CISTER
Research Centre in
Real-Time & Embedded
Computing Systems

Conference Paper

Evaluation of a low-cost multithreading approach solution for an embedded system based on Arduino with pseudo-threads

Juliana Paula Félix
Enio Filho*
Flávio Henrique Teles Vieira

*CISTER Research Centre

CISTER-TR-190611

2019/03/06

Evaluation of a low-cost multithreading approach solution for an embedded system based on Arduino with pseudo-threads

Juliana Paula Félix, Enio Filho*, Flávio Henrique Teles Vieira

*CISTER Research Centre

Polytechnic Institute of Porto (ISEP P.Porto)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: enpvf@isep.ipp.pt

https://www.cister-labs.pt

Abstract

Although projects using Arduino boards are becomingmore and more common due to their simplicity, low cost,and a variety of applications, Arduino boards consist of a simpleprocessor that does not allow the execution of threads. This paperpresents a study and evaluation of multithreading approacheson a single Arduino board. We present a group of existingsoftware approaches for dealing with concurrent actions onArduino. Among the solutions presented, we propose a case studyusing timed interrupts due to their simplicity. Although the casestudy provided requires dealing with many actions concurrently,including external actions, timed interrupts showed to be a robustsolution to the problem. Furthermore, the evaluated approachpresented great potential for being applied and implementedcommercially at low cost.

Evaluation of a low-cost multithreading approach solution for an embedded system based on Arduino with pseudo-threads

Juliana Paula Félix
Instituto de Informática
Universidade Federal de Goiás
Goiânia, Brazil
julianapaulafelix@inf.ufg.br

Enio Vasconcelos Filho
Cister Research Centre
Instituto Superior de Engenharia do Porto
Porto, Portugal
Instituto Federal de Goiás
Goiânia, Brazil
enpvf@isep.ipp.pt

Flávio Henrique Teles Vieira
Escola de Engenharia Elétrica, Mecânica e de Computação
Universidade Federal de Goiás
Goiânia, Brazil
flavio_vieira@ufg.br

***Abstract*—Although projects using Arduino boards are becoming more and more common due to their simplicity, low cost, and a variety of applications, Arduino boards consist of a simple processor that does not allow the execution of threads. This paper presents a study and evaluation of multithreading approaches on a single Arduino board. We present a group of existing software approaches for dealing with concurrent actions on Arduino. Among the solutions presented, we propose a case study using timed interrupts due to their simplicity. Although the case study provided requires dealing with many actions concurrently, including external actions, timed interrupts showed to be a robust solution to the problem. Furthermore, the evaluated approach presented great potential for being applied and implemented commercially at low cost.**

***Index Terms*—Embedded Systems, Multithreading, Arduino, Timed Interrupts**

I. Introduction

ARDUINO is an open-source electronic platform intended for anyone interested in creating interactive objects or environments [1]. Most Arduino boards consist of an Atmel 8-bit AVR microcontroller, varying its amount of flash memory, pins and features, which can be expanded by using shields. Shields are Arduino-compatible boards that can be plugged into the customarily supplied Arduino pin headers in order to provide other features, such as the addition of sensors, Ethernet, Global Positioning Systems (GPS), Liquid Crystal Displays (LCDs), and so much more.

Because of its simplicity and low cost, Arduino has been used for a wide range of applications, from simple projects such as blinking a set of LEDs, to more sophisticated ones, like controlling a robot arm [2] or a 3D printer [3]. Its applications can also be extended to commercial purposes [3–6]. Many universities are also using this device to introduce their students in the roles of programming, prototype and hardware development [7–9].

The Arduino is a very simple processor and has no operating system, which means it can run only one task at a time. In other words, it does not support threads. A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system [10]. The implementation of threads and processes differs between operating systems. In most of the cases, however, a thread is a component of a process. Multiple threads can exist within one process, running concurrently and sharing resources, such as memory, while different processes do not share these resources. In particular, a thread of a process shares its executable code and the values of its variables at any given time.

Standard Arduino API provides a programming model based on a basic structure compounded by two functions: `setup` and `loop`. The `setup` function runs only once and is followed by the `loop` function which repeats indefinitely. Thus, `setup` is usually used to prepare the program environment, while `loop` plays a role as the main program, performing a set of actions. We highlight that this programming model is a single thread model.

Although this programming model can be straightforward and easy to be programmed, it might present challenges since some actions such as IO (input/output) and state switching can conflict with each other. For instance, a simple blinking LED, usually implemented by using the function `delay`, holds the processor for the time the LED is kept on or off, while a card reader, which depends on external actions, requires the program to wait for data input so that another action can be followed. In this scenario, if the system forces the LED to blink on a regular time basis, it risks not to wait enough time to read a card and lose a card reading trial. On the other hand, if the system waits a considerable amount of time trying to ensure recognizing a card reading trial, the LED blinking might be delayed or happen asynchronously.

This hypothetical scenario is certainly not the hardest one to be solved. However, in a more complicated situation, engineers can end up opting for using more than one Arduino board to distribute tasks and avoid conflicts. Moreover, this is not just a waste of processing ability, but it also increases project cost. Therefore, many programmers have been working on providing solutions for multithreading on microcontrollers. In this paper, we provide a study on available techniques for running concurrent threads effectively on a single Arduino

board, and we analyze a case study of a situation where it occurs, applying one of the techniques described.

This paper is organized as follows: In Section II, we discuss techniques allowing an Arduino to run more than one thread at a time, Section III presents our case study and the solution we provided for it. In Section IV, we discuss the case study and efficiency of timed solutions and, finally, in Section V we present our concluding remarks.

II. Multithreading Approaches for the Arduino Platform

Due to its simplicity, Arduino does not support real Threads (parallel tasks). However, there are many scenarios which systems have to deal concurrently with synchronous tasks, such as loops for showing status or blocking procedures such as user inputs. In this sense, many solutions can be found in the literature, but most of them are based on pseudo-threads. This section provides a general description of the most common approaches and their respective details.

A. Interrupts

An interrupt is a signal that tells the processor to immediately stop what it is doing and handle some high priority processing [11]. Once all code attached to an interrupt is executed, the processor goes back to whatever task it was initially doing before the signal happened.

If one wants to ensure that a program always catches the press of a button, it would be very tricky to write a program to do anything else, since the program would need to constantly check the status of the pin attached to a button. If any other task has to be performed, which consumes time, the risk of missing a button press increases significantly.

By using interrupts, the system can react quickly and efficiently to important events that cannot be easily anticipated in software, such as monitoring user input. Moreover, it frees up the processor and allows it to do other tasks while waiting for an event to occur. Examples of tasks that might be benefited if interrupts are used include reading an I2C device [1], reading a rotary encoder [2], sending or receiving wireless data and, of course, monitoring user input.

On Arduino, this is provided by the function `interrupts()`, which allows certain priority tasks to happen in the background, while the microcontroller can get some other work done while not missing a user input, for example. This function is enabled by default, and works along with the function `attachInterrupt`, which takes three parameters [11].

The first parameter to `attachInterrupt` is an interrupt number in which we specify the actual digital pin that will be monitored. For example, if one wants to monitor pin 2, they should use `digitalPintToInterrupt(2)` as the first parameter to `attachInterrupt`.

The second parameter, usually referred to as an ISR (Interrupt Service Routine), is the function to be called when the interrupt occurs. This function must take no parameters, and they should return nothing. Generally, an ISR should be as short and fast as possible, and there should be no `delay()` call, so as to not interfere with other routines that the Arduino must execute. If a sketch[3] uses multiple ISRs, only one can run at a time. Other interrupts can be performed after the current one finishes, and their order or execution depends on the priority each one of them have.

The third parameter defines when the interrupt should be triggered. Arduino documentation [11] says there are four constants predefined as valid values: LOW to trigger the interrupt whenever the pin is low, CHANGE to trigger the interrupt whenever the pin changes value, RISING to trigger when the pin goes from low to high, and FALLING for when the pin goes from high to low.

Typically, global variables are used to pass data between an ISR and the main program and are usually declared as volatile in order to make sure they are updated correctly. Another essential point about interrupts on Arduino is that their number is limited, depending on the Arduino board. On UNO, for instance, there are only two digital pins that can be used for interrupts (pins 2 and 3). On Mega 2560, six pins can be used for interrupts (2, 3, 18, 19, 20, 21) and this number can get higher on other boards, such as DUE and 101, where all digital pins can be used, the latter restricting only the mode it can operate.

B. Timed interrupts

A timed interrupt, as the name suggests, is an interruption that is triggered when a specified time interval has been reached, similar to an alarm clock that rings when the time previously set comes. In a timed interruption, one can set a timer to trigger an interruption at precisely timed intervals. When an interval is reached, an alert can be emitted, a different part of code can be run, or a pin output can be changed, for example.

Just like external interrupts, timed interrupts run asynchronously or independently from the main program. Rather than running a loop or repeatedly calling `millis()`, one can let a timer do that work for them while the code does other things. For instance, if one wants to build an application that changes the status of a LED every 5s while it constantly does other things, they can set up the interrupt and turn on the timer, allowing the LED to blink perfectly on time, regardless of what else is being performed in the main program.

The AVR ATmega168 and ATmega328, which are used by Arduino UNO, Duemilanove, Mini and any of Sparkfun's Pro series, have three timers: Timer0, Timer1, and Timer2. The AVR ATmega1280 and ATmega2560 (found in the Arduino Mega variants) have three additional timers: Timer3, Timer4, and Timer5. Timer0 and Timer2 are 8-bit timers, meaning its

[1] I2C stands for "Inter Integrated Circuit" and an I2C device has a bidirectional two-wired serial bus which is used to transport the data between integrated circuits.

[2] An electro-mechanical device that converts the angular position or motion of a shaft or axle to analog or digital output signals.

[3] A sketch is the name that Arduino uses for a program. It is the unit of code that is uploaded to and run on an Arduino board.

count can record a maximum value of 255. Timer1, Timer3, Timer4, and Timer5 are 16-bit timers, with a maximum counter value of 65535. The details on how to use them can be found at ATmega328 [12] and ATmega2560 [13] datasheets, respectively.

C. Arduino Threads

ArduinoThreads is a library developed by Seidel [14] for managing the periodic execution of multiple tasks. The library promises to simplify programs that need to perform multiple periodic tasks, providing a way to let the program to run "pseudo-background" tasks. The user defines a Thread object for each of those tasks, then lets the library manage their scheduled execution.

There are, basically, three classes included in this Library: `Thread`, `ThreadController`, and `StaticThreadController` (both controllers inherit from `Thread`). `Thread` class is the basic class, which contains methods to set and run callbacks, check whether the thread should be run, and also create a unique `ThreadID` on the instantiation. `ThreadController` is responsible for holding multiple thread objects, also called as "a group of Threads", and it is used to perform `run` of every thread only when needed. `StaticThreadController` is a slightly faster and smaller version of `ThreadController`. It works in a way similar to `ThreadController`, but once constructed it cannot add or remove threads to run.

The heart of this approach is the `ThreadController`, which has a `run` method. Each thread is run sequentially, thus if `ThreadController.run()` method is called from the main `loop`, the program will run identically to a conventional `setup`/`loop` Arduino program. Therefore, programmers must create a timed interruption to call the `run` method in order to make threads compete with the `loop` function, but not interfere with each other.

ArduinoThreads' project is more of a threading emulation which organizes the programmer's code in an object-oriented programming way. A single timed interrupt calling a function can do the same task without any new thing to learn, but it can keep the code cleaner if the programmer has some code building preference or if the code grows too much.

D. AVR-OS Library

AVR-OS [15], developed by Cris Moos as an open source project, provides a very basic run-time that enables a program to deal with multiple threads on Arduino Uno, Mega and Mega 2560. Although its name has OS on it, which can be thought of as an operating system, it is actually a library, and therefore there is no need to replace the Arduino bootloader, being fully compatible to regular Arduino sketches.

As main characteristics, AVR-OS uses pre-emptive multitasking to switch tasks, and each task has its own stack that is restored when a task is resumed. It implements a simple thread scheduler based on an AVR timer, in order to provide ticks to switch between tasks.

E. Protothreads

Protothreads are a programming abstraction that provides a conditional blocking-wait statement intended to simplify event-driven programming for memory constrained embedded systems [16]. Protothreads can be seen as a combination of threads and events, having inherited the blocking-wait semantics from the former and the stacklessness and low memory overhead from the latter, using as low as two bytes per protothread.

While protothreads were originally created for memory-constrained embedded systems, it has also been proven to be useful as a general purpose library. It has been used in the Contiki operating system [17], and by many different third-party embedded developers [16]. Examples include a MPEG decoding module for Internet TV-boxes, wireless sensors, and embedded devices collecting data from charge-coupled devices.

Protothreads provide linear code execution for event-driven systems. It is highly portable, the library is implemented 100% in C and uses no architecture specific assembly code. It can be used with or without an underlying operating system to provide blocking event-handlers [18], providing a sequential flow of control without complex state machines or full multi-threading. Finally, it is available under a BSD-like open source license and can be downloaded at Adam Dunkels' website [18].

F. RTuinOS

RTuinOS is an event based Real-Time Operating System (RTOS) for the Arduino environment, created by Peter Vranken and it is available currently on [19], since moved from [20].

As mentioned earlier, a traditional Arduino sketch has two entry points: the function `setup`, which is the place to put the initialization code required to run the sketch, and function `loop`, which is periodically called. The frequency of looping is not deterministic but depends on the execution time of the code inside the loop.

Using RTuinOS, the two mentioned functions continue to exist and continue to have the same meaning. However, as part of the code initialization in `setup`, one may define a number of tasks having individual properties. The most relevant property of a task is a C code function, which becomes the so-called task function. Once entering the traditional Arduino `loop`, all of these task functions are executed in parallel to one another, as well as parallel to the repeated execution of function `loop`. We say that the function `loop` becomes the idle task of the RTOS.

A characteristic of RTuinOS is that the behavior of a task is not entirely predetermined at compile time. RTuinOS supports regular, time-controlled tasks as well as purely event controlled ones. Tasks can be preemptive or behave cooperatively. Task scheduling can be done using time slices and a round-robin pattern. Moreover, many of these modes can be mixed.

A task is not per se regular, its implementing code decides what happens, and this can be decided based on the context

or the situation. To achieve this flexibility, RTuinOS has an event controlled scheduler, where typical RTOS use cases are supported by providing according events, e.g. absolute-point-in-time-reached. If the task's code decides to always wait for the same absolute-point-in-time-reached event, then it becomes a regular task. However, in a situation-dependent scenario, the same task could decide to wait for an application sent event – and give up its regular behavior.

In many RTOS implementations, the primary characteristic of a task is determined at compile time. In RTuinOS, however, this is done partly at compile time and partly at runtime. RTuinOS is provided as a single source code file which should be compiled together with all the other code so that it becomes an RTuinOS application. In the most simple case, if we do not define any task, the application will strongly resemble a traditional sketch, whit a setup and a loop function. The former will run only once, in the beginning, and the latter will run repeatedly.

G. FreeRTOS port for Arduino

FreeRTOS is a Real-Time Operating System (RTOS) designed to be small and simple. Its kernel consists of only three C files and provides few routines in Assembly which needs to be rewritten to any new ported architecture. FreeRTOS provides methods for multiple threads or tasks, mutexes, semaphores, and software timers. Thread priorities are also supported. It also provides a tick-less mode for low power applications. Tick-less is an approach in which timer interrupts do not occur at regular intervals, but are only delivered as required. FreeRTOS applications can be entirely statically allocated.

There are some FreeRTOS ports for Arduino such as the ones created by Stevens [21] and Greiman [22]. While both projects provide a similar approach, Steven's project [21] is more recent and came to activity in late 2018. It can be used on many Arduino models such as Arduino UNO, Mega, MCU based, and others. In this approach, tasks are created on the `setup` function and the main `loop` function is free to run any other specific routine.

Although this approach looks interesting, it increases storage usage, and the programmer needs to pay attention to the required space size. As an example, an empty sketch that with FreeRTOS packages included compiled with Arduino IDE v1.6.9 on Windows 10, makes use of 21% more memory space on an Arduino Uno and 9% more on a Mega when compared to a genuine empty sketch. While this approach is very powerful, it also requires minimal knowledge of operating systems, threads and synchronization.

H. Qduino

Qduino [23] is an operating system and programming environment developed to run on multicore x86 platforms and Arduino-compatible devices such as Intel Galileo. It provides support for real-time multithreading extensions to the Arduino API, which promises to be easy to use and allows the creation of multithreaded sketches, as well as synchronization and communication between threads.

Furthermore, Qduino intends to provide real-time features that provide temporal isolation, between different threads and asynchronous system events such as device interrupts, an event handling framework that offers predictable event delivery for I/O handling in an Arduino sketch. One of its main offers is being a platform with smaller memory footprint and improved performance for Arduino sketches and backward compatibility that allows the execution of legacy Arduino sketches.

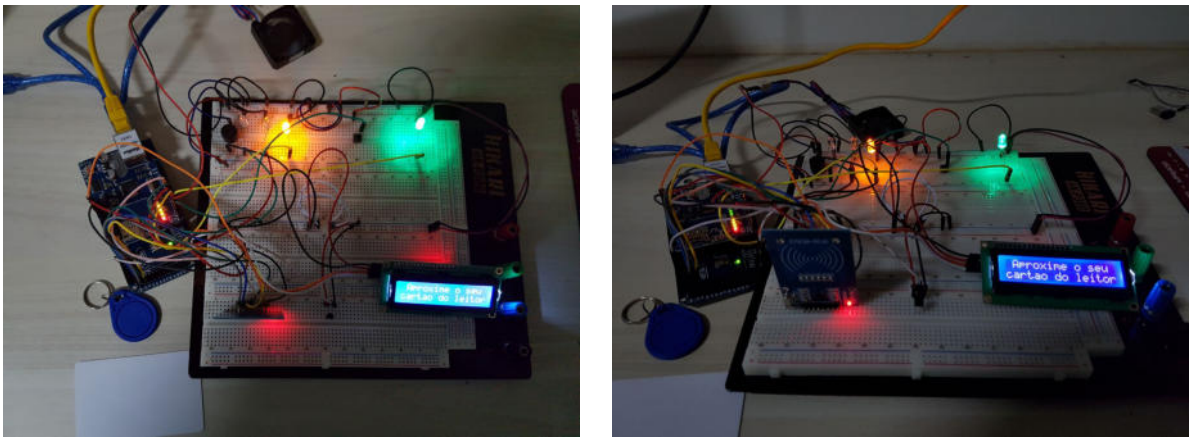
Although Qduino seems to be a robust approach for dealing with the multithreading issue, it is, unfortunately, not available for regular Arduino boards such as Arduino Uno or Arduino Mega.

III. Case Study based on Timed Interrupts

Due to the standard API using, low memory cost and simple implementation, we opted for analyzing the use of timed interrupts as an approach to allow multithreading on Arduino boards. We created a scenario for this study that requires dealing with external actions and input/output, reading a sensor and acting differently according to the read value, blinking of LEDs for a defined amount of time, showing information on a display, and continuously updating a web server based on information collected from the entire system. The system proposed and how its prototype was implemented is described next.

Our prototype provides an access control system based on RFID (radio frequency identification) [24], which simulates access granted or denied and gives feedback by turning on a green or red led for a defined amount of time while information about the access is being shown on an LCD display. To combine it with another external action to the system, we added a fan, which is turned on by pressing a push button, and whose speed is determined by the room temperature. The system also includes a web server, which can exhibit information about the place being secured and allows the system to be remotely monitored.

To build the prototype, we have used an Arduino Mega 2560 – a microcontroller board based on ATMega2560. It has 54 digital input/output pins, 16 analog inputs, 4 UARTs (hardware serial ports), a 16 MHz crystal oscillator, a USB connection, a power jack, an ICSP header, and a reset button [25]. Besides powering up all components used, the Arduino Mega is used to receive and process all data and make the required decisions. For the access control system, we used an MFRC522 RFID reader with an operating frequency of 13.56MHz and maximum data transfer rate of 10Mbit/s [26]. A set of green, red and yellow LEDs were used for simulating access granted, access denied and awaiting card to be read, respectively. A buzzer was also used to emit a different frequency sound according to the access that was granted or not. This information is also shown on an I2C 16x2 LCD [27]. Figure 1 presents a view of the actual prototype developed.



(a) Top View

(b) Perspective view.

Figure 1. System Prototype.

A. Proposed Scenario

In the system proposed, the fan was simulated with a 5V cooler, which was turned on/off with the press of a push button, and its speed was determined based on the temperature read by an LM35 sensor. Finally, the web server was possible thanks to an Ethernet shield, allowing the Arduino board to connect to the internet. The schematic of the prototype is shown in Figure 2.

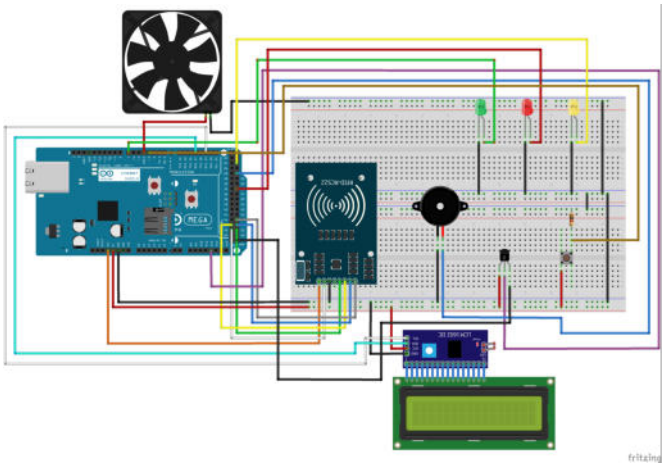


Figure 2. Circuit Diagram.

When the system starts, a yellow LED turns on, and the LCD display shows the message "Approximate your card". If an RFID tag is read, meaning a user has presented a card and wants to access the room being secured, the Arduino checks if the tag presented is assigned to a person who can access the room where the system is installed. If so, then the door of the room is open, letting the person walk in, and then closed shortly after, automatically. In our prototype, this is represented by a green led which is turned on for 10s, while the yellow LED is kept off. On the other hand, if a card is presented and the person assigned to that card is not in the list of authorized users or is not listed as a recognized user of the system, then the yellow LED turns off while a red LED is turned on for 5s.

When the time is up, the green LED turns off, and the yellow LED turns back on, indicating that the door has been closed and that a card needs to presented again in order to allow access to the room once again. Whenever a user presents a card to the RFID reader, the LCD display indicates whether the user whose card was read has been granted or denied access, and the buzzer emits a beep on different frequencies when the access has been granted or denied. At any moment, if a user accesses the web server, they will see the status of the door stated, i.e., "open" and "closed", the temperature of the room, and if the fan is on or not.

The fan installed at the room can be turned on or off whenever a button is pressed. When turned on, its speed is determined by the temperature of the room, collected periodically by the LM35 sensor. The value collected by the sensor is then passed to a map function that receives the minimum and maximum pre-set range of temperature carefully chosen to represent the minimum and maximum power, respectively. The temperature is continuously checked, powering the fan accordingly and updating the temperature value on the web server whenever a client is accessing the web server.

B. Scenario Evaluation

The scenario proposed combines many tasks which are user or state dependent. To deal with all required tasks, we provide a solution using interrupts and timed interrupts. In this section, we provide details on the solution provided and comment on its efficiency.

The solution proposed contains the `setup` function, the `loop` function, and a few other functions implemented to deal with each of the tasks required by the system. The `setup` starts the MFRC522, display, web server, interrupts, and defines the pins used as input or output.

The `loop` function is also straightforward. It is responsible for exhibiting information of "Approximate your card",

"Access granted" or "Access denied" on the LCD display and managing the changing of the status of the LEDs whenever a card is presented to the system, as well as performing the beep sound. All actions executed by `loop` are done based on the state changes of variables associated with the access control system.

To guarantee that a button press, meaning the user wants to turn on or off the fan, is never missed, we used an interrupt attached to the Arduino pin the button is connected. In this case, whenever a button press occurs, the system can detect it, and the state associated with the fan is changed to on/off. The task of effectively controlling the fan and its speed is handled by a timed function, which is executed by the system every 0.5s, no matter what other task is being handled in the system, and it acts based on the current state of the fan. The flow diagram of the function responsible for controlling the fan is illustrated in Figure 3.

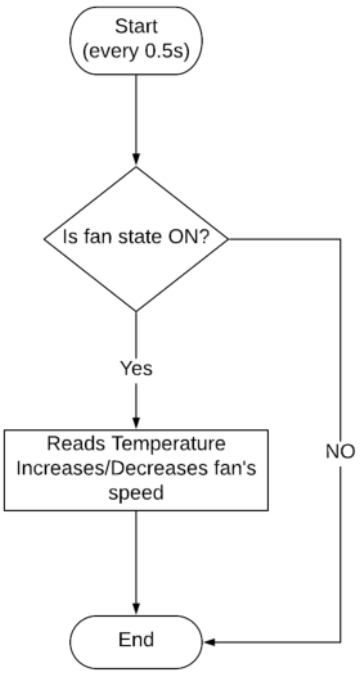


Figure 3. Fan flow diagram.

The access control is also implemented based on a timed interrupt. Every 0.5s, the system checks if a card is being presented and, in the affirmative case, the system verifies if its ID is associated with any card stored at the list of allowed users. The state variables associated with the RFID are then changed, and the loop function handles the process of opening or not a door (lightning a green or a red LED), based on the current state of those variables. Figure 4 shows the flow diagram for the critic part of the access control system.

Finally, the web server control is also controlled by a timed interrupt, triggered every 0.5s. Whenever the function associated with the web server control is called, it verifies if the server has an active client. If so, it exhibits all current

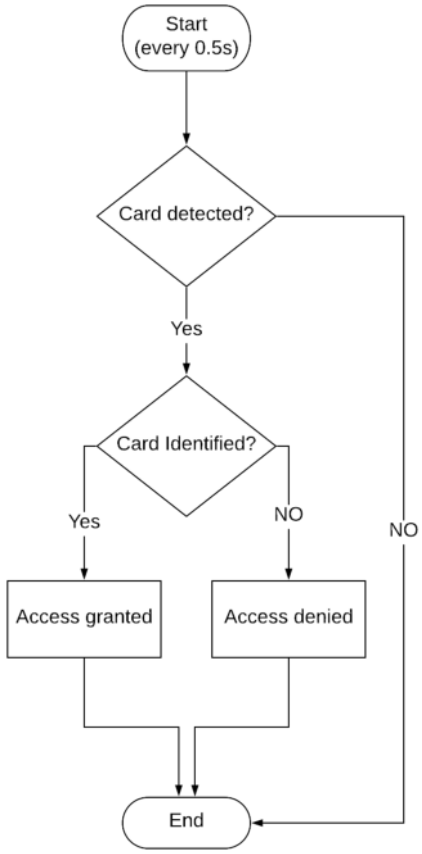


Figure 4. Access control flow diagram.

information about the system, including the status of the door (open/close/waiting for a card), the status of the fan (on/off), and the room's temperature. Since this function runs every 0.5s, the status presented on the web server is always up to date, with very little delay between the time a status was changed and the time it was first shown on the web page. Hence, whenever a card is presented to the system or the fan is turned on/off, the web server shows the card ID number that has been presented, shows if the door opened or remained closed, and shows the current state of the fan.

We empirically defined the time interval to 0.5s so that it does not affect the efficiency of the program nor does it waste processing time. The fan which has its speed defined according to the current temperature, for example, should not need to change its speed on a lower time basis, nor is the user interested in knowing the temperature of the room every millisecond.

IV. Discussion

In Section II, we described a variety of the available solutions for running or emulating multithreading on Arduino boards. We have seen that, while interrupts and timed interrupts are available by default on any Arduino, varying only the number of pins that can be used for such thing, all other

solutions described rely on external libraries. Furthermore, some of the solutions described are operating systems, such as Qduino, which is not available for the most common Arduino Boards, since it was planned to Intel Galileo Platform, or FreeRTOS, which although simple, consumes an enormous amount of memory. Moreover, no matter how simple a library can be, it requires an effort of reading and understanding how it works and how it can be useful for the desired application. We highlight that most of the solutions previously described, in essence, provide an abstraction of AVR timers to emulate threads.

We proposed a scenario in Section III that required dealing with many different external and user-dependent actions, and we evaluated the use of timed interrupts to deal with the proposed scenario. The solution proposed proved to be very simple to be implemented, it does not require any additional library as some of the approaches described in Section II, and it provided high efficiency in dealing with all the system's requirements, even though concurrent tasks have to be performed. We observed that the system could accurately identify when an authorized card was presented to the RFID reader, not missing a single card read trial, and thereafter allowing access for authorized users or denying it when an unauthorized card is used, without interrupting any ongoing task.

Moreover, the implemented web interface deserves highlighting, since it allows real-time verification of the status of the restricted environment where the system is installed and, as we show through the case presented for fan control, the system can be extended to display other information in the web system, such as the current ambient temperature and fan status, as described previously. An access history listing all individuals who had the entrance authorized, and even those who tried to enter the room and had the access denied, is another information that could be easily added to the web system with few modifications.

Since we used an Arduino Mega and it has 6 Timers, the proposed scenario fits adequately to deal with the system and user requirements. We note, however, that timed interrupts scheduled for the same interval can easily share routines, unless they perform blocking operations. Consequently, although only 6 Timers are available on Arduino Mega, much more than six time-dependent activities could be performed, as long as their routines are implemented together in the same timed interrupt.

V. Concluding Remarks

In this paper, we have gathered and described some software solutions for emulating threads on a single Arduino. From all software approaches presented, we chose timed interrupts, which can be implemented without the need of any additional library, and we analyzed its efficiency in dealing with many tasks concurrently. In order to evaluate how well timed interrupts could handle different actions, we proposed a case study of a real scenario that can be easily implemented and reproduced.

The case study proposed and evaluated required dealing with many different actions, including some user-dependent actions, such as input/output which could happen at unspecified time intervals. We proposed an access control system and implemented a prototype that required handling an RFID card, buttons, sensors, and synchronous blinking LEDs, besides continually updating the status exhibited on both an LCD and a web server.

Our solution based on timed interrupts requires no additional effort of adding a library since timed interruptions are part of the standard Arduino API. The solution presented to the case study proposed was able to handle all different tasks efficiently, even though blocking tasks and loop routines were being executed concurrently, while still maintaining the security and efficiency of the access control system. The software solution proposed shows that timed interrupts have the potential to be used in a wide range of applications, as well as could be commercially adopted.

Besides the simplicity of implementation, timed interrupts can help not only reduce the amount of Arduino boards in a project, consequently reducing the cost of a project, but it can also help to reduce the complexity of synchronization, connections among them, and so forth. We conclude that many areas that require the application of microcontrollers such as Arduino could benefit from this approach, since one could efficiently execute more functions with the inclusion of more components, such as GPS, sonar, gyroscope, and so on, all connected to a single Arduino. As a future work, we intend to compare the timed solution with some other available solutions, such as the ones described earlier in this paper, in order to adequately evaluate and compare the advantages and disadvantages of each solution.

Acknowledgments

The authors would like to thank CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brazil), CNPQ (Conselho Nacional de Desenvolvimento Científico e Tecnológico – Brazil), IFG (Instituto Federal de Goias – Brazil) for their support. We would also like to express gratitude towards the anonymous reviewers whose valuable comments and suggestions greatly improved the quality of the paper. This work was partially supported by National Funds, through FCT/MCTES (Portuguese Foundation for Science and Technology), within the CISTER Research Unit, (UID/CEC/04234);

References

[1] "Arduino homepage," https://www.arduino.cc/, 2019, accessed: 2019-01-20.

[2] Y. Pititeeraphab and M. Sangworasil, "Design and construction of system to control the movement of the robot arm," in *Biomedical Engineering International Conference (BMEiCON), 2015 8th.* IEEE, 2015, pp. 1–4.

[3] S. P. Deshmukh, M. S. Shewale, V. Suryawanshi, A. Manwani, V. K. Singh, R. Vhora, and M. Velapure, "Design and development of xyz scanner for 3d printing," in *Nascent Technologies in Engineering (ICNTE), 2017 International Conference on.* IEEE, 2017, pp. 1–5.

[4] Z. H. Soh, M. H. Ismail, F. H. Otthaman, M. K. Safie, M. A. Zukri, and S. A. Abdullah, "Development of automatic chicken feeder using arduino

uno," in *Electrical, Electronics and System Engineering (ICEESE), 2017 International Conference on*. IEEE, 2017, pp. 120–124.

[5] M. Kamisan, A. Aziz, W. Ahmad, and N. Khairudin, "Uitm campus bus tracking system using arduino based and smartphone application," in *Research and Development (SCOReD), 2017 IEEE 15th Student Conference on*. IEEE, 2017, pp. 137–141.

[6] J. Toji, Y. Iwata, and H. Ichihara, "Building quadrotors with arduino for indoor environments," in *Control Conference (ASCC), 2015 10th Asian*. IEEE, 2015, pp. 1–6.

[7] J. Sarik and I. Kimiss, "Qduino: A multithreaded arduino system for embedded computing," in *Frontiers in Education Conference (FIE), 2010, IEEE*. IEEE, 2010, pp. T3C–1–T3C–5.

[8] S. Jindarat and P. Wuttidittachotti, "Smart farm monitoring using raspberry pi and arduino," in *International Conference on Computer, Communications, and Control Technology (I4CT), 2015*. I4CT, 2015, pp. 284–288.

[9] A. Garrigos, D. Marroqui, J. Blanes, R. Gutierrez, I. Blanquer, and M. Canto, "Designing arduino electronic shields: Experiences from secondary and university courses," in *Global Engineering Education Conference (EDUCON), 2017, IEEE*. IEEE, 2017, pp. 934–937.

[10] A. S. Tanenbaum, *Modern operating system*. Pearson Education, Inc, 2009.

[11] "Arduino reference," https://www.arduino.cc/reference/en/, 2019, accessed: 2019-01-20.

[12] "Atmega328/p datasheet," http://ww1.microchip.com/downloads/en\/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf, 2016, accessed: 2019-01-20.

[13] "Atmel atmega 640 / v-1280 / v-1281 / v-2560 / v-2561 / v datasheet," http://ww1.microchip.com/downloads/en/\DeviceDoc/Atmel-2549-8-bit-AVR-Microcontroller-ATmega640-1280-1281-2560-2561_datasheet.pdf, 2014, accessed: 2019-01-20.

[14] I. Seidel, "Arduino thread," https://github.com/ivanseidel/ArduinoThread, 2013.

[15] C. Moos, "avr-os," https://github.com/chrismoos/avr-os, 2013.

[16] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying event-driven programming of memory-constrained embedded systems," in *Proceedings of the 4th international conference on Embedded networked sensor systems*. Acm, 2006, pp. 29–42.

[17] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki-a lightweight and flexible operating system for tiny networked sensors," in *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*. IEEE, 2004, pp. 455–462.

[18] "Protothreads," http://dunkels.com/adam/pt/index.html, 2019, accessed: 2019-01-20.

[19] P. Vranken, "Rtuinos," https://svn.code.sf.net/p/rtuinos/code/trunk/doc/\doxygen/html/index.html, 2017.

[20] ——, "Rtuinos," https://github.com/PeterVranken/RTuinOS, 2013.

[21] P. Stevens, "Arduino freertos library," https://github.com/feilipu/Arduino_FreeRTOS_Library, 2017.

[22] B. Greiman, "Freertos arduino library," https://github.com/greiman/FreeRTOS-Arduino, 2013.

[23] Z. Cheng, Y. Li, and R. West, "Qduino: A multithreaded arduino system for embedded computing," in *Real-Time Systems Symposium, 2015 IEEE*. IEEE, 2015, pp. 261–272.

[24] S. Ahuja and P. Potti, "An introduction to rfid technology." *Communications and Network*, vol. 2, no. 3, pp. 183–186, 2010.

[25] "Arduino store," https://store.arduino.cc/usa/arduino-mega-2560-rev3, 2019, accessed: 2019-01-20.

[26] "Mfrc522 datasheet," https://www.nxp.com/docs/en/data-sheet/MFRC522.pdf, 2016, accessed: 2019-01-20.

[27] "Datasheet i2c 1602 serial lcd module," https://opencircuit.nl/ProductInfo/1000061/I2C-LCD-interface.pdf, 2019, accessed: 2019-02-23.