Real-Time Multitasking in Arduino

Pasquale Buonocunto, Alessandro Biondi, Pietro Lorefice
Scuola Superiore Sant'Anna, Pisa, Italy
Email: {p.buonocunto, alessandro.biondi}@sssup.it
pietro.lorefice@gmail.com

Abstract— **This work-in-progress paper presents an extension to the Arduino framework that introduces multitasking support. This allows to have more concurrent tasks instead of the single cyclic execution provided by the standard Arduino framework. The extension is implemented by integrating in a seamless way the ERIKA open-source Real-Time OS, maintaining the simplicity of the programming paradigm typical of the Arduino framework.**

I. Introduction

In recent years, Arduino established as the most popular platform for rapid prototyping. Arduino consists of both a physical programmable circuit board (often referred to as a microcontroller) and an IDE (Integrated Development Environment) that runs on your computer. Its widespread adoption is highly related to its major strength: the simplicity during the programming and execution phases. In fact, unlike most others programmable circuit boards, Arduino does not need an external hardware (i.e., programmer) to load the new code onto the board, because an USB cable is sufficient for power supply, programming and communication. Additionally, the Arduino IDE uses a simplified version of the C++ language, making it easier to learn. This happens because Arduino provides a framework that allows the user to program a fully-working application without knowing any details of the underlying hardware. Moreover, there is a huge number of free third party libraries and code examples that permits to immediately use external devices, with the minimum effort to acquire the needed knowledge. Finally, the Arduino board is designed with a common form factor that allows to access the functions of the microcontroller into a more standardized way. However, although the Arduino programming model is simple and effective, it does not support concurrency and it is strongly limited to only a block of instructions that is cyclically repeated.

Contribution: We present RT-Arduino, an extension for the Arduino framework that introduces the real-time multitasking support. While the classical Arduino programming model consists of a single main-loop containing the code to be executed, RT-Arduino allows to specify a number of different loops, each one executed with a given frequency.

The main strength of the proposed approach is that the RT-Arduino programming model is compliant with the classical Arduino concept. RT-Arduino provides a very simple interface to specify a multitasked application, introducing few differences with respect to the original programming model. Each loop is mapped on an OSEK-task that are scheduled by the ERIKA Enterprise RTOS; the RTOS configuration is automatically generated limiting as much as possible the parameters that have to be specified by the user.

II. Related Work

For this kind of hardware which is typically used in different Arduino boards, a lot of operating systems are available.

Contiki is a lightweight operating system built around an event-driven kernel which optional preemptive multithreading [7]. Contiki has the proto-threads abstraction: ability to write thread-like programs with blocking calls on top of event-driven kernel [8]. As an alternative to cooperative proto-threads, Contiki provides preemptive threading model implemented as optional library;

TinyOS is an embedded operating system written in the nesC programming language as a set of cooperating tasks and processes [2]. A TinyOS component can post a task, which the OS will schedule to run later. Tasks are non-preemptive and run in FIFO order. This simple concurrency model is typically sufficient for I/O-centric applications, but reveals inadequate with CPU-intensive applications. This led to the development of a separate thread library called TOSThreads [10];

Mbed-Rtos is a platform for developing smart devices that are based on 32-bit ARM Cortex-M microcontrollers [1]. An optional library implementing the standardized CMSIS-RTX API provides RTOS functionality, such as thread management (define, create, and control), events (signals, message and mail) and resources management (mutex and semaphore);

There are also other multithreaded operating systems designed for wireless sensor networks that provides Unix-like abstractions. Examples are NuttX, ChibiOS/RT, LiteOS, Mantis and MansOS [14]. All of them are quite similar and provide preemptive scheduler for multithreaded applications. They differ especially in hardware abstraction architectures and available library and device drivers. Hsowever, each one has some drawback, including the require to learn some particular language or a new programming paradigm, and they do not provide real-time constraints, but only limited multiprogramming facilities. Moreover, the multiprogramming support is often included as an optional component, thus increasing the total footprint in terms both of ROM and RAM.

A. *Erika Enterprise RTOS*

Our solution is based on ERIKA Enterprise, a real-time kernel [9] which allows achieving high predictable timing behavior with a very small run-time overhead and memory footprint. ERIKA Enterprise is an innovative OSEK/VDX RTOS for small microcontrollers that includes highly predictable real-time kernel mechanisms and 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 scheduling algorithm, ERIKA Enterprise implements other scheduling algorithms such as Fixed Priority with preemption thresholds, Stack Resource Policy (SRP) [3], Earliest Deadline First (EDF) [11], resource reservations (FRSH) [12] and hierarchical scheduling (HR) [4], [5] which can be used to schedule tasks with real-time requirements. In particular, ERIKA supports periodic and aperiodic task scheduling according to fixed and dynamic priorities; interrupt handling for urgent peripherals operation (interrupts always preempt task execution); and time bounded resource sharing through the Immediate Priority Ceiling protocol [13], [6].

In Erika Enterprise, all the RTOS objects like tasks, alarms and resources are static (i.e., predefined at compilation time). To specify the objects composing a particular application, Erika Enterprise uses the OIL (OSEK Implementation Language) configuration files. OIL is a text description language defined as part of the OSEK/VDX standard, that is used for RTOS and application configuration. RT-Druid is an application provided with Erika Enterprise that is in charge of processing the OIL configuration in order to generate the specific Erika Enterprise code that defines the requested configuration. To allow writing, compiling, and analyzing applications in a comfortable environment, the RT-Druid plug-in for Eclipse is used. It also provides support for code generation and integration with static analysis tools.

III. System description

In the Arduino notation, the name *sketch* is used to denote a program for the Arduino framework. The sketch consists in a unit of code that is processed, compiled and then uploaded on the Arduino board. In the following we use ERIKA MAKE and Arduino MAKE to refer to the ERIKA Enterprise and Arduino MAKE build processes, respectively. The entire build flow process of RT-Arduino is showed in Figure 1. It involves several different phases. The first one takes the sketch code as input and give as output two files: the .OIL, which describes the ERIKA application, in terms of tasks used, resources, and other options, and the C file containing the whole code of the application, ready to be compiled by the Arduino MAKE phases. The OIL file, instead, is feeded to RT-DRUID that parses it and generate ERIKA configuration files (eecfg.h and eecfg.c) that contains all the data structures for the application described in the OIL file. At this point, the compilation of ERIKA can take place, and so the ERIKA MAKE phase generates a static library as output. Finally, the LINK phase put together this library with the object files resulted from the Arduino MAKE phase, thus building the binary file ready to be loaded into the microcontroller.

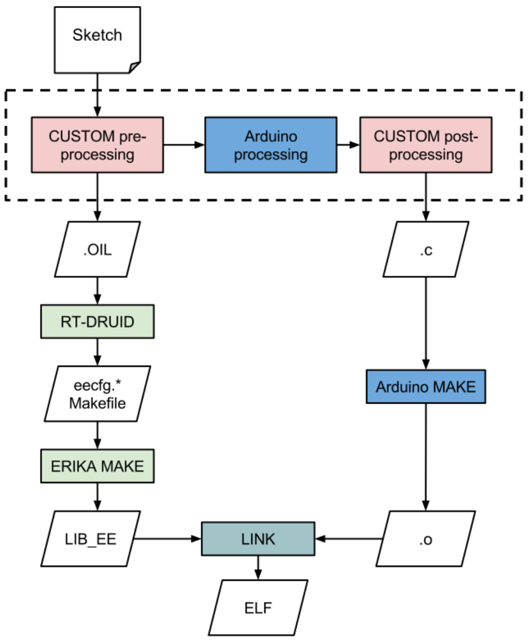

Figure 1: Build process

A. *Custom parser description*

CUSTOM pre-processing During this phase, the sketch file is preprocessed to extract the information useful for generating the OIL file. In particular, the sketch is parsed to identify the RT-Arduino loops. To make the experience for the user as seamless as possible, the constructs used to declare a task are very similar to the one used to declare a C function, in the form *void loopX(p)*, *X* and *p* being integer numbers representing the identifier of the loop and its period. For each loop, an OSEK-task configuration is generated and associated to the code inside the loop. In addition, the period of the loop is extracted in order to configure an OSEK alarm triggering the task activation, by using a global OSEK counter. This is done by parsing the sketch looking for the above declarations and adding them to a predefined OIL template, which also specifies the details (CPU, MCU, etc.) of the Arduino board on which the program will then be uploaded.

Arduino processing In this phase the code is subject to default Arduino transformations, needed to produce a compiler-compatible code. In particular, the original sketch (in .pde or .ino formats) is converted to a standard .cpp file, and any additional files beside the main one are appended to it. It will also contain any user-defined import of external libraries.

It is worth noting that at this stage the file is not yet ready to be compiled, since the task declarations, as seen in the following examples, are not standard C, and need to be modified in order to be recognized by both the Erika framework and the compiler.

CUSTOM post-processing The last phase is responsible to transform the sketch into an ERIKA Enterprise application, and to further modify the .cpp file produced in the previous step in order to make it compiler-compatible. In further detail, each task declaration needs to be modified into an Erika declaration, in the form *TASK(loopX)*, and along with any other element of the Erika framework, wrapped into an *extern "C"* declaration, since Arduino sketches are written using C++ while Erika is plain C. At this point, the file is ready to be compiled, but it still requires additions in order to make it fully functional. For this reason, the necessary framework initialization functions are added during the setup phase of the application (before any user-defined code is executed), and each automatically created alarm is started in order to eventually start the task execution. In this way, the activation of the tasks is completely transparent to the user.

IV. Examples

In this section we present an example of RT-Arduino application compared with the equivalent formulation that would be necessary using the classical Arduino programming model. The selected example consists in a very simple multi-rate led blinking application. In this example the application is in charge of make blinking three different leds, each one at a different frequency. The three leds Led1, Led2 and Led3 have to blink every 3s, 7s, and 11s, respectively.

```
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);

}
```

Figure 2: Example of multi-rate led blinking using the classical Arduino programming model.

Figure 2 shows the considered example implemented with the classical Arduino programming model. The single *loop()* of Arduino contains a *delay* instruction that is responsible to define the time granularity of the loop. The value passed as argument of the delay function has to be set to the MCD 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 in order to determine which led has to blink. On the other hand, Figure 3 shows the same program formulated using the RT-Arduino programming model. Using our 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.

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

void   loop1(3000) {

           digitalToggle(led1);
}

void   loop2(7000) {

           digitalToggle(led2);
}

void   loop3(11000) {

           digitalToggle(led3);
}
```

Figure 3: Example of multi-rate led blinking using RT-Arduino.

It is worth observing that this is a very simple example that can be handled with the original Arduino framework without excessive programming complexity. In practice, the situation can be more complex making hard the emulation of a multithreading behavior without using an RTOS. In addition, another main advantage of introducing the multithreading support in Arduino consists in enabling possible preemption among the loops. For example, it is not uncommon to have a loop *loop_large()* composed by a time-consuming block of code, which is executed with a large period. In this case, using the classical Arduino programming mode as in Figure 2, it is not possible to preempt the execution of *loop_large()* in favor of another loop with smaller period.

A. Internal processing

Figure 4 shows the OIL configuration generated by RT-Arduino for the multi-rate blinking example. As the figure shows, an OSEK-task specification is provided for each loop defined in the RT-Arduino sketch. In addition, an OSEK-alarm is associated to each task. Task priorities are implicitly assigned following a rate-monotonic order (i.e., the lower the period, the greater the priority).

V. Open issues

The Arduino framework is designed to be single-threaded, thus all of its code is not thread-safe, including all the external third-party libraries. There are two approaches to implement multiprogramming: the first is to have non-preemptive tasks, however this sometimes can be an unacceptable limitation and is prone to overrun by a badly-written task code that can cause low priority tasks to starve. On the other end, the second approach is to allow an arbitrary task preemption. However, preemption can lead to inconsistencies due to the lack of synchronization mechanisms in Arduino.

Different kinds of solutions have been proposed in existing literature and are widely used in modern RTOS, i.e. mutexes and different scheduling policy. Mutex semaphores have been introduced to handle mutual exclusive access to critical sections. The use of mutex semaphores requires the explicit specification of critical sections inside the code. In the

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

          MCU_DATA = ATMEL_SAM3 {
              MODEL = SAM3xxx;
          };

          KERNEL_TYPE = FP;
      };

      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; };
      };
```

Figure 4: The OIL configuration generated for the multi-rate led blinking example.

context of real-time operating systems, several resource sharing protocols such as Priority Inheritance Protocol (PIP) [13], Priority Ceiling Protocol (PCP) [13], and Stack Resource Policy (SRP) [3] are used to deal with mutual exclusion.

Another possible solution consists in the use of a different scheduling policy, like the limited preemptive scheduling [15]. In general, using this approach every task can be partitioned into multiple preemptive and non-preemptive sections, and it can be used to avoid critical races among tasks. A limit case of this approach consists in defining all the tasks to be non-preemptively executed.

Usually, it is common to have a simple implementation of the limited preemptive scheduling, by providing a single, non-preemptive, high-priority task, like as done in MansOS [14]. However, this is quite limiting because of the additional constraint for 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 simple scenario, like small WSN application, but can not be considered a general solution for a more complex embedded system application.

All of the previous discussed mechanisms are already implemented and usable in the Erika Enterprise kernel, that we integrated in Arduino. However, to be correctly used, all explicit synchronization mechanisms require the user to have a deep understanding of the problems related to multiprogramming and real-time systems, and also to correctly design and implement the whole application. This represents a major drawback since we aim to seamlessly integrating the mutual-exclusions functionalities already provided by the Erika Enterprise kernel into the Arduino framework.

VI. Conclusions

This WiP provides a first implementation of the integration between the Erika Enterprise kernel and the Arduino framework in order to provide a multiprogramming support to the well know Arduino framework.

References

[1] Embedded platform for ARM devices, 2014.

[2] Tinyos operating system for low-power wireless devices, 2014.

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

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

[5] A. Biondi, G. Buttazzo, and M. Bertogna. Schedulability analysis of hierarchical real-time systems under shared resources. Technical report, RETIS Lab, Scuola Superiore SantAnna, 06 2013.

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

[7] 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*.

[8] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proc. of the 4th International Conference on Embedded Networked Sensor Systems*. ACM, 2006.

[9] 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.

[10] K. Klues, C.-J. M. Liang, J. Paek, R. Musăloiu-E, P. Levis, A. Terzis, and R. Govindan. Tosthreads: Thread-safe and non-invasive preemption in tinyos. In *Proc. of the 7th ACM Conference on Embedded Networked Sensor Systems*. ACM, 2009.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.