Uniform scheduling of internal and external events under SRP-EDF

Simon Aittamaa
Lulea University of Technology
97187 Lulea Sweden
simon.aittamaa@ltu.se

Johan Eriksson
Lulea University of Technology
97187 Lulea Sweden
johan.eriksson@ltu.se

Per Lindgren
Lulea University of Technology
97187 Lulea Sweden
per.lindgren@ltu.se

ABSTRACT
With the growing complexity of modern embedded real-time systems, scheduling and managing of resources has become a daunting task. While scheduling and resource management for internal events can be simplified by adopting a commonplace real-time operating system (RTOS), scheduling and resource management for external events are left in the hands of the programmer, not to mention managing resources across the boundaries of external and internal events. In this paper we propose a unified system view incorporating earliest deadline first (EDF) for scheduling and stack resource policy (SRP) for resource management. From an embedded real-time system view, EDF+SRP is attractive not only because stack usage can be minimized, but also because the cost of a pre-emption becomes almost as cheap as a regular function call, and the number of preemptions is kept to a minimum. SRP+EDF also lifts the burden of manual resource management from the programmer and incorporates it into the scheduler. Furthermore, we show the efficiency of the SRP+EDF scheme, the intuitiveness of the programming model (in terms of reactive programming), and the simplicity of the implementation.

1. INTRODUCTION
Embedded real-time systems are naturally defined as time-bound reactions to external and internal events. The correctness of hard real-time systems relies on executing all reactions in accordance to their time-bounds. In addition to meeting the reaction deadlines, system resources need to be adequately managed. Embedded software plays an increasingly important role for the realization of such systems. To aid system development, resource management and scheduling can be simplified by the use of a real-time operating system (RTOS). However, commonplace RTOSs treats external and internal events in a non-uniform manner, both with respect to scheduling and resource management[4, 10, 9]. The scheduling of internal events (managed by the RTOS) is generally overruled by the underlying hardware interrupt mechanism (scheduling reactions to external events). The obvious effect is that scheduling is non-uniform between internal and external events, complicating both system design and analysis. However, more importantly is that the resource management provided by the RTOS is in effect set out of play. All resources that might be accessed by external event reactions (interrupt handlers) must be explicitly protected. This forces the programmer to manually manage critical sections (by interrupt masking) whenever resources claimed might be accessed by external events. This requires the programmer to diverge from the uniform system view and severely complicates the programming of real-time systems.

In this paper we present a method for scheduling and resource management that allows external and internal events to be treated uniformly from a programmers perspective. Our proposed solution deploys earliest deadline first (EDF) scheduling, and manages shared resources under the stack resource policy (SRP). Earliest Deadline First (EDF) scheduling has been shown to be optimal, given that there exists no shared resources [8]. In section 2 we introduce a collaborative hardware/software scheme that performs *pure* EDF (pure in the sense that both external and internal events are scheduled uniformly) scheduling onto platforms featuring static priority based interrupt hardware.

In addition to meeting the reaction deadlines, system resources need to be adequately managed. The stack resource policy is a priority ceiling protocol with the following properties [2]:

- schedulability test for systems with shared resources,
- deadlock free execution,
- resource management is incorporated into scheduling,
- task scheduling onto a single execution stack,
- pre-emption becomes as cheap as procedure calls,
- eliminates yielding and context switch overhead, and
- SRP gives a tight bound for priority inversion.

In conclusion SRP brings a set of sought after features for resource constrained real-time systems. The deadlock free execution increases system robustness, while the single stack

execution eliminate the need (and memory overhead) of multiple execution stacks and multiple execution contexts. This in turn eliminates yielding and context switch overhead, since a task is never started unless all resources are available for the task to complete. Moreover, pre-emption becomes as cheap as a procedure call, since there are no context switches in a SRP scheduled system. Finally, the bounded priority inversion ensures system responsiveness.

2. EDF SCHEDULING

In the following we assume that each task is triggered by an event. Each task has an absolute *baseline* (time of release), and an absolute *deadline*, the time in between gives a permissible execution window for the task. The intuition behind earliest deadline first scheduling is simple, tasks should be executed according to their deadline. This corresponds to scheduling the top element of a priority queue, holding all released tasks, sorted by their absolute deadline. Whenever a new event occurs, the corresponding task is released, its absolute deadline should be computed, and the task should be inserted in the priority queue accordingly. Typically a task release can stem either from external or internal events. Internal events may be postponed to occur at a later point in time, facilitating e.g., periodic events. Common-place micro controllers support efficient hardware scheduling of external and timer events through interrupt handlers. However, the scheduling policies of interrupt handlers are pre-dominantly adopting static priority schemes, which place a major hurdle to efficient implementation of EDF scheduling.

We address this problem by introducing a collaborative hardware/software kernel scheme (pure EDF) that deploys EDF scheduling on both internal and external events.

2.1 Design criteria

The following key design criteria should be met:

- pure EDF scheduling of both external and internal events
- high timing accuracy (high timer granularity and bounded timer jitter)
- low overhead

2.2 Kernel anatomy for pure EDF

In the following we outline the mechanisms of a platform independent EDF kernel in accordance to the discussed design criteria.

The task structure has the following selectors:

.baseline - absolute point in time

.deadline - absolute point in time

.relativeDeadline

.code

Key components are:

- state variables
  - *rq* - ready queue, ordered by absolute deadline in ascending order
  - *tq* - timer queue, ordered by absolute baseline in ascending order
  - *dl* - the currently shortest absolute deadline
- kernel operations
  - *dispatch(t)*

```
if t.deadline < dl then {
    predl = dl - push current deadline
    dl = t.deadline
    t.code() - execute task
    dl = predl - pop pre-empted deadline
    if rq != {} then dispatch(rq.dequeue())
} else rq.enqueue(t)
```

  - *postpone(t)*

```
tq.enqueue(t)
timer.schedule(t.baseline)
```

  - *interrupt(i)*

```
t = interruptTaskVector[i]
t.baseline = timer.getTime()
t.deadline = t.baseline + t.relativeDeadline
dispatch(t)
```

  - *timer.interrupt*

```
while tq.top().baseline <= timer.getTime() {
    r = tq.dequeue()
    rq.enqueue(t)
} if tq != {} then
    timer.schedule(tq.top().baseline)
dispatch(rq.dequeue())
```

- *rq.enqueue(t)*
- *tq.enqueue(t)*
- *rq.dequeue(t)*
- *tq.dequeue(t)*

In the following we will elaborate on the general considerations needed for meeting the stated criteria by a re-entrant kernel:

External events

For a pure EDF scheduler *all* scheduling decisions should be made based on basis of absolute deadlines. Hence, we need a mechanism to capture the absolute arrival times of external events, giving the baseline for the event and the corresponding task. (The baseline for internal events can be derived from the originating external event as discussed later.) In some cases we can rely on the underlying hardware to perform time-stamping of external events, however, in general hardware support is as best limited to a subset of the event sources. A generic solution is to perform time-stamping in the interrupt handler (*interrupt(i)*). The accuracy is in this case highly dependent on the blocking time of the interrupt handler, hence all *interrupts* are handled in a pre-emptive fashion.

Internal events

Internal events, may emanate either directly from the execution of a task, or being postponed to be released at a future point in time (given a postponed baseline, relative to that of the originating task). Hence we need mechanisms to directly dispatch (*dispatch(t)*) and postpone events (*postpone(t)*). For realizing the latter case, we use the micro-controller hardware timer set to trigger an interrupt, that in turn will schedule the *timer.interrupt* task to release the postponed event.

Timer management

The timing accuracy is directly dependent on the operating frequency of the timer. In tick based kernels, timing events occur periodically, causing pre-emption overhead to the currently executing task - hence system load will increase with increased timing accuracy [4, 7]. Fortunately, most modern micro-controllers offer remedy by means of output-compare functionality. For such platforms we may use a free running timer, and set the absolute time for an interrupt to be scheduled (*timer.schedule(t)*).

Absolute time representation

As the timer has a finite representation (number of bits), the free running timer will eventually overflow (wrap around), such giving a truncated representation of the absolute point in time. Under the condition that the number of timer bits is sufficient to encode the largest baseline offset we may undertake this truncated view of time. If we need a larger time span for baseline offsets, a virtual timer can be deployed that extends the hardware timer (least significants bits) with arbitrary number of most significant bits managed in software, implemented e.g., by simply advancing the most significants bits on a timer overflow. In the case that the hardware timer bits suffice, the only effect of increasing the timing accuracy is that the range of baseline offsets is decreased (this, without the performance penalty of a fine granularity tick based system). In case we need to cope with larger baseline offsets, the overhead is limited to that of the virtual timer implementation and the additional cost of accounting for the range of the virtual timer on related operations.

It should be noted that his scheme does not guarantee any bound on the time-stamping error, this must be ensured by the interrupt source or in software, e.g. by disabling interrupt sources until they are allowed to trigger again.

Priority queue management

Most kernel primitives must perform some queue management, thus the performance of the kernel is directly related to the efficiency of the queue management. While this can be done in several ways, we have chosen to use a simple linear queue for clarity. A heap-based data-structure would be more suitable for larger task-sets.

Example

Assume the following system configuration:

- *v=[timerTask, t1]*
- *rq*=$\{\}$, *tq*=$\{\}$
- *dl* = ∞
- *timerTask.relativeDeadline = 2*
- *timerTask.code = timer.interrupt*
- *t1.relativeDeadline = 7*
- *t1.code = ..., postpone(t2), ..., dispatch(t3)*
- *t2.baselineOffset = 4*
- *t2.relativeDeadline = 2*
- *t3.baselineOffset = Inherit*
- *t3.relativeDeadline = Inherit*

Initially the systems runs a non-terminating idle task with infinite deadline (hence *dl* is ∞). At this stage the system is possibly in low power mode awaiting stimuli. A task *t1* with relative deadline 7 is bound to interrupt source *s1*. Task execution, pre-emption, and permissible execution window is shown in figure 1. Stack allocation is shown in figure 2.

At time 2, an external event *s1* occurs, the corresponding interrupt handler is invoked by hardware, pre-empting the idle task. The baseline is set to 2, and the absolute deadline is set to 9. Task *t1* is dispatched and since it has the earliest deadline, *dl* is set to 9, stack resources are allocated, and *t1* is executed. Assume that *t1* first creates the postponed task *t2* with a relative deadline of 2, and a baseline offset of 4. This task is queued in the timer queue (*tq*) and the next timer interrupt is scheduled to occur at time 6. Then *t1* dispatches task *t3*, under the inherited base- and dead-line of *t1*, enqueuing it into the ready queue (*rq*).

At time 3, *t1* terminates, *dl* is restored to ∞, *t3* is dequeued from *rq* and dispatched. Since *dl* is ∞, *dl* is set to 9 and *t3* is executed.

At time 6, the timer interrupt pre-empts *t3*, sets baseline to 6, deadline to 8, and dispatches the *timerTask*. Since the *dl* is 9, *dl* is set to 8 and *timer.interrupt* is executed. Task *t2* is dequeued from *tq* and enqueued into *rq*. Since *tq* is now empty no further timer interrupt is scheduled at this time. *timer.interrupt* returns to dispatch, the previous deadline (9) is restored, and the first task in *rq* is dispatched. Since *dl* is 9, *dl* is set to 8, stack resources are allocated and *t2* is executed.

At time 7, *t2* terminates, stack resources are deallocated and the previous deadline is restored (9). Since *rq* is empty the dispatch exists, and the pre-empted *t3* is resumed. At time 8, *t3* terminates, stack resources are deallocated and the previous deadline is restored (∞). Since *rq* is empty, dispatch exists and the interrupt handler for *s1* terminates, resuming the pre-empted idle task.

3. SRP SCHEDULING UNDER EDF

The EDF scheduler discussed in section 2 can easily be extended to support shared resources with the Stack Resource Protocol. There are only a few extensions that need to be made compared to the EDF Scheduler (section 2). Here we outline the changes to the previous implementation:

The task structure is extended with the following selector:

.preemptionLevel

The added/modified key components are:

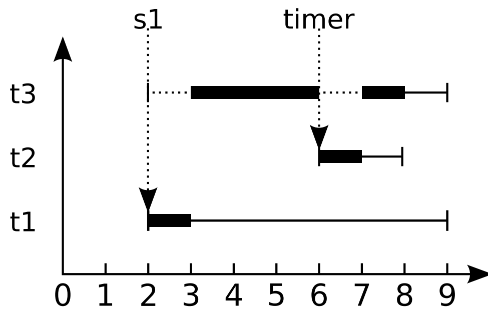

Figure 1: Example task execution, pre-emption, and permissable execution windows.

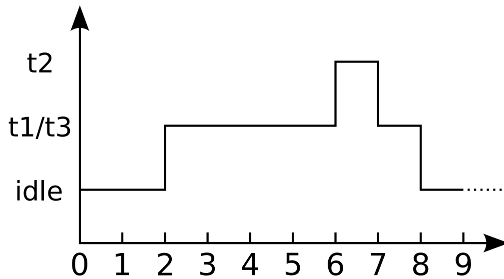

Figure 2: Example stack usage.

- added state variables
  - *sc* - the system ceiling
- modified/added kernel operations
  - *dispatch(t)*
    if $t.deadline < dl$ AND $t.preemptionLevel < sc$
    then {
    $predl = dl$ - push current task
    $dl = t.deadline$
    sync(t) - execute task and claim resource
    $dl = predl$ - pop pre-empted task
    if rq != {} then *dispatch(rq.dequeue())*
    } else *rq.enqueue(t)*
  - *sync(t)*
    $saved_ceiling = sc$
    $sc = t.preemptionLevel$
    t.code()
    $sc = saved_ceiling$

The fundamental idea with SRP is to allow resource-sharing in a well defined manner. Since we have resources we must in some way ensure mutual-exclusion, this is reflected by the addition of the *sync()* primitive. The *sync()* primitive ensures that the correct system ceiling is maintained. It should be noted that (as seen in the modified *dispatch(t)* pseudo-code) a lower pre-emption level means higher priority.

4. IMPLEMENTATION

4.1 Kernel

The kernel implementation shown in listing 1 and 2 is a minimalistic implementation of SRP-EDF and a derivative of TinyTimber as described in [7]. One important note is that in section 3 resources are equivalent to tasks. However, in the kernel implementation, tasks and resources are separate data-structures.

4.2 Example Application

The example application given in listing 3 is an implementation of the previously described example in section 2.2 for SRP-EDF.

Listing 1: kernel-source.h

```
// Implementation dependant MACROS
#define NTASKS           // Tasks (application dependant)
#define DISABLE()        // Disable interupts globally
#define ENABLE()         // Enable interupts globally
#define STATUS()         // Get interrupt status
#define TIMERGET(x)      // Get cur time
#define TIMERSET(x)      // Set next compare event
#define RETURN_TO(x)
// Push a new function on the thread stack for the ISR
// to return from. Used since we cannot call user code
// from interrupt handlers on most platforms
#define SLEEP()          // Enter sleep mode
#define SEC(x)           // Seconds to time ticks
#define MSEC(x)          // Millisec to time ticks
#define USEC(x)          // Microsec to time ticks
#define initObject(x)    // Initialize object with
                         // pre-emption level

typedef struct {
  int pl;
} Object;
```

Listing 2: kernel-source.c

```
struct task_block {
  Task *next;         // for use in linked lists
  Time baseline;      // event time reference point
  Time deadline;      // absolute deadline (=priority)
  Object *to;         // receiving object
  int (*code)(int);   // code to run
  int arg;            // argument to the above
};

struct task_block        tasks[NTASKS];
Task taskPool           = tasks;
Task taskQ              = NULL;
Task timerQ             = NULL;
Time timestamp          = 0;
Task cur_task           = NULL;
int system_ceiling      = MAX_INT;

static void enqueueByDeadline(Task p, Task *queue) {
    Task prev = NULL, q = *queue;
    while (q && (q->deadline - p->deadline <= 0)) {
        prev = q;
        q = q->next;
    }
    p->next = q;
    if (prev == NULL)
        *queue = p;
    else
        prev->next = p;
}

static void enqueueByBaseline(Task p, Task *queue) {
    Task prev = NULL, q = *queue;
    while (q && (q->baseline - p->baseline <= 0)) {
        prev = q;
        q = q->next;
    }
    p->next = q;
    if (prev == NULL)
        *queue = p;
    else
        prev->next = p;
}

static Task dequeue(Task *queue) {
    Task m = *queue;
    *queue = m->next;
    return m;
}

static void insert(Task m, Task *queue) {
    m->next = *queue;
    *queue = m;
}

void dispatch(void) {
  DISABLE();
  if ( taskQ &&
     ( (cur_task == NULL) ||
       ( (cur_task->deadline < taskQ->deadline)
         && (system_ceiling > taskQ->to->pl)
       )
     )
  ) {
    Task saved_task = cur_task;
    cur_task = taskQ;
    taskQ=taskQ->next;
    // In the paper sync is inlined
    sync(cur_task->to, cur_task->code, cur_task->arg);

    insert( cur_task, &taskPool ); // recycle task
    cur_task = saved_task;
  }
```

```
  ENABLE();
}

void TIMER_INTERRUPT_HANDLER(void) {
  Time now;
  TIMERGET(now);

  while (timerQ && (timerQ->baseline - now < 0))
    enqueueByDeadline( dequeue(&timerQ), &taskQ );
  if (timerQ)
    TIMERSET(timerQ->baseline);

  RETURN_TO(taskswitcher);
}

Task intsched(Time dl, Object *to,
              int (*code)(int), int arg) {
  Task m;
  Time now;

  TIMERGET(now);

  m = dequeue(&taskPool);
  m->to = to;
  m->code = code;
  m->arg = arg;
  m->baseline = now;
  m->deadline = dl + m->baseline;
  m->rel_deadline = dl;

  enqueueByBaseline(m, &timerQ);

  TIMERSET(timerQ->baseline);
  RETURN_TO(taskswitcher);
  return m;
}

Task postpone(Time bl, Time dl, Object *to,
              int (*code)(int), int arg) {
  Task m;
  Time now;
  DISABLE();

  m = dequeue(&taskPool);
  m->to = to;
  m->code = code;
  m->arg = arg;

  // Negative values => INHERIT
  m->baseline = bl < 0 ? cur_task->baseline
                       : cur_task->baseline;
  m->deadline = dl < 0 ? cur_task->deadline
                       : m->baseline + dl;
  m->rel_deadline = dl; // Used for preemtion levels
  enqueueByBaseline(m, &timerQ);
  TIMERSET(timerQ->baseline);

  ENABLE();
  return m;
}

// Extention, used for syncronous requests
// to other objects, not discussed in paper
// but used for sharing resources.
int sync(Object *to, int (*code)(int), int arg) {
  int result;
  int saved_ceiling_stacked;
  int status = STATUS();

  DISABLE();
  saved_ceiling_stacked = system_ceiling;
  system_ceiling = to->pl;

  ENABLE();
  result = code(to, arg);
  DISABLE();

  system_ceiling = saved_ceiling_stacked;

  // Try to dispatch any preemted events
  taskswitcher();
  if (status) ENABLE();
  return result;
}

void initialize(void) {
  // Set up the timer, taskpool etc,
  // Implementation dependent
}

void idle(void) {
  ENABLE();
  while(1) SLEEP();
}
```

Listing 3: application-source.c

```
#include "LPC17xx.h"
#include "uTimber.h"

typedef struct{
    Object obj;
    // Internal state variables here.
} myobj;

myobj objt2 = { initObject( _OBJT2_PL_ ) };
myobj objt3 = { initObject( _OBJT3_PL_ ) };
myobj eintobj = { initObject( _EINT0_PL_ ) };

int t1(myobj *, int);
int t2(myobj *, int);
int t3(myobj *, int);

int t1(eintobj *self, int arg){
    POSTPONE(MSEC(4), MSEC(2), &obj2, t2);
    POSTPONE(-1, -1, &obj3, t3); // Inherit bl and dl.
}

int t2(myobj *self, int arg){
    // Perform 1 ms of work.
}

int t3(myobj *self, int arg){
    // Perform 4 ms of work.
}

void EINT0_IRQHandler(void){
// Level triggered interrupt handlers needs to be disabled
// in the hardware interrupt handler, otherwise we create
// a infinite loop. The task in responsible for enabling it
// again after servicing interrupt (if appropriate).
    NVIC_DisableIRQ(18);
// Schedule task, with a deadline of 7 ms.
    intsched(MSEC(7),&eintobj, t1, 0);
}

void main(void){
    initialize();
    NVIC_EnableIRQ(18);
    idle();
}
```

5. EXPERIMENTAL RESULTS

5.1 Platform, Compiler and Setup

The platform used when measuring was an NXP LPC1769 [1] featuring an Cortex-M3 MCU, 512k Flash, and 64k SRAM. The GNU Compiler Collection (GCC) version 4.4.3 was used to compile the test-code. All code was compiler with the compiler flags *-mcpu=cortex-m3 -mthumb -O2*, the *-mthumb* switch is required since the Cortex-M3 core only support the Thumb-2 instruction set. All code was run from the flash and the flash accelerator module was disabled.

5.2 Memory Requirements

The memory requirement of the kernel, is as follows:

- Code-size 644 bytes (Flash)
- Data-size 20 bytes (SRAM)
- Task-size $24 * n$ bytes (SRAM)
- Object-overhead $4 * m$ bytes (SRAM)

Where n is the total number of tasks and m is the total number of objects. All memory requirements are derived from the compiled code.

5.3 Timing Behaviour

The timing behaviour of the kernel was measured in clock-cycles by sampling the RIT timer of the LPC1769.

In the first series of test seen in table 1 all assume the best-case. That is, both the external and internal events have the earliest deadline, the resources are available, and the only a single event occurs. However, the synchronous call is always a constant-time operation. The number of cycles measured is defined as follows:

Internal Event The number of cycles between the release-time of the task and the execution of the first instruction of the task.

Table 1: **Runtime of kernel primitives.**

Kernel Primitive	Clock Cycles
Internal Event	138
External Event	123
Synchronous Call	31

Table 2: Runtime of *postpone(t)* primitive, with the given taskQ length.

Queue Length	Clock Cycles
Empty	127
Length 1	136
Length 2	147
Length 3	160
Length 4	172

External Event The number of cycles between triggering of an interrupt and the execution of the first instruction of the interrupt-task.

Synchronous Call The number of cycles between invocation of the sync method and execution of the first instruction of the code argument.

To give an example of how the queue length impacts the timing behaviour we study a series of worst-case consecutive invocations of the *postpone(t)* primitive. The results from the test can be seen in table 2. The same worst-case behaviour can be expected for all kernel primitives that incorporate queue-handling. The accuracy of time-stamping can be made free of artifacts due to blocking (dominated by the queue-handling) if the interrupt hardware supports timer capture for external events (interrupts). In our simplistic prototype implementation we can clearly see the linear effect of using a linked list in the queue-handling. Other data structures, such as heaps, buckets, etc. provide a significant improvement for larger queue lengths, which leads to reduced queue-handling overhead, as well as improved accuracy of time-stamping in software (due to reduced blocking time).

6. RELATED WORK

Scheduling policies have been extensively studied from theoretical perspectives, and are at least for single core/single CPU systems to be considered as being well understood [11]. However, in practice results apply only if the model used for analysis corresponds to the system at hand. Work on scheduling theory often undertakes an ideal system model, neglecting scheduling overhead and interrupt handling. In [6] the cost of additional interrupt handling is included in the feasibility and schedulability test. However, in their model interrupt handlers are treated separately from application tasks (interrupts being scheduled at a higher priority). Our model differs in that interrupt handlers are indeed treated as being part of the application, where the occurrence of an interrupt corresponds to the release of a task. This integrated task and interrupt management model can also be found in [3]. However, in our work we focus on resource constrained systems under EDF and extend the results to EDF-SRP scheduling for systems with shared resources. In the context of stack based EDF schedulers, we also find AmbientRT [5]. However, under AmbientRT external events are treated separately from the application task set.

7. CONCLUSIONS AND FUTURE WORK

With the outset that embedded real-time systems are naturally defined as time-bound reactions to external and internal events, EDF scheduling is a natural choice. We have developed a scheme that through efficient software scheduling of interrupts accomplish pure EDF even on commonplace platforms that deploy static priority scheduling of interrupt handlers. Furthermore, we have show how the pure EDF scheme can be extended to perform SRP under EDF. This gives us efficient shedulability test, deadlock free single stack execution, efficient pre-emption management and tightly bound priority inversion. We have shown the efficiency of the proposed schema quantified by experiments on our prototype implementations.

Future work include investigating the possibility of hardware support for EDF+SRP scheduling of internal and external events, as well as time-stamping of external events (interrupts). We are also working on a complete system analysis (incorporating interrupt overhead, queue-handling overhead, blocking time, worst-case execution time, etc.) for SRP+EDF scheduled system. As a part of the complete system analysis we are investigating the impact of the underlaying data-structures relating to the queue-handling overhead and blocking time.

8. REFERENCES

[1] 32Bit ARM Cortex-M3, NXP LP1769. http://www.nxp.com/pip/LPC1769_68_67_66_65_64_4.html.

[2] T. P. Baker. A stack-based resource allocation policy for realtime processes. In *IEEE Real-Time Systems Symposium*, pages 191–200, 1990.

[3] L. E. L. del Foyo, P. Mejia-Alvarez, and D. de Niz. Predictable interrupt scheduling with low overhead for real-time kernels. *Real-Time Computing Systems and Applications, International Workshop on*, 0:385–394, 2006.

[4] FreeRTOS-A Free RTOS for ARM7, ARM9, Cortex-M3, MSP430, MicroBlaze. http://www.freeRTOS.org.

[5] T. Hofmeijer, S. Dulman, P. Jansen, and P. Havinga. Ambientrt - real time system software support for data centric sensor networks. In *Proceedings of the 2004 Intelligent Sensors, Sensor Networks and Information Processing Conference*, pages 61–66. IEEE Computer Society Press, 2004.

[6] K. Jeffay and D. L. Stone. Accounting for interrupt handling costs in dynamic priority task systems. pages 212–221, 1993.

[7] P. Lindgren, J. Eriksson, S. Aittamaa, and J. Nordlander. Tinytimber, reactive objects in c for real-time embedded systems. In *DATE*, pages 1382–1385. IEEE, 2008.

[8] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.

[9] O. V. Portal. http://www.osek-vdx.org.

[10] D. C. Sastry and M. Demirci. The qnx operating system. *Computer*, 28(11):75–77, 1995.

[11] J. A. Stankovic, M. Spuri, M. D. Natale, S. S. S. Anna, and G. C. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 28:16–25, 1995.