Basanta-Val, Pablo, García-Valls, Marisol. A library for developing realtime and embedded applications in C. *Journal of Systems Architecture*, (2015), 61(5-6), 239-255.

DOI: https://doi.org/10.1016/j.sysarc.2015.03.003

Journal of Systems Architecture

A library for developing real-time and embedded applications in C

Pablo Basanta-Val, Marisol García-Valls

Universidad Carlos III de Madrid, Avda de la Universidad no 30, 28911 Leganés, Madrid, Spain

A B S T R A C T

Next generation applications will demand more cost effective programming abstractions to reduce increasing maintenance and development costs. In this context, the article explores the integration of an efficient programming language and high level real time programming abstractions. The resulting abstraction is called Embedded Cyber Physical C (ECP C) and it is useful for designing real time applica tions directly on C. The abstraction has its roots on the real time Java: one of the most modern program ming languages, which benefited from mature programming patterns previously developed for other languages. It also targets embedded processors running on limited hardware. ECP C takes the program ming abstractions described in real time Java and reflects them into a C application system, providing extensions for multi threading, resource sharing, memory management, external event, signaling, and memory access. It also reports on the performance results obtained in a set of infrastructures used to check ECP C, providing clues on the overhead introduced by these mechanisms on limited infrastructures.

1. Introduction

Next generation real time applications require abstractions to hide the complexity of the underlying infrastructure [31,39,16,9,56]. Among the multiple options included in the state of the art one identifies the use of MDA (model driven archi tecture) [16,51,36,50] and increasing the portability of different pieces of software with patterns as two complementary ways to reduce the complexity involved in the development of next generation real time systems. The use of these two methodologies comes with advantages in maintenance and development costs. However, they are also sources of inefficiency and overhead that have to be properly quantified as designing applications.

Recently, low cost hardware infrastructures such as Arduino [4,48,26], and Raspberry Pi [17,49] have appeared, guaranteeing universal access to replicable infrastructures. This type of plat forms represents an opportunity to embedded systems designers to developed with extraordinarily cheap, replicable, and accessible hardware (less than 5€ per node costs). Another common feature of this type of infrastructure is that they have a very reduced amount of resources available: for instance, some versions of Arduino offer only 32 Kbytes. Therefore, they demand specific adaptations of existing real time technology, like operating systems and pro gramming frameworks, to be efficiently deployed on them. Running heavy infrastructures like the RTSJ (The Real Time Specification for Java) [7] in these small computation infrastruc tures seems prohibitive.

This type of constraint is not new in the embedded community where the use of microcontrollers running minimalistic real time operating systems such as SimpleRTK [23] and ChibiOS [14] has a long tradition. However, the use of programming abstractions and generic real time programming models like those available in real time Java which include high level facilities for multithreading programming, scheduling policies, and schedulers are not properly supported by these ad hoc kernels, which are mainly focused on accessing resources.

To alleviate this lack, it is proposed a programming library for real time systems in the context of the C language, which is the main programming language in most embedded infrastructures. The approach takes its computational model from the program ming models described in RTSJ [27,10], which has benefited from the integration of high level programming facilities into its API. Also, it is partially inspired in other high integrity profiles such as Ravenscar Java [30] and SCJ (Safety Critical Java) [28] that produced computational subsets for different application domains.

Another advantage of C is the efficiency in the amount of mem ory and speed processor required to implement the application. It also has practical advantages in terms of compilers and runtimes

available and an important number of programming libraries written in C.

The definition of a programming framework for developing real time systems is not new in the context of the C language. This idea appeared in several works [32,32,51,15] that addressed lacks in concurrent support for real time systems, other low level facilities, and also in the real time POSIX standard [37].

The proposed approach, called ECP C (Embedded Cyber Physical C), addresses different limitations of the C programming model, borrowing architectonical elements defined in RTSJ. It also incorporates low level facilities required for a cyber physical system, in terms of "connectivity" via I/O interfaces [21,49,52].

ECP C imitates real time Java because it is one of the most modern real time programming languages. It is commonly accepted [10] that there are three real time programming languages: C, Ada, and Java. Among them, C is the most popular language for embedded systems but it lacks many high level programming facilities. It is also the most primitive and currently it lacks some building blocks required in a real time system such as a scheduler, threading, and real time synchronization protocols. Ada addressed many of limitations of C like the lack of a proper definition for types and it also standardized the programming language and the real time extensions. Real time Java was the last programming language to enter the scene, learning from previous C and Ada experience. Therefore, throughout this article ECP C is compared with real time Java.

The rest of the article describes the library (Sections 2 and 3), provides an application example developed with the library (Section 4), and evaluates the performance of the library (Section 5) on a small microcontroller running on a naked implementation and on top of an embedded operating system. Then, the article connects the approach with the related work (Section 6). Finally, Section 7 draws conclusions and introduces future work.

2. Enhancement areas and profiles in ECP-C

Before entering the description of the API of ECP C, this section lays the foundations necessary to understand ECP C from a high level perspective. This architecture is based on the concept of enhancement areas, borrowed from RTSJ, where each area refers to a set of facilities required to develop applications. ECP C also includes the possibility of having constrained profiles, each one targeting to a different end system.

ECP C targets at real time applications with support for fixed priority scheduling systems theory, typically used to support many real time applications [43,33]. Taking as a starting point the standard libraries included in a plain C tool chain like (AVR GCC, 2014), ECP C defines seven enhancement areas. Each enhancement area refers to a specific set of drawbacks or lacks in the support offered by the standard C library to a real time system developer.

The list of areas that potentially may be improved includes efficient support for multi threading abstractions; resource sharing policies in charge of controlling the access to shared information; efficient memory management algorithms with constrained memory; external events connecting ECP C to external events; asynchronous signaling in charge of providing notifications among threads via signals; and mechanisms to access physical memory positions with read write support. These facilities are analyzed in Section 3.

2.1. Enhancement areas in ECP C

2.1.1. Threading

This enhancement area refers to the lack of support in the C standard library to multithreading programming, which is an essential building block to develop real time systems. Currently, to develop concurrent programs in C one has to resort to other libraries such as POSIX threads. To address this drawback, ECP C offers a concurrency model by means of real time threads, which may be grouped into *periodic*, *sporadic*, and *aperiodic*, following the classical activation patterns, which may be optionally mapped to POSIX threads when they are available. In ECP C threads may be implicitly allocated, removed, and use at least 28 preemptive priorities as in RTSJ. This minimum set of priorities is required to guarantee that most of the scheduling algorithms may run properly [43,57 59].

2.1.2. Resource sharing

The threading model is complemented by a set of mechanisms to perform efficient thread synchronization, which are not available in the standard library. In some cases, they may be supported with POSIX mutexes and variable conditions. This type of support is required by many state of the art algorithms which use specific policies to reduce priority inversion when threads share information. In ECP C, threads may share information by using safe synchronization functions. The list of mechanisms supported in ECP C includes non blocking queues, semaphores, signals, and an atomic block primitive which is not included in RTSJ. ECP C does not support the `synchronized` statement of Java because of its inherent complexity.

2.1.3. Memory management

The memory allocation is also part of the set of enhancement areas and it is in charge of offering efficient memory management algorithms. The current support included in many tool chains may be insufficient when the application has to make an extensive use of the dynamic memory allocation and deallocation. The allocators included in a plain tool chain, like AVR GCC, were not specifically designed for real time performance and they require specific algorithms. Therefore ECP C offers facilities to intercept these mechanisms and use more efficient alternatives. ECP C allocates and deallocates memory blocks into two allocation contexts: its private stack, storing information in the record allocated in each function; and the heap via specific `malloc()` and `free()` statements, shared by all threads.

Optionally, ECP C enables other types of strategies that are similar to the scoped memory of RTSJ (with `malloc()`/`free()` and `free all()` functions) that may offer special behaviors like low fragmentation, raw memory allocation, and special memory allocators (see [6,35,29]). The definition of this type of manager is out of the scope of the article which only defines the interface that accesses these resources.

2.1.4. External events

Another generic limitation in a plain tool chain for C is the lack of generic mechanisms to connect external events, typically triggered by interrupts, to the application. To address these limitations properly, ECP C provides a way to connect external events (essentially, interrupts) to functions handled by the application.

2.1.5. Asynchronous thread signaling

Another limitation of a typical GCC tool chain, which is shared with a Java infrastructure, is the lack of mechanisms that enable transference of signals from one thread into another. This mechanism enables the development of richer applications. In the ECP C infrastructure, threads may communicate via asynchronous signals.

2.1.6. Raw memory access

Lastly, many embedded systems require direct access to physical memory positions to perform low level I/O communications.

ECP C standardizes this low level access required to read/write a certain memory address in an 8/16/32 bits format, following the RTSJ model.

2.2. Three profiles

ECP C defines profiles as a subset of a whole infrastructure called general ECP C profile. Each profile defines operational units in the infrastructure that are typically targeted to different specific application domains. Currently, it defines three profiles called micro, mini, and general profile.

2.2.1. Micro ECP C profile

This profile includes a minimal support for systems that do not require resource deallocation facilities. This type of profile is useful for embedded systems with an initialization followed by a mission phase running forever. It is also the simplest profile from the perspective of the implementation, because it does not require deallocation support.

2.2.2. Mini ECP C profile

This profile refers to a micro subset with support for resource deallocation. This type of profile is targeted to applications that require efficient and dynamic resource management.

2.2.3. General ECP C profile

This profile refers to a more general API, with an expressiveness power similar to the support included in the RTSJ specification. This profile targets at a platform similar to the whole RTSJ API but intended for C developers.

Each different profile interprets (i) portability; (ii) scalability; and (iii) extendibility in a different way. Currently, in ECP C portability is given by the C language and there are no specific mappings to real time POSIX, which may be developed to increase applications portability when running on top of an operating system. Scalability is not a major issue in the *micro* and *mini* profiles which run in very constrained execution environments; this concern is more related to the general ECP C. However, this type of support is more in tune with the general profile, which may require the use of pluggable schedulers. Mini and micro profiles, which do not demand that support, target to smaller infrastructures. Regarding extendibility, several approaches to increase and extend the functionality currently offered by ECP C. One may define new functions and additionally and anther should be to specify object attributes to configure the behavior of the underlying algorithms.

The rest of the article refers to the *mini* and *micro* subsets, only.

3. Micro and mini ECP-C profiles

This section covers the main functions designed for the mini and micro profiles and it also establishes analogy with the RTSJ whenever possible. In addition, the differences among RTSJ and ECP C are also properly explained.

3.1. Threading API

From the perspective of real time Java, the ECP C threading model included in the micro and mini profiles is a reduced version of the API included in RTSJ. This restriction reduces the flexibility of ECP C easing the programming model and reducing the efforts required to implement the infrastructure.

As RTSJ does, the proposed library supports three types of threads [12]: *periodic*, *sporadic*, and *aperiodic* (described in Listing 1, Listing 2, and Listing 3).

The main difference between the two ECP C profiles and RTSJ is that they do not define the scheduler as an entity included in the API. The RTSJ decouples the creation of thread from its nature: i.e., any RTSJ's `RealTimeThread` from its *periodic*, *sporadic* or *aperiodic* behavior and from the scheduling algorithms (stored in the `Scheduler` entity). The two ECP C profiles are simpler; they define one allocator function per type of thread (*periodic*, *sporadic* and *aperiodic*) only. This way, ECP C promotes de use of a simple scheduling system, based on off line analysis techniques that assign a priority to each task of the system.

Another relevant difference among RTSJ and ECP C is that ECP C defines (see Listing 1, Listing 2, and Listing 3) the size of the stack used in each thread. RTSJ does not include this type of support and by default the stacks are by 64 Kbytes, which are not acceptable for

```
THREAD *periodic_thread_create(
                          void (*entry)(void),
                          void *params_and_data,
                          long * stack,
                          long stack_size,
                          long priority,
                          long offset_us,
                          long period_us,
                          long max_deadline_us,
                          long max_cost_us,
                          void (*error_handler)(long errno));
```

Listing 1. Periodic thread creation (micro profile, and mini profile).

```
THREAD *sporadic_thread_create(
                          void (*entry)(void),
                          void *params_and_data,
                          long * stack,
                          long stack_size,
                          long priority,
                          long offset_us,
                          long period_us,
                          long max_deadline_us,
                          long max_cost_us,
                          long mit_policy,
                          long queue_length_max,
                          void (*error_handler)(long errno));
```

Listing 2. Sporadic thread allocation in ECP-C (micro profile, and mini profile).

small devices with reduced memory footprints (with less than 32 Kbytes).

The periodic thread allocation function (Listing 1) defines the function invoked in each activation (`entry`), a pointer to the application thread parameters (`params and data`) available for the thread, a pointer to the stack (`stack`), the size of the stack (`stack size`), the priority of the thread (`priority`), an initial offset with respect to the global clock of the system (`offset`) in microseconds, and a minimum period (`period us`) also in microseconds. It also allows the definition of a deadline (`max deadline us`) which is checked at runtime and a maximum cost per invocation (`max cost us`) in microseconds. In RTSJ, this type of information is offered in a more generic object oriented programming model, which may be extended by means of generic scheduling, release, memory, and processing group parameters. Therefore, ECP C constrains the programming model of RTSJ to a more specific set of scenarios.

Optionally, developers may define an error handler (`error handler`) invoked in cost overruns and as a thread receives a signal from another thread (see Section III E). This functionality is executed in the signaled thread who should take into account this extra cost as a part of its execution time. In RTSJ, the thread may use an event handler that runs in another thread to handle this signal, while the micro and mini profiles offer a more rudimentary workaround. In ECP C, the worst case execution time of the handler is added to the cost of the signaled thread. This is another difference with RTSJ, where these handlers are asynchronously executed in other threads. ECP C provides a simpler mechanism which does not require another thread.

There is no support for the execution budget enforcement technology in ECP C, which is also an optional feature in RTSJ [54]. RTSJ allows the specification of a budget via `ProcessingGroupParameters` that allow assigning a periodic budget to a group of tasks. RTSJ does not specify the type of server used to enforce this basic policy; it only describes its period (T), a cost (C), a priority (P), and a deadline (D).

The sporadic thread (see Listing 2) resembles the periodic thread: there is a function, which is invoked in its activation, and a set of initialization parameters. The main difference between both models is within the activation pattern. For sporadic threads, it does not depend on the system clock, but on an external signal, typically coming from another concurrent entity and triggered via a `thread signal` function. This queue of external events is controlled via `SAVE`, `EXCEPT`, `IGNORE`, and `REPLACE` policies (see Listing 6) taken from the RTSJ specification. These policies are useful to: (1) control the minimum inter arrival (`max period us`) among activations; and (2) to decide what to do when the maximum number of events (`queue length max`) is reached. In the initialization, the programmer creates a mask for these policies (see example in Section 4). The meaning of these policies is the same in RTSJ specification and in ECP C.

In both cases, the motivation is to control the behavior of the events received in the sporadic thread through asynchronous signals, sent via `thread signal`. These asynchronous signals trigger the activation of sporadic tasks from another thread.

Lastly, the aperiodic thread refers to threads that cannot offer bounds on a minimum inter arrival time (Listing 3). In essence, aperiodic threads do not define a minimum inter arrival time among two activations. As in the sporadic case, the activation of the task is triggered by a `thread signal` function from an external thread. Likewise, the `aperiodic` thread includes a maximum number of events which are pending to be processed. All these policies (i.e. `OVER SAVE`, `OVER EXCEPT`, `OVER IGNORE`, and/or `OVER REPLACE`) follow the behaviors prescribed in RTSJ.

Following the RTSJ's model, ECP C aperiodic activities may define a maximum deadline and cost for applications. If the cost and deadline are unbounded, they should be set to the maximum deadline of the system (i.e. `MAX INT`).

Threads may be removed from the system by using a `thread remove` function. This option is only available in the mini profile to avoid burdens related to dynamic task allocation (i.e., memory fragmentation, and extra run time overhead due to deallocation costs).

In addition, there are general functions (see Listing 4) included to access to the information of the thread, to sleep a thread for a certain amount of time (`sleep`), to force a scheduling event (`yield`), and to access the global clock of the system (`get time` and `get error in time`). These functions help the programmer access to system information; RTSJ behaves in the same way. ECP C also includes functions for starting the scheduling subsystem (`threads subsystem start`), and signaling (`thread signal`) sporadic and aperiodic tasks.

In a real time application, `threads subsystem start` defines the beginning of the application and it has no equivalent in RTSJ. However, other embedded programming frameworks include this functionality to mean the beginning of the scheduling subsystem.

The error handler (Listing 4) of each thread predefines a set of codes used to deal with overruns (`0x03`), memory exhaustion (`0x02`), deadline (`0x01`) and period (`0x00`) misses. If the handle function is invoked with a `0x04` code, then the thread has been signaled for another thread with application specific purposes. The remaining defined codes (i.e., `SAVE`, `IGNORE`, `REPLACE` and `EXCEPT`) are for sporadic, and aperiodic threads to configure their input queue of pending activations. Lastly, there is also a specific symbol (`PIP POLICY`) to describe a PIP (Priority Inheritance Protocol) policy.

3.2. Communication and synchronization

The library offers basic mechanisms to share data among the threads of an application. The list of high level mechanisms includes non blocking queues and real time semaphores. In addition, it also offers a mechanism to perform atomic blocks executed without interference from other tasks.

```
THREAD* aperiodic_thread_create(
                            void (*entry)(void),
                            void *params_and_data,
                            int* stack,
                            long stack_size,
                            long priority,
                            long offset_us,
                            long max_deadline_us,
                            long max_cost_us,
                            long over_policy,
                            long queue_length_max,
                            void (*error_handler)(long errno));
```

Listing 3. Aperiodic thread allocation in ECP-C (micro profile, and mini profile).

```
//Thread management
int    thread_remove(THREAD* who); //**
void*  thread_get_params();
void   thread_sleep(long time_in_us);
void   thread_yield();
long   thread_get_time();
long   thread_get_error_in_time();
void   threads_subsystem_start();
int    thread_signal(THREAD* th); //async signal for sporadic threads

//Handlers
ECP_FRAMEWORK_ATS_SIGNAL                    0x04
ECP_FRAMEWORK_COST_OVERUN                   0x03
ECP_FRAMEWORK_MEMORY_EXHAUSTED              0x02
ECP_FRAMEWORK_DEADLINE_MISS                 0x01
ECP_FRAMEWORK_PERIOD_MISS                   0x00

//Policies used in event queues
ECP_FRAMEWORK_SAVE_POLICY                   0x01
ECP_FRAMEWORK_EXCEPT_POLICY                 0x02
ECP_FRAMEWORK_IGNORE_POLICY                 0x03
ECP_FRAMEWORK_REPLACE_POLICY                0x04
ECP_FRAMEWORK_OVER_SAVE_POLICY              0x10
ECP_FRAMEWORK_OVER_EXCEPT_POLICY            0x20
ECP_FRAMEWORK_OVER_IGNORE_POLICY            0x30
ECP_FRAMEWORK_OVER_REPLACE_POLICY           0x40

// PIP policy symbols
ECP_FRAMEWORK_PIP_POLICY                    -1
```

Listing 4. Main functions related to the threading API (micro and mini profiles). It also includes the main symbols defined in ECP-C (**=available in the mini profile only).

```
//Thread queuing mechanisms
QUEUE *       thread_queue_create(size_t howmany);
void          thread_queue_delete(QUEUE * queue); //**
void*         thread_queue_nb_get_element(QUEUE* queue);
int           thread_queue_nb_set_element(QUEUE* queue, void* element);
int           thread_queue_stored(QUEUE* queue);
int           thread_queue_size(QUEUE* queue);

//Semaphore
SEMAPHORE*    thread_semaphore_create(int initialization, int PCE);
//PCE=-1 -> PIP and PCE>=0 refers to initialization of the ceiling
void          thread_semaphore_acquire(SEMAPHORE* sem);
void          thread_semaphore_release(SEMAPHORE* sem);
void          thread_semaphore_free(SEMAPHORE* sem); //**

//Atomic blocks
ECP_FRAMEWORK_ATOMIC_SECTION
```

Listing 5. Controlling shared resources in ECP-C (micro profile and mini profile). (**= available in the mini profile only).

Regarding queues, ECP C (Listing 5) allows the programmer to create the queue (`queue create`), to remove it (`queue delete`), to get (`get element`) and store (`set element`) elements, and to access the number of stored elements (`queue stored`) and its size (`queue size`). Queue deallocation is only feasible in the mini subset (via `queue delete`). The basic model for queues is fully non blocking in get/set functions. It returns `-1` as the queue is full in a write operation, and `NULL` in a read operation as the queue is empty. This model is similar to the non blocking queues included in RTSJ that approach the problem in a similar way.

In ECP C `semaphore acquire()` and `semaphore release()` functions support semaphores. Semaphores are initialized with a PCE (Priority Ceiling Emulation) policy as in RTSJ and with a PIP policy when PCE is `-1`.

RTSJ offers the `synchronized` statement as the basic element to control the concurrency. However, this mechanism, which is the default synchronization statement in Java, is quite complex to be the default concurrency control mechanism for ECP C, which opts for a simpler approach.

There is also a macro to mark a piece of code as an atomic piece of code. The idea behind is that the code is not to be interrupted for its execution, so the code contained within this macro has to consist of a reduced number of statements. This macro is useful for low level access to specific hardware and for simple read/write operations. Currently, RTSJ does not offer an atomic section in its API but it could benefit from the inclusion of this functionality within its API. In ECP C implementations, a typical implementation for this facility uses enable and disable interrupts to support atomicity in the macro.

3.3. Memory management API

ECP C (Listing 6) supports dynamic memory allocation via `malloc` and `free` statements that allocate and deallocate memory. With the `malloc` and `free` statements defined in ECP C, the application gains low level control on the type of allocation carried out. As in the previous case, deallocation facilities are not available in the micro profile because this profile does not deallocate resources.

These functions may implement efficient memory allocators like [6,35].

3.4. External events API

To standardize low level access, the library includes functions that attach and detach interrupt handlers from to the system. The approach taken in the mini profile is to offer a simple support to this functionality. This minimum functionality allows both operations: (i) interrupt attachment, and (ii) interrupt detachment, while the micro profile includes interrupt attachment only. When attached, each interrupt has a `mode` that refers to rising (0), falling (1), or changes (2) in the source of the interrupt. This type of support is sufficient to attach low level interrupts to well known ports.

RTSJ offers an interaction with the external ecosystem with specific POSIX signals mapped to specific Java methods. Therefore, the RTSJ specification may be extended to support the model described for ECP C, to enable a low level access to hardware resources.

3.5. Asynchronous thread signaling

Another feature of the ECP C profile taken from RTSJ is the ability to send signals from one thread into another asynchronously. The API includes functions to enable the reception of signals, and the definition of a maximum bound for the number of pending signals. These signals are processed by a handler in each thread. To be able to receive them, the receiver thread enables its reception in advance via an `ATS enable` function.

The communication model for the ATS is a signal transferred from one thread into another. The thread that signals and the handler of the signaled thread may run in parallel (i.e. the signaler does not have to wait for the end of the signaled thread). Likewise, the handler function of the signaled thread is executed as a segment of code in the signaled thread.

RTSJ uses exceptions to implement this type of functionality, while the micro and mini ECP C use callback functions for a similar functionality.

3.6. Raw memory access

In general, threads are not aware of the type of memory used to store their data. Only in some specific cases (e.g., to access low level I/O mechanisms mapped to memory positions) it is necessary to offer support to this functionality. To regularize the access to raw memory, the micro and mini ECP C profile (Listing 7) offer methods to read and write basic data types. The model provided by RTSJ offers unique types of memory that may be mapped to different positions as blocks. ECP C does not typically run on a virtual machine. Therefore, it does not need the specific objects included in Java to offer access to memory positions, which are already accessed from hardware.

4. Example application

This section shows a simple example application that uses the major properties of ECP C micro and mini frameworks. Listing 8 and Listing 9 provide a sample application with two threads developed in the ECP C described in Section 3. Its goal is to show a

```
//Memory management
void*  thread_malloc(size_t size);
void   thread_free(void* ptr); //**

//External events
void   attach_interrupt(
                    int interrupt,
                    void (*handler)(void),
                    int mode);
void   detach_interrupt( //**
                    int interrupt);

//Thread signaling
int    thread_ATS_signal(THREAD* th);
void   thread_ATS_enable();
void   thread_ATS_disable();
void   thread_ATS_set_maximum(int max);
int    thread_ATS_get_maximum();
```

Listing 6. Memory management, external events, and asynchronous thread signaling APIs. (**=available in the mini-profile only).

```
int           thread_read_byte(long position);
float         thread_read_float(long position);
int           thread_read_uint16_t(long position);
int           thread_read_uint32_t(long position);
void          thread_write_byte(
                    long position,
                    long data);
void          thread_write_float(
                        long position,
                        float data);
void          thread_write_uint16_t(
                        long position,
                        int data);
void          thread_write_uint32_t(
                        long position,
                        int data);
```

Listing 7. Raw Memory Access API.

```
00:  #include <ecp_framework>
01:  static THREAD * signaled= NULL;
02:  static long counter=0;
03:  static SEMAPHORE* sem=thread_semaphore_create(1,-1);
04:  void sporadic_task1(){ //Sporadic task activated with signals
05:  long counter_aux=0;
06:  thread_semaphore_acquire(sem);
07:   counter=counter_aux;
08:  thread_semaphore_release(sem);
09:  Serial.print("[T1] THREAD 1------\n
10:  counter:\t");
11: }
12:  void periodic_task2() {
13:  if (Serial.available() > 0)
14:  {
15:   byte incoming_byte = Serial_read();
16:   thread_semaphore_acquire(sem);
17:    counter=counter_aux;
18:   thread_semaphore_release(sem);
19:   result_signal1=thread_signal(signaled); //Async signaling
20:   }
21:  }
22:  void error_handler_task1(long errno)
23:  {
24:   Serial_print("[T1] Error-\t \n erno:");
25:   Serial_println(erno);
26:  }
27:  void error_handler_task2(long erno)
28:  {
29:   Serial_print("[T2] Error\t \n  erno:");
30:   Serial_println(erno);
31: }
```

Listing 8. Two tasks: one periodic task reading from the serial port device and a sporadic task that writes on the serial port.

simple application with two threads (sporadic and periodic) that exchange information (see Fig. 1). To activate the sporadic thread, the periodic thread sends signals with the ECP C API.

The application has the following features:

The first thread in the listing (Listing 8: 04) has a sporadic behavior. In each activation, the sporadic task sends the new counter information via the serial interface with `Serial print` (Listing 8: 09).

```
01: int main(){
02: THREAD*th1=sporadic_thread_create(
03:    &sporadic_task1,                 //invoked function
04:    NULL,                            //task parameters
05:    NULL,                            //stack pointer
06:    200,                             //stack size
07:    30,                              //priority
08:    10000,                           //Offset=10ms
09:    1000000,                         //T=1 second
10:    1000000,                         //Deadline=1 second
11:    300000,                          //Maximum cost=0.3 segs
12:    (ECP_FRAMEWORK_OVER_EXCEPT_POLICY
13:    |ECP_FRAMEWORK_SAVE_POLICY),     //MIT pol.
14:    2,                               // Maximum pending events
15:    &error_handler_task1);
16:    signaled=th1;
17: THREAD*th2=periodic_thread_create(
18:    &periodic_task2,                          //invoked function
19:    NULL,                                      //task parameters
20:    NULL,                                      //stack pointer
21:    220,                                       //stack size
22:    40,                                        //Priority=40
23:    0,                                         //Offset=0;
24:    1000000,                                   //T=1 second
25:    1000000,                                   //D=1 second
26:    300000,                                    //Max cost 0.3 secs
27:    &error_handler_task2);
28:    threads_subsystem_start();
}
```

Listing 9. Main code in charge of allocating the two tasks and starting the scheduling subsystem.

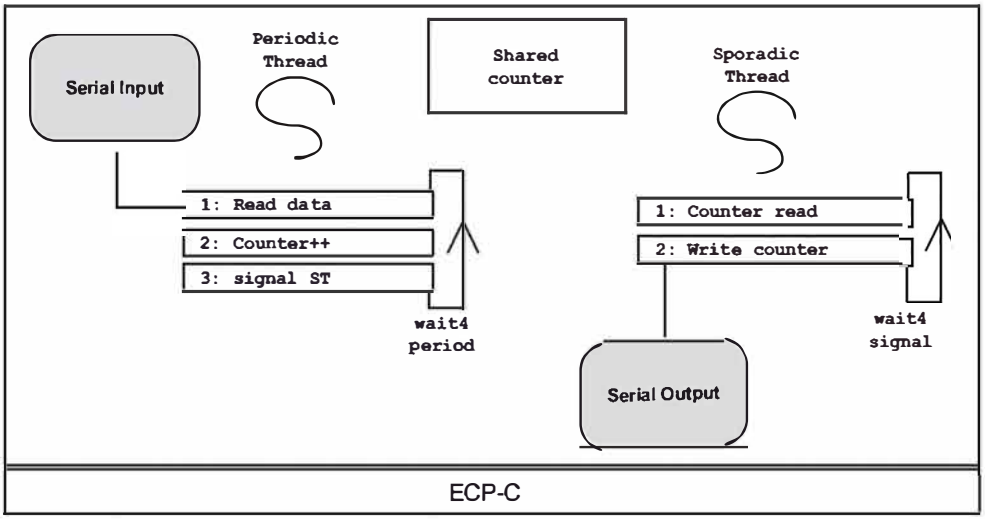

Fig. 1. ECP-C example application with two threads sharing information.

The second thread is a periodic thread (Listing 8: 05). It reads data from the serial interface via `Serial read` (Listing 8: 09). The infrastructure calls this method periodically.
Both threads share a global counter (Listing 8: 03). It is incremented by the periodic task (when it reads data from the serial interface), and read from the sporadic task when there are data. To avoid race conditions, the access to shared data is protected via a PIP semaphore (initialized in Listing 8: 03 to 1) with a PIP policy and used by the first (Listing 8: 06) and the second thread (Listing 8: 16).
The periodic task sends a signal to the second sporadic thread to control its sporadic activation (Listing 8: 19). The infrastructure controls the activation of the second thread so that its activation pattern is the one defined in the instantiation (Listing 8: 02 15).
Each thread has a handler method, invoked by the infrastructure in a case of a deadline miss, a cost overrun, memory exhaustion and when an asynchronous signal is received.

Lastly, to run the example, a `main` function is necessary to setup the whole real time infrastructure. The `main` function is in charge of allocating and setting up the two threads and starting the scheduling subsystem:

The first thread is initialized as sporadic (Listing 15: 02 15) with a 200 bytes stack and runs a 30 priority. It also has a minimum inter arrival time of 1 second with `OVER EXCEPT` and `EXCEPT` policies.
The second thread is periodic (Listing 15: 17 26) and its stack has 220 bytes, and runs at a 40 priority. It does not need to configure its activation because it is driven by the system clock.
Lastly, to start the scheduling system, it is necessary that `main` invoke the `thread subsystem start` function. Since that time on, the multithreading runs. Besides, the infrastructure activates context switching, overrun control, and clock abstractions. The developer cannot use this function to define background activities; which are supported by an aperiodic thread running at a lower priority.

5. Empirical evaluation

The goal of this section is to provide an evaluation of the absolute overhead introduced by the micro and mini framework running on constrained hardware. From the application perspective, the ECP C is a source of overhead and indeterminism. Threads need space in memory and take some time to perform context switching, which are sources of overhead.

A prototype of the library was developed (see Fig. 2) to analyze the performance of ECP C. The evaluation consisted of two different infrastructures: a raw implementation running on AVR GCC, and another on an embedded operating system. This dual infrastructure is representative for two common cases: applications running directly on raw hardware, and applications using the services of an operating system to build their logic. In both cases, the empirical evaluation sheds light on the extra time required for each configuration.

In a raw infrastructure, ECP C runs directly on the hardware which is, a priori, an optimal approach in performance terms. With a real time operating system as an intermediate element, ECP C reduces its implementation costs but increasing the overhead instead. The difference in performance represents the cost benefit relationship offered by the abstraction.

Two different prototypes (Fig. 2) were developed for an ATmega328 microcontroller [1]:

One for the naked AVR GCC tool chain included in its development kit (AVR GCC, 2014).
Another for a minimal real time operating system called ChibiOS [14].

In addition, the prototype includes an RS 232 interface to communicate the microcontroller with other systems. De facto, the RS 232 is the I/O interface for the ECP C framework:

In the naked implementation, the RS 232 interface is directly accessed from the tools included in the AVR GCC tool chain.
In ChibiOS, it has a driver controlling the access to the RS 232 serial communications.

On this infrastructure, the empirical goals are the following:

(i) To measure the absolute footprint introduced by ECP C when it is directly implemented on a naked microprocessor and when there is a real time operating system in between. The evaluation comprises a double perspective: absolute CPU time and memory.

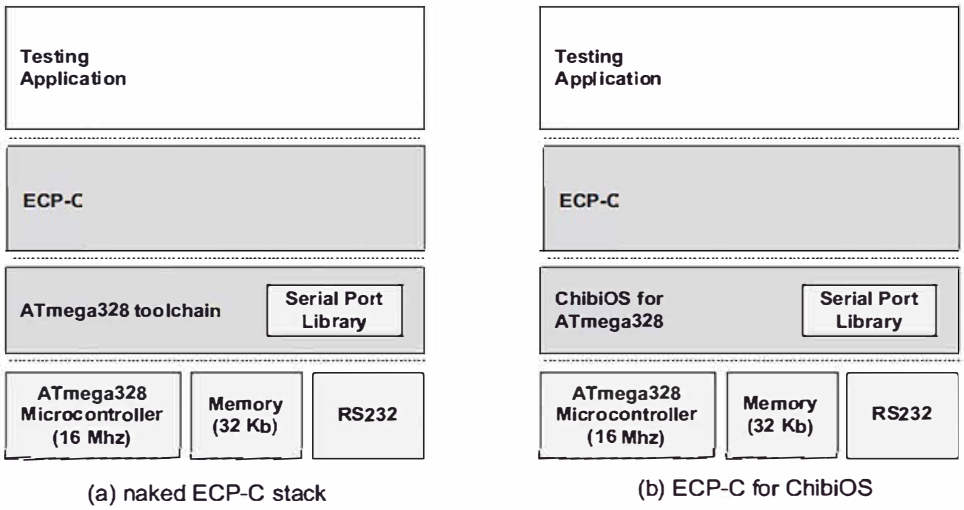

Fig. 2. The two ECP-C implementations: naked ATmega328 and ChibiOS for ATmega328.

Table 1
Resource allocation functions performance (Atmega328 running at 16 MHz and 32 Kbytes). Worst-case execution times. Maximum precision error: 0.3 µs.

Function	Time naked (µs)	Time ChibiOS (µs)	Memory naked (bytes)	Memory ChibiOS (bytes)
`x thread create(size)`	187.7	195.2	74 + stacksize	56 + stacksize
`queue create(elems)`	10.1	15.1	18 + elem*2	18 + elem*2
`semaphore create()`	7.1	7.4	4	4
`malloc(x)`	6.4	6.8	x	x

(ii) To evaluate the performance from the perspective of real time applications. To this end, a specific benchmark was developed for ECP C, adapting previous benchmarks to the new type of applications.

All these results contribute to evaluate the trade off among the use of ECP C directly on a naked microprocessor and with embed ded real time operating systems.

The benchmark consists of two subsets targeting different aspects of the infrastructure. The first subset is a microbenchmark related to the footprint of the main functions of the library (Section 5.1), and the second is an evaluation of the overhead intro duced by the scheduling subsystem in charge of allocating the CPU (Section 5.2). The microbenchmark explores the performance of the main functions in the API of ECP C. The second subset refers to applications with operational frequencies ranging from 83 Hz to 200 Hz, which have been derived from a previous AUTOSAR benchmark. This AUTOSAR benchmark evaluated the overhead introduced by the scheduler and the RS 232 communications on a set of tasks.

Table 2
Run-time function performance (Atmega328 running at 16 MHz and 32 Kbytes). Worst-case execution-times. Maximum precision error: 0.3 µs.

Function	Time naked (µs)	Time ChibiOS (µs)
`thread signal`	9,8	10,1
`queue set element`	7,7	8,0
`queue get element`	6,7	7,1
`semaphore P()`	2,3	2,1
`semaphore V()`	2,7	2,4
`ECP FRAME ATOMIC`	4,1	6,2
`gettime(bytes)`	3,9	3,9
`get error in time`	1,6	1,8
`yield()`	1,7–36	2–36
`sleep(x)`	x	x
`thread ATS enable`	3,1–36	3,1–36
`thread ATS disable`	3,1	3,0
`thread ATS signal`	2,8–36	3,1–36
`read byte`	1,2	1,4
`read uint16`	1,2	1,4
`read uint32`	1,6	1,7
`read float`	1,6	1,7
`write byte`	1,2	1,3
`write uint16`	1,6	1,4
`write uint32`	1,6	1,8
`write float`	1,6	1,8

5.1. Microbenchmark results

The results for the footprint (Tables 1 4) show the cost in time and memory for the naked ECP C implementation and for ECP C running on ChibiOS. The results show how some functions take a variable amount of time and memory, which depends on the parameters sent during the allocation of resources, while others are constant time. For all functions, this section analyzes the performance of the different policies defined for resource reserva tion/deallocation in ECP C.

The analysis starts with resource reservation for the allocation functions (see Table 1). Both implementations, i.e. the naked implementation and ChibiOS provide similar performance in CPU and the memory tests.

The most remarkable exceptions are the thread allocations and queue creation, which are more efficient in ECP C than in ChibiOS. The queuing model in ECP C is minimalistic, while in ChibiOS is more sophisticated, with features not required in ECP C. The same is true for the threading model of ECP C running on ChibiOS; ChibiOS performs some type of operations that are not required for the micro and mini profiles.

Another exception is the memory required to represent the thread, which in the naked implementation of ECP C is not

Table 3
Resource removal performance (Atmega328 running at 16 MHz and 32 Kbytes). Worst-case execution times. Maximum precision error: 0.3 µs.

Function	Time Naked (µs)	Time ChibiOS (µs)	Released memory (Naked)	Released memory (ChibiOS)
`Thread deallocate`	9.8	14.6	74 + stack_size	56 + stack_size
`Queue remove`	8.0	12.0	18 + elements*sizeof(void*)	
`Sem free`	6.2	10	4	
`Free(x)`	3.2	4.6	Allocated bytes	

Table 4
ROM sizes for different configurations. Results for the Naked Implementation and ChibiOS on Atmega328 (16 MHz–32 Kbytes). All cases use a minimal application to maximize the performance.

Configuration	ROM size in bytes
`Minimum application on AVR-GCC`	444
`Minimum application on ChibiOS`	1004
`Minimum application on naked ECP-C`	1684
`Minimum application on ECP-C (ChibiOS)`	2226

optimized like in ChibiOS. The reason why that happens is because ChibiOS uses a specific data representation for threads with 8 bits format. The raw implementation uses a 16 bits format, resulting in higher computational overheads.

The results (see Table 1) show that thread allocation takes less than 188 µs in the naked prototype and less than 196 µs in the ChibiOS. Likewise, the memory required in the queues is 74 bytes in the naked implementation and 56 bytes in ChibiOS. The difference in the memory required to store the state of the thread is because ChibiOS uses an optimized implementation, while naked ECP C does not. The memory required to allocate the stack in simple cases (like example shown in Section 4) is 200 bytes, which is the main overhead of a thread.

Queues require an internal array allocated during its initialization, which takes by 10 µs in the naked implementation and 16 µs when the implementation runs on ChibiOS default queuing systems. The two implementations require a variable amount of memory. This memory depends on the number of stored elements: 18 bytes for the internal queue representation, and 2 bytes per each additional element.

The results show that the semaphore allocation takes about 7 µs in both implementations, and requires an internal counter (a 4 bytes integer) to store the information.

Lastly, the memory management implementation provided by the naked implementation, which is the default included in the GCC, and ChibiOS are constant time (7 µs) provided that no memory reallocations occur for run time. Note that this assumption is true for the micro profile which prohibits resource deallocation functions.

Regarding run time functions (Table 2), it should be highlighted that all functions are memory safe (i.e., they do not allocate dynamic memory during its execution). They also show that in terms of overhead the evaluation identified three main behaviors. The first refers to simple and lightweight access (read/write) functions, which take less than 4 µs. The second refers to complex methods (like ATS methods, thread signaling, and queuing methods) which are all in the 4 36 µs range depending on the type of signaling carried out. The third type refers to special functions with special behaviors. One of them is `sleep` which stops a thread for certain amount of microseconds. Another is `yield` which forces an internal context switch.

The evaluation showed the following remarkable results:

The signal sent from one thread into another has a constant time cost in ChibiOS and the naked implementation of 10 µs because in both cases they are implemented from scratch. ChibiOS does not offer such support in its tasking model.

All `get` and `set queue` operations in the framework response under 8 µs for both the naked and the ChibiOS queues because they use similar implementation strategies. The naked implementation of ECP C is also a bit more efficient than the implementation running on ChibiOS.

The implementation of the `P()` and `V()` mechanisms in the semaphore take different times for the `V()` functionality. The operating system support offered by ChibiOS outperforms the implementation provided by the naked implementation. The reason why that difference happens is that the implementation provided by ChibiOS is optimized while the naked implementation does not.

The macro used to mark a piece of code as atomic (`ECP FRAME ATOMIC`) is more efficient in the naked implementation than on ChibiOS because the naked implementation has assembler optimizations while ChibiOS uses a less efficient API for interrupt inhibition.

The functions that access time (`gettime` and `get error in time`) offer similar performance in the naked implementation and when running on ChibiOS.

The `yield` function takes different costs depending on the type of operation carried out. The minimum cost refers to a `yield()` function that does not switches the thread: 1,7 µs. The second cost, 36 µs, refers to a new thread taking the CPU.

The `sleep` function sleeps a thread for a certain amount of time. In both stacks, the amount of time taken by the sleep function is the time requested in the invocation.

The mechanisms (`enable`, `disable`, and `signal`) used for controlling when a thread is asynchronously interrupted from another thread demand a variable amount of time. In all cases, there is a minimum cost referred to the time required to perform a change in the state of the thread: 3 µs. This cost increases to 36 µs as a notification to another thread is require. All mechanisms required for reading and writing data take less than 2 µs for the two ECP C implementations

Results obtained for the resource deallocation (Table 3) show the following patterns:

The deallocation of a thread takes a bit more in the ChibiOS (14.6 µs) than in the naked implementation (9.8 µs). This is because ChibiOS has to remove internal elements not required to implement the thread.

The same rationale is valid for the removal of queues and semaphores: the model of ChibiOS performs additional operations not required in the naked implementation. This extra effort transforms costs from 8.0 to 12.0 µs, for a queue; and from 6.2 to 10 µs, for a semaphore.

Lastly, the `free` operation in the naked implementation takes less time (3.2 µs) than in ChibiOS (4.6 µs). This difference in costs has to do with the default memory allocator of the naked implementation and the implementation included in ChibiOS, which are slightly different.

Lastly, Table 4 shows the minimum ROM size for different stacks that report the size of the image of a program when it is used a raw GCC infrastructure and it uses ECP C in combination with ChibiOS. Results show that the footprint of ECP C is better when it is compiled in a naked GCC infrastructure than on ChibiOS, which adds a 126% memory overhead. This difference in performance is because ChibiOS includes libraries that are not really used in ECP C.

Comparing the real time operating system and the naked implementations for ECP C, the main conclusion is that the most inefficient approach is the use of an intermediate operating system like ChibiOS. However, given the infrastructure used in the evalua tion, this inefficiency is not mainly in the support given to basic programming models, which are close in performance. The microbenchmark showed that most of the inefficiency comes from the memory required for the infrastructure, which is more reduced in the naked implementation than in ChibiOS.

5.2. Application evaluation

The last part of the empirical evaluation deals with the evalua tion of ECP C on a benchmark for AUTOSAR [3] applications. This benchmark was initially proposed in (Guoqiang [24] and extended later in [5] to different Java architectures. For the particular case of ECP C, the previous use cases have been extended with two inter ests in mind:

The first is to assess the overhead introduced by the scheduling subsystem implemented in ECP C. This scheduling mechanism limits the number of tasks that may be integrated within the system.

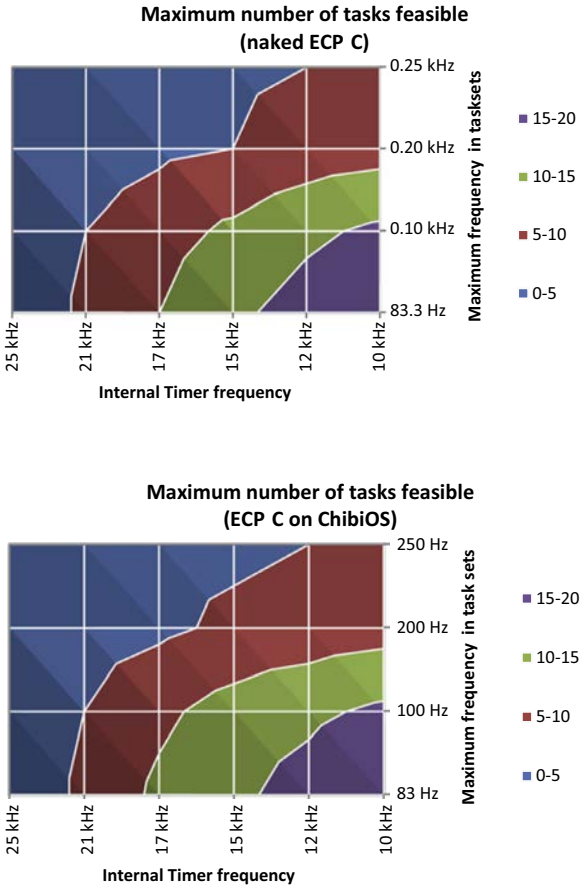

Fig. 3. Maximum number of threads allowed in the system. (16 MHz-ATmega328).

The second was to evaluate the communication overhead intro duced by the software stack on an RS 232 link, which is the basic I/O mechanism offered by the development infrastructure.

In both cases, the goal is to assess the performance of the naked ECP C and ChibiOS. Therefore, the first set of results refers to the maximum number of tasks feasible in the system (Fig. 2). In ECP C, the number of tasks depends on the frequency of the tasks of the system and the maximum frequency of the timer interruption used to support the clock and scheduling facilities. The results show the influence of the frequency of the tasks (y axis) and the timer frequency (x axis) on the number of tasks allowed in the sys tem. For low frequencies (task set operational frequency= 83 Hz, and internal update timer frequency = 10kHz) the naked ECP C implementation may support 15 tasks running in parallel. For the same configuration, ChibiOS supports 14 only. The difference in performance is mainly due to the internal algorithms included in ChibiOS that performs additional checks in the context switch of a task which are not required by ECP C.

The experiment showed the following results:

In the naked implementation and on ChibiOS (y axis in Fig. 3), the maximum number of tasks decreases as their frequency increases because each task has a context switching overhead that depends on the frequency. This is the reason why increas ing the number of tasks, the available computational time decreases.

In addition, the ECP C implementation includes a timer that performs small updates (x axis in Fig. 3) in the system. As in the case of several tasks running in the system, the higher the frequency of this timer is, the lower the number of tasks that may be hosted in the system.

The second set of results indicates that the higher the frequency of the interrupt, the higher the amount of CPU required for internal system updates (Figs. 4 7). The performance results show that the ECP C naked platform is not able to support 32 tasks with an inter nal timer with a 25 kHz timer for any frequencies in the evaluation range (from 83 to 250 Hz). ECP C on ChibiOS offers lower perfor mance as in the previous case. In general terms, this lack in perfor mance is attributed to the duplication of functions related to the ChibiOS real time operating system.

The analysis continues by providing detailed information on the cost of the scheduling system (which depends on the number of tasks, their frequency, and the frequency of the timer). For the low frequency frequencies (results included in Figs. 4 and 5) the following results are remarkable:

The benchmark is not able to offer low overhead (<10% of the total) for the operational frequencies of the benchmark. Even when the number of tasks is reduced (e.g.: 1), the timer (run ning at 8.3 kHz) takes an important amount of CPU time. The rationale is valid for the ChibiOS and also for the naked imple mentation of ECP C.

For this low frequency timer (8.3 kHz), the number of tasks that may be feasible in the system is relatively high (up to 32 tasks) in ChibiOS and the naked implementation.

On the other extreme of the benchmark, the results for the update timer, with a 25 kHz frequency, show how the number of feasible tasks reduces from 32 to only 4. The main reason for this low performance is the overhead introduced by the work associated to the timer task.

The results for high frequency task sets (from 200 Hz to 250 Hz) show how the time required executing the sub tasking system increases with respect to the previous results. Fig. 6 shows the

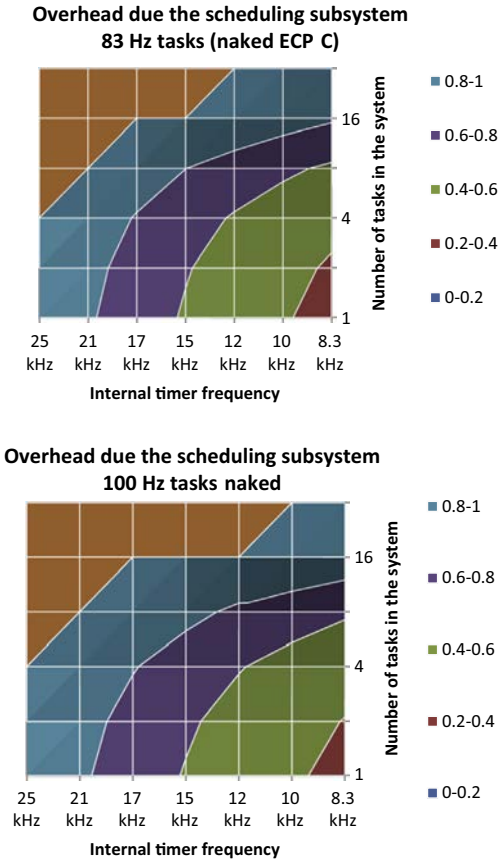

Fig. 4. Benchmark results. Scheduling System Overhead (16 MHz-ATmega328) with naked implementation.

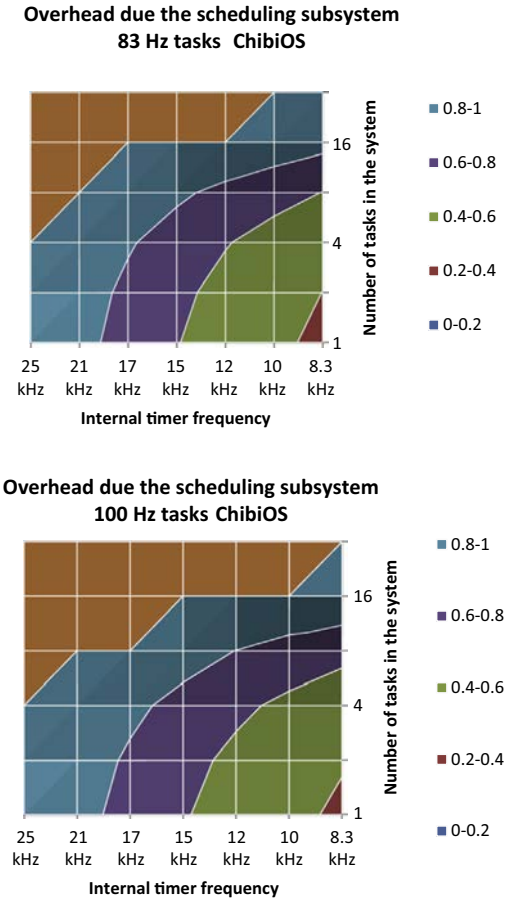

Fig. 5. Benchmark results. Scheduling System Overhead (16 MHz-ATmega328) with a ChibiOS implementation.

amount of CPU required for the naked implementation of ECP C and Fig. 7 for the ChibiOS implementation. By combining the information of the two experiments, the following results are remarkable:

The scenarios with less overhead, those with tasks running at 83 Hz frequency and internal timer with 8.3 kHz, take more than 40% of the available time to perform task switching and internal ECP C updates. This type of overhead means an increase of 20 points in relationship with the previous low frequency scenario.

The previous results for low frequency task sets show that the system may support up to 32 tasks, which in this high overhead scenarios reduces to a maximum of 8 tasks with applications running at 250 Hz.

As in the previous case, the worst case scenarios refer to those with the highest frequency in timers (e.g. 21 kHz). Previous low level frequency results showed that the maximum of 8 tasks reduces now to 4 or 2 tasks depending on the frequency of the specific subset.

Although the naked implementation is more efficient than the implementation running on ChibiOS, these differences are not very meaningful. The main motivation comes from the overhead introduced by the internal task model of ChibiOS, which performs some type of functionality not strictly required for ECP C.

The last set of results refers to the overhead introduced by the RS 232 interface which represents the basic I/O interface for the ECP C library. In this experiment the evaluation infrastructure has been modified, including a new real time Java node that communicates with ECP C by means of an RS 232 interface (see Fig. 8). For this last experiment, a 2.5 GHz machine with Oracle Real Time Java 1.5 [8] and a 16 MHz ATmega382 microcontroller connected via a RS232 link at 9600 bauds were used.

On the experiment, it has been measured the overhead introduced by the infrastructure on the different frequencies of the application. Basically, ECP C echoes all information sent from the Java infrastructure. Fig. 9 shows the main results obtained for naked and ChibiOS infrastructures:

For a reduced amount of data (1 4 bytes), the overhead introduced by the RTSJ and the ECP C naked implementation is 40% of the available bandwidth.

This overhead diminishes to less than 10% for 128 bytes, which is an acceptable performance.

Lastly, in serial communications, ECP C for ChibiOS is 3.5% better than on a naked infrastructure because the communication with the serial port uses an optimized implementation for ChibiOS that allows buffer optimizations. This type of facility is not currently implemented in the naked ECP C implementation.

The evaluation analyzes the impact on the benchmarked applications, which range from 83 Hz to 250 Hz frequency band, performing some I/O operations on the RS 232 device. For all these frequencies, the test measures the total overhead from the total time. The results for the naked implementation and ChibiOS show that I/O operations introduce a high penalty in applications (see Fig. 10):

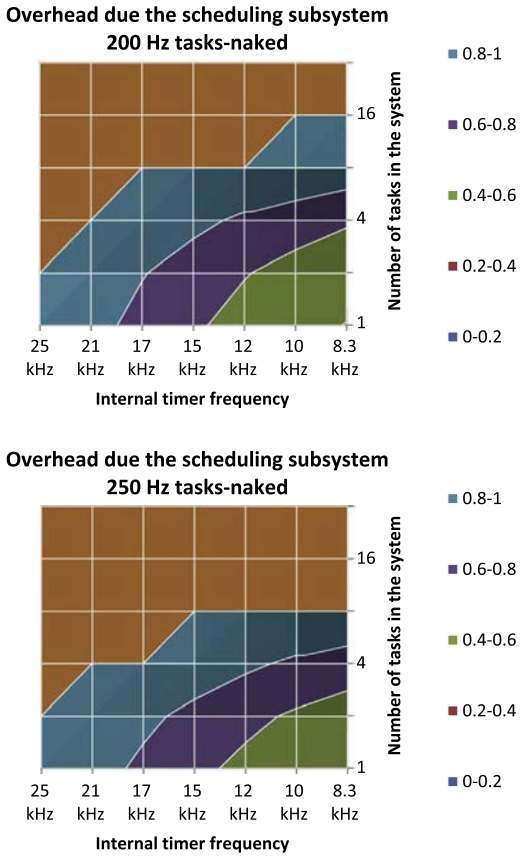

Fig. 6. Benchmark results. Scheduling System Overhead (16 MHz-ATmega328) with naked implementation.

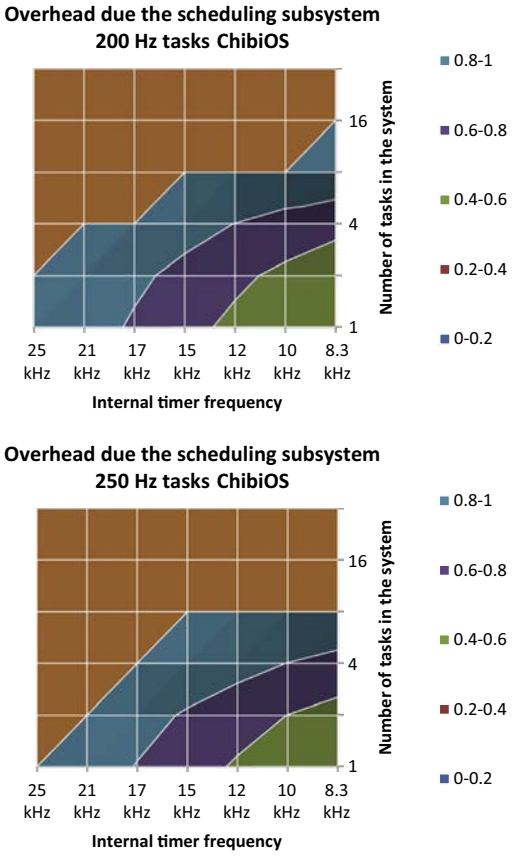

Fig. 7. Benchmark results. Scheduling System Overhead (16 MHz-ATmega328) with a ChibiOS implementation.

Even for low level frequencies running at 83 Hz and reduced amount of data of 1 single byte, the amount of time required to process data is relatively high (43% for the naked implementation, and 39% for ChibiOS). For this reason, the benchmark has been extended with new lower frequencies, 10 Hz, where the overhead is relatively low (<10%): 6% for the naked implementation of ECP C, and 4% for the scenario with ChibiOS. Another relevant result is that the benchmark collapses at 0.2 kHz. Applications with higher frequencies may not be running on ECP C because the time they require for I/O surpasses the application deadline. Applications threads running above this operational frequency should not use I/O. This low level facility has to be delegated to low frequency threads.

In addition, the previous trends on the performance of I/O with ChibiOS and the naked implementation upheld in the new scenario. The ChibiOS I/O outperforms the naked ECP C in all experiments.

6. Related work

The related work comprises real time operating systems, and other frameworks proposed for several Java and non Java programming languages.

Real time operating systems have long tradition [47] and there are many operating systems could be integrated to offer support to the proposed ECP C profiles (as done with the ChibiOS in the empirical section of the article). Nevertheless, the initial ECP C choice was to be created from scratch directly on the libraries included with the GCC [2]. Theoretically, this choice reduces footprint to a minimum but at the cost of having to implement low level facilities, such as context switch, as part of the infrastructure. Among all these real time operating systems, two motivating kernels were ORK [18] and SimpleRTK [23]. ORK proposes a small Ada C kernel useful to develop high integrity applications. However, ORK does not provide support for periodic, sporadic and aperiodic tasks that are implemented by the programmer with specific patterns. In ECP C, these patterns are provided by the infrastructure as its core programming abstraction in addition to mechanisms to access raw memory positions and interrupts. The second motivational approach is SimpleRTK, which is based on a previous microkernel described in [34]. The SimpleRTK footprint is similar to the ECP C footprint. However, SimpleRTK is silent on how to support different activation patterns in its core, which is a main goal in ECP C.

Although the proposed models are based on real time Java, other frameworks from other communities should be taken into consideration in the related work section. One of these communities is Ada. This community has extended the basic Ada runtime with support for concurrent facilities [11,40,53]. The subset described in this article has commonalities with previous Ada's work in a wide sense since both ECP C and Ada frameworks provide developers with a programming abstraction. Nevertheless, the mini and micro ECP C profiles are more particular in goals and in functionality than the concurrency facilities proposed for Ada.

Another important part of the community develops C applications. In the state of the art several frameworks have been proposed for the C programming language [15,38,32], and [51] addressing different programming issues. The language support for concurrency defined for RTC [32] includes a framework based

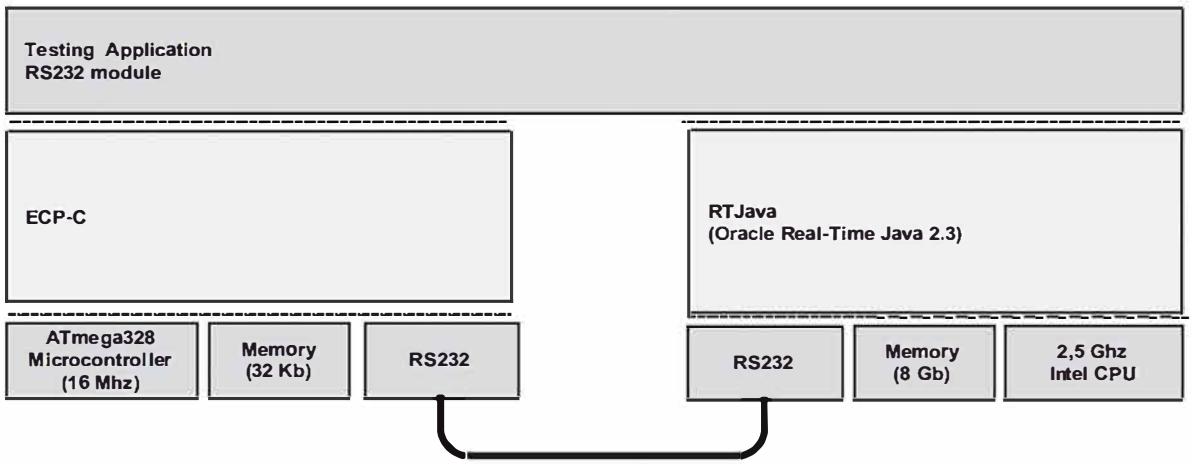

Fig. 8. Stack used to benchmark the RS-232 interface when it is used to communicate a Java application with an ECP-C application.

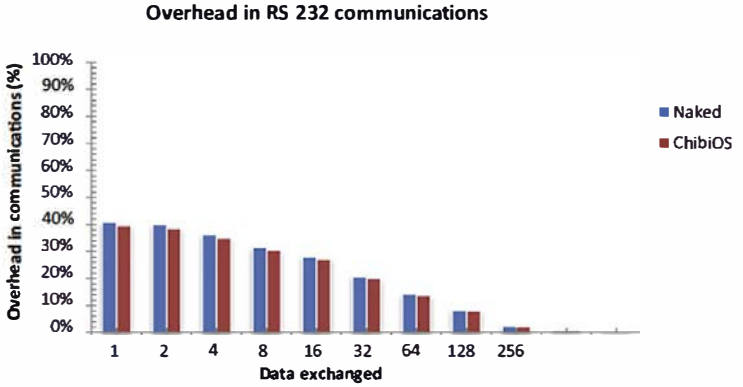

Fig. 9. Performance results. Overhead (16 MHz-ATmega328–2.5 Ghz Real-time Java stack via a 9600 RS-232 link) due to end points (CPU and runtimes).

on transactions, resources, and constraints that may be formally validated. RTC requires changes on the C compiler to offer a proper implementation. Some limitations of RTC have been addressed in [38] where the authors produced experience based extensions used to enhance real time C applications in the context of the HARTIK [13] and SHARK [22] kernels for robotics applications. The bottom up approach in HARTIK and SHARK defines a high level computational model that extends the kernel model into the user programming space. ECP C follows a top down approach, defining a programming model taken from a set of high level requirements mapped to a low level infrastructure. Recently [15] addressed this issue handling timing constraints in soft real time applications. ECP C and [15] share the idea of detecting and han dling timing constraints. The main difference among both approaches is that ECP C is more focused on the underlying con currency model than in [15]. The last piece of related work is mbeddr [51] which is an extensible C based programming frame work for embedded systems. In mbeddr, developers use a cleaned up C language with the goal of producing C code able to be integrated with IDEs to perform checks. In this context, ECP C does not introduce constraints in the C programming language but it provides a model for developing C based applications.

The last look into the related work returns back to the real time Java community and two major efforts: RTSJ and SCJ. In RTSJ, the API used to access raw memory and underlying hard ware is under refinement [20]. The API proposed in [20] is sim pler and enables a wide access to devices mapped to memory.

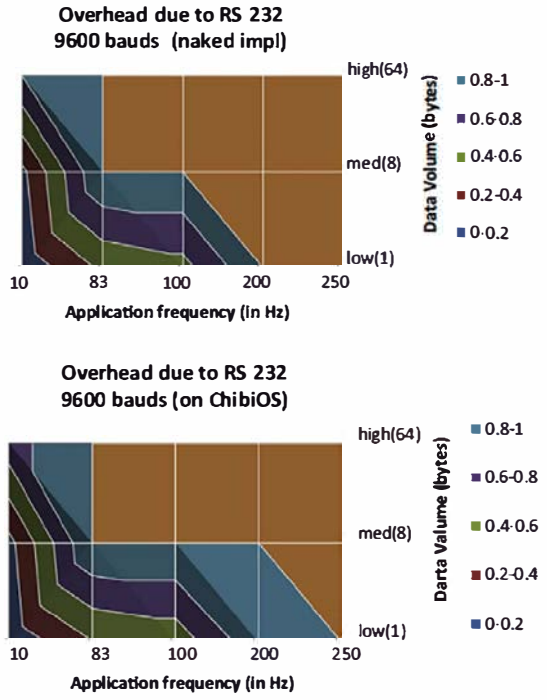

Fig. 10. Benchmark results. Overhead (16 MHz-ATmega328–2.5 Ghz Real-time Java stack via a 9600 RS-232 link). Application maximum frequencies ranging from 10 Hz to 0,25 kHz.

Table 5
Comparing the ECP-C against other Most Related Approaches Available in the State-Of-The-Art.

Framework /Operating System / Technique	Programming Language	Programming Patterns support	Implementation overhead	Computational Efficiency	Commonalties with ECP-C	ECP-C Contribution
RTSJ: General framework for real-time programming	Java	*High*	*High*	Low	Both take the form of a framework offering: real-time multithreading, priority inheritance protocols, raw memory access. Both integrate the concept of enhancement areas	Targeted to small microcontrollers running with C libraries In the mini and micro profiles, the API is shorter and closer to C
SCJ: A safety critical profile for real-time programming	Java	Med	Med	Low/Med	Both offer simple APIs that may be used to develop real-time applications	ECP-C is more general in goals. It is not targeted to high integrity applications
ORK: Operating System for real-time programming	Ada	Med	Low	High/Med	Both provide basic programming abstractions for periodic and sporadic tasks	ECP-C is focused on general applications that require real-time performance
ChibiOS: Efficient Operating System	C	*Low*	*Low*	High	Both provide the application with multithreading	ECP-C includes a programming model that offers support to real-time tasks
Real-time Utilities for Ada 2005	Ada	High	High	Med	Both offer facilities to build efficient real-time applications that consist of tasks	ECP-C is more specific than the Ada utilities
SimpleRTK: Real-time Micro-kernel	C	Med	Low	High	Both target to small size microcontrollers with reduced amount of memory	SimpleRTK does not support a model targeted to periodic tasks, while ECP-C provides a more complete programming model
Exception management [15]	C	Med	Low	High	This work and ECP-C have asynchronous mechanisms to detect CPU management faults	ECPC-C offers an integrated programming model
ECP-C (this work)	C	*High*	*Low*	High	-	-

The proposed ECP C model may be extended with stack overflow, and interrupt overload interrupts taken from the models described in [20] which are still under definition. Another source of influence for ECP C comes from a Java to C translator [44]; this infrastructure allows programming in Java. EPC C performs a similar translation between C and Java at a specification level from RTSJ to ECP C.

In addition to RTSJ, this work has influence from SCJ [28,55,42]. SCJ defines three integration levels: L0, L1, and L2 for different safety critical applications. L0 and L1 refer to simple applications running with a cyclic executive with periodic events in the L0 case, and with a sporadic event based behavior for L1. L2 includes the possibility of having threads to improve the programming model. The ECP C only includes different threads (periodic, sporadic, and aperiodic) that may share resources. ECP C is conceptually closer to the Ravenscar Java [30] approach than to SCJ. Ravenscar Java [30] is a SCJ prequel that defines applications with periodic threads and sporadic event handlers.

Associated with the infrastructure, there are also a number of benchmarks dealing with real time systems and applications per formance. The benchmark used in the empirical evaluation of ECP C consists of two parts: an ad hoc microbenchmark and appli cation scenario derived from AUTOSAR [5]. ECP C has commonali ties with the RTBenchmark [41] which measures the infrastructure quality. The main difference among both is that the benchmarking checked in ECP C is more specific than in RTBenchmark. There are also differences in the footprint of the benchmarks. Whereas the microbenchmarks in ECP C are small, in RTbenchmark requires 89 Kbytes, which are not available in the micro or mini infrastructures.

Some other benchmarks are more general and closer to the application. This is the particular case of Mibench [25], which includes a set of commercially representative embedded programs taken from automation, industrial, consumer applications, net working, and telecom applications. In contrast to this huge bench mark, which requires 300 Kbytes, ECP C has a small application test representative for AUTOSAR applications with operational fre quencies ranging from 80 Hz to 250 Hz.

6.1. Comparative

Table 5 establishes commonalities among the ECP C and other techniques and technologies. Among then, two works had a crucial influence on ECP C: the SimpleRTK kernel and the real time speci fication for Java (RTSJ). SimpleRTK proposes an efficient way to support real time operating facilities on top of small microcon trollers; this idea has been taken by ECP C but in a wider context. However, ECP C is designed for a larger audience, like the real time Java programming model, which is not targeted to any real time application. However, the use of a virtual machine abstraction could extremely inefficient for small microprocessors; this is the idea why C may be more interesting for developing embedded real time applications than Java.

In performance terms, it is difficult to establish empirical evi dence that confirms the statement because the current infrastruc ture does not allow a direct comparison. In some cases, the technology and/or infrastructure are too large to fit in this small amount of memory (e.g. with RTSJ), while in others (like in RTK) the comparison requires a mapping that has not been carried out to be able to compare the performance. Given that limitations, Table 5 defines three ranges (low, medium and high) speculating on the efficiency of the different runtimes. Java runtimes have been classified as inefficient computational infrastructures, and C run times as potentially the most efficient approaches, with Ada in between.

7. Conclusions

Next generation applications will benefit from having program ming frameworks that may help them to hide implementation aspects of a real time application. To reduce the complexity of using a plain infrastructure, it is proposed a real time threading library for C based applications. The resulting infrastructure is called ECP C and it is designed to offer a small footprint in compar ison to other high level programming languages. Mimicking real time Java, ECP C offers six enhancement areas: threading, resource sharing, memory management, external events, asynchronous sig naling and memory access. The empirical evaluation showed the feasibility of the approach in terms of footprint and performance by analyzing the overhead introduced by the abstraction into two different infrastructures: a naked implementation and on a small real time operating system.

Our future work is focused on extending the model to larger environments which may require the implementation of the gen eral ECP C profile, which will have to deal with scalability, and portability issues and its implementation on low cost Raspberry Pi which may run real time Linux kernels. Our ongoing work also includes the definition of a proper distribution model for the ECP C framework, departing from: [46,45,5,19]. In addition, another open challenge is the efficient integration benchmarks (mainly RTBenchmark and Mibench) in a constrained infrastruc ture with few bytes available, by splitting and dynamically down loading different parts of the benchmark into the constrained infrastructure.

Acknowledgements

This work has been partially funded by Distributed Java Infrastructure for Real Time Big Data (CAS14/00118) and by eMadrid: Investigación y Desarrollo de tecnologías educativas en la Comunidad de Madrid (S2013/ICE 2715). This research was sup ported by the national project REM4VSS (TIN 2011 28339) and by European Union's 7th Framework Programme Under Grant Agreement FP7 IC6 318763. The authors also acknowledge their anonymous reviewers for their efforts in improving the quality of the article.

References

[1] Atmel, ATMega 328 data sheet. http://www.atmel.com, 2012.
[2] AVR, The AVR GCC tool-chain. Available on-line at http://www.nongnu.org/avr-libc/, 2014.
[3] AUTOSAR, Release 4.0 Overview and Revision History. www.autosar.org., 2012
[4] M. Banzi, Getting Started with Arduino. " O'Reilly Media, Inc", 2009.
[5] P. Basanta-Val, M. Garcia-Valls, A distributed real-time Java-centric architecture for industrial systems, IEEE Trans. Ind. Inf. 10 (2014) 27–34.
[6] P. Basanta-Val, M. García-Valls, I. Estévez-Ayres, AGC Memory: A new Real-Time Java Region Type for Automatic Floating Garbage Recycling. ACM SIGBED. 2, 2005.
[7] Bollella G. et al., The real-time specification for Java. http://www.rtsj.org/, 2001.
[8] G. Bollella, B. Delsart, R. Guider, C. Lizzi, F. Parain, Mackinac: making hotspot real-time, in: 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2005, pp. 45–54.
[9] B. Bouyssounouse, J. Sifakis, Embedded Systems Design: the ARTIST Roadmap for Research and Development, Springer, Secaucus, NJ, USA, 2005.
[10] A. Burns, A. Wellings, Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX, 4th ed., Addison-Wesley Educational Publishers Inc., USA, 2009.
[11] A. Burns, A. Wellings, Concurrent and Real-Time Programming in Ada, 3Rev, Ed ed., Cambridge University Press, New York, NY, USA, 2007.
[12] A. Burns, A. Wellings, Real-Time Systems and Programming Languages, Addison-Wesley, 2001.
[13] G.C. Buttazzo, HARTIK: A real-time kernel for robotics applications, in: Real-Time Systems Symposium, 1993, Proceedings., 1993, pp. 201–205.
[14] ChibiOS, The ChibiOS/RT project page. http://www.chibios.org/, 2012.
[15] T. Cucinotta, D. Faggioli, Handling timing constraints violations in soft real-time applications as exceptions, J. Syst. Softw. 85 (2012) 995–1011.
[16] C. Cuevas, L. Barros, P.L. Martínez, J.M. Drake, MDE technology as support for real-time systems development environments, Revista Iberoamericana de Automática e Informática Industrial RIAI 10 (2013) 216–227.
[17] J. Dean, R. Bruce, M. Cameron, Changing the world with a Raspberry Pi, J. Comput. Sci. Colleges 29.2 (2013) 151–153.
[18] J.A. de la Puente, J. Zamorano, J. Ruiz, R. Fernandez, R. Garcia, The design and implementation of the open Ravenscar kernel, Ada Lett. XXI (2001) 85–90.
[19] J.A. Dianes, M. Díaz, B. Rubio, ServiceDDS: a framework for real-time P2P systems integration, in: 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010, pp. 233–237.
[20] P. Dibble, J. Hunt, A.J.a. Wellings, Programming embedded systems: interacting with the embedded platform, in: M.T. Higuera-Toledano, A.J. Wellings (Eds.), Distributed, Embedded and Real-time Java Systems, Springer, 2012, pp. 129–158.
[21] P. Fritzon, Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach. Wiley & Sons. ISBN 1118858972, 2014.
[22] P. Gai, L. Abeni, M. Giorgi, G. Buttazzo, A new kernel approach for modular real-time systems development, in: Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS '01), IEEE Computer Society, Washington, DC, USA, 2001, pp. 199–207.
[23] D. Garcia, M. Velasco, P. Marti, SimpleRTK: Minimal Real-Time Kernel for Time-Driven and Event-Driven Control. http://esaii.upc.edu/people/pmarti/10RRsimpleRTK.pdf, 1-51, 2010.
[24] Guoqiang Wang, M. Di Natale, A. Sangiovanni-Vincentelli, Improving the size of communication buffers in synchronous models with time constraints, IEEE Trans. Ind. Inf. 5 (2009) 229.
[25] M.R. Guthaus et al., MiBench: A free, commercially representative embedded benchmark suite, in: Proceedings of the Workload Characterization, Washington, DC, US, 2001, pp. 3–14.
[26] J.O. Hamblen, G.M. van Bekkum, An embedded systems laboratory to support rapid prototyping of robotics and the internet of things, IEEE Trans. Educ. 56 (1) (2013) 121–128.
[27] M.T. Higuera-Toledano, About 15 years of Real-Time Java. JTRES 2012, 2012, 34–43.
[28] JSR-302, Safety Critical JavaTM Technology. http://jcp.org/en/jsr/detail?id=302, 2011.
[29] H. Kim, R. Rajkumar, Memory reservation and shared page management for real-time systems, J. Syst. Architect. 60 (2) (2014) 165–178.
[30] J. Kwon, A. Wellings, S. King, Ravenscar-Java: a high-integrity profile for real-time Java, Concurr. Comput.: Pract. Exp. 17 (2005) 681–713.
[31] E.A. Lee, Cyber Physical Systems: Design Challenges. International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2008.
[32] I. Lee, S. Davidson, V. Wolfe, RTC: Language support for real-time concurrency, in: Proceedings of the IEEE Real-Time Systems Symposium (RTSS 91), 1991, 43–52.
[33] J. Liu, Real-time Systems, Prentice Hall, 2000, ISBN 978-0-13-099651-0.
[34] R. Marau, P. Leite, M. Velasco, P. Marti, L. Almeida, P. Pedreiras, J.M. Fuertes, Performing flexible control on low-cost microcontrollers using a minimal real-time Kernel, IEEE Trans. Ind. Inf. 4 (2008) 125–133.
[35] M. Masmano, I. Ripoll, A. Crespo, J. Real, TLSF: a new dynamic memory allocator for real-time systems, Euromicro Conf. Real Time Syst. (2004) 79–86.
[36] M.A.d. Miguel, E. Salazar, Model-based development for RTSJ platforms, in: Workshop on Java Technology for Real-Time and Embedded Systems, 2012.
[37] OpenGroup, IEEE Std1003.1-2008/Cor1-2013. Available (2014) on line at http://pubs.opengroup.org/onlinepubs/9699919799/, 2013.
[38] L. Palopoli, G. Buttazzo, P. Ancilotti, A C language extension for programming real-time applications. Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth International Conference on, 1999, 103–110.
[39] Rajkumar, R., Insup Lee, Lui Sha, Stankovic, J., 2010. Cyber-physical systems: The next computing revolution. 47th ACM/IEEE Design Automation Conference (DAC), 731.
[40] J. Real, A. Crespo, Incorporating operating modes to an ada real-time framework, Ada Lett. 30 (2010) 73–85.
[41] RTBench, 2013. Real time micro benchmark suite. Available online, 2014, on sourceforge.net/projects/rtmicrobench/.
[42] M. Schoeberl, H. Sondergaard, B. Thomsen, A.P. Ravn, A profile for safety critical Java, in: 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, 2007, pp. 94–101.
[43] L. Sha et al., Real time scheduling theory: a historical perspective, Real-Time Syst. 28 (2–3) (2004) 101–155.
[44] H. Schorrig, T. Henties, Java2C – developing in Java, deployment in C, JTRES (2010) 73–75.
[45] E.T. Silva, M.A. Wehrmeister, F.R. Wagner, C.E. Pereira, An approach to improve predictability in communication services in distributed real-time embedded systems, in: 5th International Workshop on Java Technologies for Real-time and Embedded Systems, 2007, pp. 121–126.
[46] J. Silva, T. Elias, F.R. Wagner, E.P. Freitas, C.E. Pereira, Hardware support in a middleware for distributed and real-time embedded applications, in: 19th Annual Symposium on Integrated Circuits and Systems Design, 2006, pp. 149–154.
[47] J.A. Stankovic, R. Rajkumar, Real-time operating systems, Real-Time Syst. 28 (2004) 237–253.

[48] Petteri Teikari et al., An inexpensive Arduino-based LED stimulator system for vision research, J. Neurosci. Methods 211 (2) (2012) 227–236.
[49] E. Upton, G. Halfacree, Raspberry PI User Guide, John Wiley & Sons, 2013.
[50] Vega-Rodriguez, Methodologies and tools for the design space exploration of embedded systems, J. Syst. Architect. (2014) 53–54.
[51] M. Voelter, D. Ratiu, B. Schaetz, B. Kolb, Mbeddr: an extensible C-based programming language and IDE for embedded systems, in: Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity, 2012, pp. 121–140.
[52] P. Yuemin et al., A demand response energy management scheme for industrial facilities in smart grid, IEEE Trans. Ind. Inf. 10 (4) (2014) 2257–2269.
[53] A.J. Wellings, A. Burns, A framework for real-time utilities for Ada 2005, Ada Lett. XXVII (2007) 41–47.
[54] A.J. Wellings, G. Bollella, P.C. Dibble, D. Holmes, Cost enforcement and deadline monitoring in the real-time specification for Java, ISORC (2004) 78–85.
[55] A. Wellings, M. Kim, Asynchronous event handling and safety critical Java, in: Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems, 2010, pp. 53–62.
[56] J. White, B. Dougherty, R.E. Schantz, D.C. Schmidt, A.A. Porter, A. Corsaro, R&D challenges and solutions for highly complex distributed systems: a middleware perspective, J. Internet Serv. Appl. 3 (2012) 5–13.
[57] Andrew Burkimsher, Iain Bate, Leandro Soares Indrusiak. A survey of scheduling metrics and an improved ordering policy for list schedulers operating on workloads with dependencies and a wide variation in execution times, Future Genation Computing Systems 29 (8) (2013) 2009–2025.
[58] Leandro Soares Indrusiak, End-to-end schedulability tests for multiprocessor embedded systems based on networks-on-chip with priority-preemptive arbitration, Journal of Systems Architecutre 60 (7) (2014) 553–561.
[59] Pablo. Basanta-Val, Norberto. Fernandez-García, Andy. Wellings, Neil. Audsley, Improving the predictablity of distributed stream processors.