Improving task responsiveness with limited preemptions. In *Proceedings of the 14th IEEE International Conference on Emerging Technologies & Factory Automation*, ETFA'09. IEEE Press.