Abstract Timers and their Implementation onto the ARM Cortex-M family of MCUs

Per Lindgren, Emil Fresk, Marcus Lindner, and Andreas Lindner
Luleå University of Technology
Email:{per.lindgren, emil.fresk, marcus.lindner, andreas.lindner}@ltu.se

David Pereira and Luís Miguel Pinho
CISTER / INESC TEC, ISEP
Email: {dmrpe,lmp}@isep.ipp.pt

ABSTRACT

Real-Time For the Masses (RTFM) is a set of languages and tools being developed to facilitate embedded software development and provide highly efficient implementations geared to static verification. The RTFM-kernel is an architecture designed to provide highly efficient and predicable Stack Resource Policy based scheduling, targeting bare metal (single-core) platforms.

We contribute by introducing a platform independent timer abstraction that relies on existing RTFM-kernel primitives. We develop two alternative implementations for the ARM Cortex-M family of MCUs: a generic implementation, using the ARM defined SysTick/DWT hardware; and a target specific implementation, using the match compare/free running timers. While sacrificing generality, the latter is more flexible and may reduce overall overhead. Invariants for correctness are presented, and methods to static and run-time verification are discussed. Overhead is bound and characterized. In both cases the critical section from release time to dispatch is less than 2us on a 100MHz MCU. Queue and timer mechanisms are directly implemented in the RTFM-core language (*-core* in the following) and can be included in system-wide scheduling analysis.

1. INTRODUCTION

In the mainstream of embedded programming, C-code still remains the predominant means for software development. To facilitate the development, a vast number of light-weight operating systems are available, e.g., FreeRTOS [1], ChibiOS [2], and RIOT [3] and for larger platforms Linux/POSIX based and Win32 derivates. In common, they provide a thread based concurrency model, where the programmer has to take the full responsibility of coordinating scheduling and resource management, as very little support is given by the programming models and supporting tools [4].

In this paper, we explore a language based approach. The reactive programming model of RTFM-core (*-core* in the following) provides *tasks* with timing constraints and critical sections (treated as single-unit *resources*). As such, -core provides a model suitable to specify the timely behavior of the embedded software, as well as a formal underpinning amendable to both static and run-time verification. The supporting `rtfm-core` compiler produces C code that compiled together with a RTFM run-time system renders an executable. The RTFM-kernel is an architecture targeting bare metal (single-core) platforms designed to provide highly efficient and predicable Stack Resource Policy (SRP) based scheduling by exploiting the underlying interrupt hardware.

However, in prior work no kernel support was given for asynchronous tasks with timing offsets. In this paper, we address this problem with the goal to provide a transparent, abstract, and generic way of managing timer queue(s) and underlying hardware timer(s). Transparent w.r.t its use, i.e., the programmer should not need to think in terms of hardware timers when specifying the application at hand. Abstract in terms of the RTFM-kernel, (the obligation of the kernel is merely to manage scheduling) thus we seek a solution where the kernel itself is free of dependencies both to timer queue implementations and timer hardware specifics. Furthermore, the solution should be generic enough to cover a broad range of embedded platforms with little or no effort of porting. Additional requirements for robustness, performance, and predictability are efficiency, bound time implementations, complying with the task and resource model of SRP, along with invariants for correctness.

In this paper, we contribute beyond prior work by introducing a platform independent timer abstraction that relies on the existing kernel primitives. The proposed abstraction allows application and target specific implementations of timer queues and timer handlers. The timer handlers are treated as ordinary tasks in the system, while each queue is managed under protection of a critical section (resource) in the system.

Requirements to support abstract timers with respect to analysis and code generation in the `rtfm-core` compiler are discussed along with their performance implications. As a proof of concept, we develop and characterize two alternative timer implementations for the ARM Cortex-M family of MCUs: a generic (single queue/handler) implementation using the ARM defined SysTick/DWT hardware, and a multi-queue/handler implementation exploiting vendor specific match-compare/free running timer hardware.

Our experimental results indicate that for both generic and vendor specific timers the critical section from task release time to dispatch is less than 2us on a 100MHz MCU. We show that the vendor specific timers can be exploited to reduce latency, total overhead, and priority inversion in the system. Furthermore, we discuss the outsets for SRP based analysis of programs scheduled by virtual timers under the RTFM-kernel.

Finally, we present ongoing and future undertakings and sum up the presented contributions to conclude the work.

This work was partially supported by Portuguese National Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within project FCOMP-01-0124-FEDER-037281 (CISTER); and by FCT and EU ARTEMIS JU, within project ARTEMIS/0001/2013, JU grant nr. 621429 (EMC2); and Svenska Kraftnät (Swedish national grid).

EWiLi'15, October 8th, 2015, Amsterdam, The Netherlands.

2. THE RTFM-CORE LANGUAGE

The RTFM-core language is based on the notions of tasks and resources in correspondence to the Stack Resource Policy (SRP) model defined in [5]. For a detailed description on the original work on -core, we refer the reader to [6]. Here we give a brief overview.

2.1 RTFM-core programming model

In -core, tasks execute concurrently and run-to-completion. A task may request asynchronous execution of other tasks and *claim* (named) single-unit resource(s) for the duration of critical section(s) in a nested manner. Functionality is expressed using ordinary C-code. In recent work [7], the -core language has been extended to provide messages (task execution requests with timing offsets):

`async after` X `before` Y $t(...)$, where X defines the offset from the release time of the sender (baseline), Y gives the relative deadline, and t is the identifier of the task to execute.

2.2 RTFM-kernel design

In short, each task is implemented directly as an interrupt handler bound to the interrupt vector. Requesting a task for execution amounts to pending the corresponding interrupt, while claiming a resource for a critical section amounts to manipulating the interrupt hardware such to reflect the semantics of the system ceiling under SRP. The RTFM-kernel encapsulates the operations required for SRP based scheduling in a minimalistic API implemented as C-code macros. Those of interest for the presentation are: `RTFM_pend(i)`, which requests execution of the corresponding task `i`; `RTFM_lock(c)`, which reads and stores the old ceiling value on the stack and sets the new ceiling; and finally, `RTFM_unlock(c)`, which restores the old ceiling value from the stack.

Currently, the scheduling primitives have been implemented for the ARM Cortex-M range of MCUs [8]. The system ceiling is enforced either through interrupt masking (M0/M0+), or through (atomic) accesses to the NVIC `BASEPRI` register (M3 and above).

2.3 RTFM-core compiler

The `rtfm-core` compiler analyzes the declarative (static) task, resource, and communication structure and generates a C-code output referring to the RTFM-kernel primitives. Code generation and kernel primitives can be tailored to C-compiler specifics (currently supporting `gcc` and `compcert`).

3. TIMER ABSTRACTION

3.1 Definitions

We introduce the following definitions:

Definition We denote a task to be *postponed* if originating from an asynchronous message:

`async after` X `before` Y

with a defined baseline offset ($X > 0$). We denote the set of postponed tasks as OT.

Definition We have a set of virtual timers $\{VT_1 \dots VT_n\}$. Each virtual timer i is associated with a set of postponed tasks $ot(VT_i) \subseteq OT$, and a timer queue $tq(VT_i)$ (sorted by release time).

Definition We introduce a mapping M from virtual timers VT's to physical timers PT's, allocated on the target hardware.

A physical timer is *shared* if $M(VT_i) = M(VT_j), i \neq j$. We have the two edge cases when M is a 1-1 (complete) mapping between virtual and physical timers and the case when we have a single (shared) physical timer.

Definition For a physical timer PT_i, we denote $bw(PT_i)$ as the *bit-width* and $f(PT_i)$ as the *frequency* of operation (in Hz), $ra(PT_i)$ as the *range* of the timer (in seconds), derived from $2^{bw}/f$, and $pr(PT_i)$ as the *precision* of the timer (in seconds) given as $pr = 1/f$.

E.g., the range is given by $2^{bw(PT_i)}/f$, then a 32-bit timer operating at 1MHz gives a range of $2^{32}/(1*10^6 Hz) = 4295s$, with a precision of $1 * 10^{-6}s = 1us$.

3.2 ARM Cortex-M defined timers

The Cortex-M range of MCUs share the ARM defined core providing a 24-bit SysTick timer and a 32-bit debug timer (defined in the DWT unit).

3.2.1 SysTick timer

The SysTick timer is provided in order to generate periodic interrupts. When enabled, it counts downwards and when transitioning from 1 to 0 it sets a flag and (optionally) generates a SysTick interrupt. On zero, it assumes the value of the `RELOAD` register, hence a periodic behavior can be achieved with a minimal of programming effort. The current counter value (`CURRENT`) can be read, while a write to `CURRENT`, forces `CURRENT` = `RELOAD`. The frequency of operation is determined by setting the clock source (core/external). (Some implementations provide the option to prescale the core clock, e.g., /8.) The priority of the SysTick interrupt is programmable and an interrupt can be pended by setting the `PENDSTSET` bit in the `ICSR` (Interrupt Control and State Register). The SysTick timer is stopped when the processor is halted during debug.

3.2.2 Debug timer

The debug unit (DWT) provides a 32-bit free running cycle count register (`DWT_CYCCNT`). However, the DWT is instrumental for providing debugging support and hence not free to arbitrary use. However, we can safely enable and read the current `DWT_CYCCNT` value and use it as a 32-bit glitch-free time base. When the CPU is halted (e.g., during debugging), the counter is stopped.

3.3 Generic timer implementation

A flow chart is given in Figure 1. Whenever a new message enters first the queue (F_{yes}) the timer handler (task) is invoked. In the timer handler, if the release time has already expired (E_{yes}), the queued task is pended for execution, else (E_{no}) the timer is programmed for releasing the the task at its time for expire. In case a task is pended, the timer is iteratively dequeued until either the queue is empty (Q_{no}) or the release time not expired (E_{no}). In the latter case, the timer is setup to generate an interrupt for the next task to be released.

The timer handler is sketched in Listing 1, while the SysTick (set timer specific) implementation is outlined in Listing 2, along with a flow chart for its operation in Figure 2.

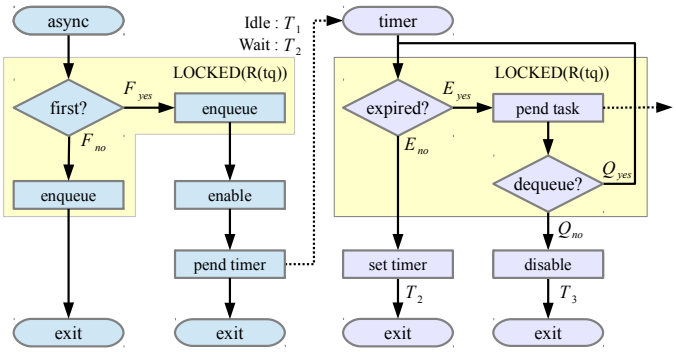

Figure 1: Timer queue (left) and timer handler (right) flow charts.

```
1  ISR SysTick_Handler {
2    lq = RTFM_LOCK(Q);
3    while ((T_CURR() - tq_h->bl) >= 0) {
4      // E_yes
5      RTFM_pend(tq_h->id);
6      if (tq_deq() == NULL) {
7        // Q_no
8        T_DISABLE();
9        RTFM_UNLOCK(lq);
10       return;
11     }
12     RTFM_UNLOCK(lq);
13     lq = RTFM_LOCK(Q);
14   }
15   T_SET(tq_h->bl);
16   RTFM_UNLOCK(lq);
17 }
```

Listing 1: `SysTickHandler.core`.

`T_CURR` is a macro for reading the `DWT_CYCCNT` (debug cycle counter), while `T_ENABLE()`/`T_DISABLE()` are macros for enabling/disabling the SysTick interrupt.

The `SYSTICK_MASK` is defined as the max reload value for the 24-bit counter. For brevity, initialization code is omitted. However, worth to mention is that we read `DWT_CYCCNT` to obtain a defined point in time (baseline) for the *birth* of the system. As a proof of concept, we have implemented a simple insertion sort queue (Listings 3 and 4).

3.3.1 Invariants for correctness

The invariants concern the logic of the interaction between the queue and the timer handler. Figure 3 depicts the overall timer operation. The following invariants should hold:

```
1  #define SYSTIC_MASK       ((1<<24)-1)
2  void T_SET(RT_Time t) {
3    RT_time diff = t - T_CURR();
4    if (diff > SYSTIC_MASK) {
5      SYSTICK_RELOAD = SYSTIC_MASK;
6    } else {
7      if (diff <= 0) {
8        PEND_SYSTIC()
9      }
10     SYSTICK_RELOAD = (diff & SYSTIC_MASK)-1;
11   }
12   SYSTICK_CURRENT = 0; // write to force reload
13 }
```

Listing 2: `SysTickSet.core`.

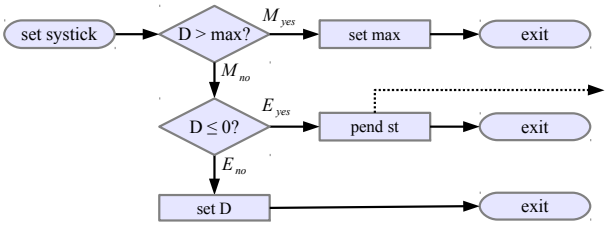

Figure 2: SysTick implementation.

```
1  typedef struct TQ {
2    RT_Time bl;
3    RT_Tid id;
4    struct TQ* next;
5  } TQ;
6
7  TQ  tq[TQ_LEN];    // queue
8  volatile
9  TQ* tq_h = NULL;   // head pointer
10 TQ* tq_f = tq;     // free pointer
11 TQ* tq_n;          // new
12 TQ* tq_c;          // current
13 RTFM_lock_t lq;
```

Listing 3: `tq.h` header.

- *Idle* state:
  - the timer queue is *empty*, and
  - the timer interrupt is *disabled*.
- *Wait* state:
  - the timer queue is *non-empty*, and
  - the timer interrupt is *enabled*.

The invariants are upheld by the implementation, in the following the (informal) reasoning.

Queue correctness.

Idle Assume the timer is in *Idle* state (the queue is *empty*), and the application emits an

`async after X ... t (...).`

Since the timer queue is *empty*, we follow the right branch (F_{yes}), i.e., we enqueue (X, t), *enable* the timer interrupt `T_ENABLE()`, and pend the timer interrupt `T_PEND`, which causes the transition T_1 to be taken. At this point, the queue is *non-empty*, and the timer is *enabled*.

Wait Assume the timer is in *Wait* state (the queue is *non-empty*), the timer interrupt is *enabled*), and the application emits an

`async after X ... t (...).`

In this case, we take an (implicit) T_2 transition (enabling the timer interrupt) and remain in *Wait* state. (Enabling the timer interrupt again does not invalidate the invariants and is more efficient in the worst case then conditionally enabling the interrupt.)

```
1  void tq_enq(RT_Time t, RT_Tid id) {
2    lq = RTFM_LOCK(R(tq));
3    if (tq_f == NULL) TQ_panic();
4    // allocate and fill new entry
5    tq_n = tq_f;
6    tq_n->bl = t; tq_n->id = id;
7    tq_f = tq_f->next;
8    if (tq_h == NULL || tq_h->bl - t > 0) {
9      // F_yes, put first in list
10     tq_n->next = tq_h; tq_h = tq_n;
11     RTFM_UNLOCK(lq);
12     T_ENABLE(); T_PEND(); return;
13   }
14   // F_no, put in middle or last
15   tq_c = tq_h;
16   while (tq_c->next != NULL && tq_c->next->bl < t) {
17     tq_c = tq_c->next;
18   }
19   tq_n->next = tq_c->next; tq_c->next = tq_n;
20   RTFM_UNLOCK(lq);
21 }
22
23 TQ* tq_deq() {
24   tq_c = tq_h;
25   if (tq_h != NULL) {
26     tq_h = tq_h->next;
27     tq_c->next = tq_f;
28     tq_f = tq_c;
29   }
30   return tq_c;
31 }
```

Listing 4: `tq.c` implementation.

Timer handler correctness.

Idle Assuming the *Idle* invariant, the timer interrupt is *disabled*. (Thus, the time-interrupt handler is *not* invoked even in case a compare match occurs and the interrupt is raised.)

Wait The time-interrupt handler is invoked when an interrupt occurs and the interrupt is *enabled*. The interrupt has been raised either due to a T_1 transition or due to the timer hardware on a compare match. Assuming the *Wait* invariants, there is (at least) one element in the queue, thus we can safely access `tq_h` for the `<expire?>` check. From there, the following cases apply:

E_{no} On E_{no} we program the SysTick timer `[set timer]` and return from the time-interrupt handler `[exit]`. This corresponds to a transition T_2, where we remain in *Wait* state (waiting for a compare match). This occurs in the case the timer is programmed the first time or on an overflow (when range of the timer is insufficient to reach the release time of the queued task). Notice, the latter may occur repeatedly until the `<expire?>` condition is met.

E_{yes} We release the expired task `[pend task]` and check if more messages are queued `<dequeue?>`. The following cases apply:

Q_{no} No further messages are queued and we disable the interrupt `[disable]`. This corresponds to the transition T_3 back to *Idle* state. At this point, the queue is *empty* and the interrupt is *disabled*.

Q_{yes} There is still at least one message in the queue and we check the `<expired?>` condition for the next queued task (a T_2 transition).

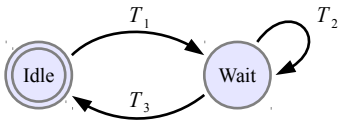

Figure 3: Timer states, *Idle* is the initial state.

3.3.2 Correctness under Concurrency

The sending task (accessing the timer queue through emitting an `async after X ...`) and the timer handler run concurrently and potentially preemptively to other tasks. Hence, we may be exposed to race conditions. To this end, we may either turn to re-entrant (lock-free) queue implementations [9] or protect the queue as a resource in the system. For this presentation, we turn to the locking mechanisms provided by the RTFM-kernel. In Figure 1, the LOCKED(R(tq)) areas (marked yellow/boxed) indicate the critical sections on the resource $R(tq)$. For the implementation, this amounts to `lq=RTFM_LOCK(R(tq))` on entering and `RTFM_UNLOCK(lq)` on exiting. Since the queue operations are protected by the resource $R(tq)$, they are from the outset of concurrency safe. While the release of an expired task t `[pend task]` is executed while holding the resource $R(tq)$, the SRP protocol ensures that t is only dispatched if it has a priority higher than the current ceiling. If t accesses the queue (through an `async after X ...`), then $\lceil R(tq) \rceil \geq p(t)$, which prevents dispatching t until $R(tq)$ is unlocked. (Moreover, under the assumption that the timer handler task is given, a priority equal or greater than t, t will not be dispatched until the timer handler task finishes.)

3.3.3 Characterization

The presented timer abstraction and its implementation gives the following key characteristics:

- Given a bound size queue, `tq_enq` is a bound time operation,
- `tq_deq` is a constant time operation (accessing and advancing only the head of the list),
- timer handling is safe w.r.t. invariants, and
- it allows implementation (and analysis) as part of the -core application[1].

Timing characteristics have been determined by measuring the clock cycle count (`DWT_CYCCNT`) on the current implementations (as presented in the paper). The experiments have been conducted on an STM32 F4, running at full speed (168MHz). The measurements have been repeated and consistent cycle counts have been observed. For the experiments we have used `gcc` v4.8.3 (**OL** gives the optimization level), with the default settings for the target architecture. All measurements include the overhead of the instrumentation code, hence safe and pessimistic w.r.t. actual performance.

The queue implementation has been characterized and is shown in Table 1. The **Baseline** gives the cycle count (including the call/return) overhead for inserting last in a

[1]In particular, the `tq_enq` is part of the execution time for the sending task, and the critical section (holding $R(tq)$) of the timer handler is constant time (although the execution of the timer handler may involve iterations). Notice here the "slight escape" from the critical section when releasing multiple tasks.

queue holding 1 element. The **LC** gives the *Linear coefficient* (added cost for each element in the worst case). As expected for insertion sort, we found the coefficient indeed to be linear.

Table 2 shows the latency from set release-time to dispatch in clock cycles. This gives an upper bound to the dispatch overhead (dispatching multiple queued tasks without leaving the handler always infer lower latency). The blocking (related to tq) inferred by the timer handler is brought to a constant by escaping the critical section for each iteration. The **Best/Nominal** values give the execution path when the queued task is *not* at the end of the queue, while the **Worst Case** includes disabling the timer interrupt.

Table 1: Complexity of the queueing algorithm.

OL	Baseline	LC
O0	178	26.5
O1	95	10
O2	78	9
O3	78	9
Og	96	9.5

Table 2: Latency from set release time to dispatch.

OL	Best Case	Nominal Case	Worst Case
O0	229	298	338
O1	122	153	188
O2	123	153	217
O3	123	153	217
Og	124	155	195

From this we can conclude that the generic implementation is capable of a low latency dispatch (< 2us, scaled to a 100MHz MCU). We have given the necessary WCET characterization for blocking, useful to SRP based timing analysis (e.g., response time and overall schedulability).

3.4 Vendor specific timers

An ARM Cortex based MCU typically comprises an ARM defined core and a set of vendor specific peripherals (typically including a set of timers/counters). Each timer/-counter has a defined set of features (supporting the intended use). The requirements for implementing the abstract timer architecture boils down to the following:

- n-bit width counter (+ for larger n) with
- interrupt capability (r), programmable priority (+),
- frequency (rate) relation to core-clock defined (r) or programmable (+), and
- programmable reload (r), match compare register (+).

While (r) this is required/sufficient, the suitability is improved (+) by a larger bit width, programmable priority, programmable frequency, and match compare functionality.

As representative use cases, we have studied two popular ARM Cortex MCUs, namely the NXP LPC1769 and the STM32 F4. In the case of the NXP LPC1769 (and similar), a Repetitive Interrupt Timer (RIT) is provided and a set of 4 equivalent and fully programmable 32-bit timers (the latter meeting all our requirements suitability criteria). In the case of the STM32 F407VET (and similar), we find a set of 12 16-bit timers and 2 32-bit timers meeting the requirements and suitability criteria.

For the implementation, the specialization to a vendor specific timer is isolated to the `[set timer]`. Writing the match compare is always 32-bit under the ARM memory model (the underlying timer hardware merely discards the 16 MSBs), hence the characterization applies in all cases. Table 3 gives the overhead for the isolated SetSysTick, while Table 4 depicts the overhead of setting a Vendor Specific (STM32 F407VET 32-bit) timer.

Table 3: Characterization of the Timer Set function for the SysTick Timer.

OL	Best Case	Worst Case
O0	54	67
O1	35	49
O2	33	41
O3	33	41
Og	33	41

Table 4: Characterization of the Timer Set function for a Vendor Specific timer.

OL	Best Case	Worst Case
O0	37	45
O1	21	28
O2	21	22
O3	21	22
Og	21	22

3.5 Compiler support

In order to automatically generate code for the proposed virtual timers, the -core to C compiler is required to undertake the following (additional) steps in the analysis: 1) derive the set of postponed tasks OT, 2) associate each postponed task $t_i \in OT$ to a VT_j, such that $p(VT_j) = p(t_i)$ (i.e., assign a virtual timer to each priority level in the tasks set OT), 3) derive a mapping M from VT to PT, 4) derive for each tq_i, where $PT_i \in PT$ the static queue length (tq_i being a potentially shared queue for PT_i, $M(VT_i) = PT_i$), 5) associate each tq_i to a resource $R(tq_i)$ with a ceiling value assigned under SRP (derived from the priorities of the tasks accessing the queue), 6) derive a time base $tb(PT_i)$ for each PT_i, and 7) generate C code definitions accordingly.

In the generated C-code, each task has a defined baseline set by reading the hardware timer (`T_CURR()`) for externally triggered tasks or given by the sending task. To the kernel we introduce a (queue and timer implementation independent) macro `RTFM_async_i(...)` scaling the virtual time based (in us) to that of the target PI_i. Our prototype -core compiler implementation assumes the case of a single (shared) physical timer. (The evaluation of multiple timers has been conducted manually.)

3.6 Timing performance

For scheduling analysis, the timer handlers can be seen as ordinary tasks, invoked once for the release of each message (plus the number of the range overflows present, e.g., in case of SysTick based solutions). With the outset that the mapping M is complete, there will be no priority inversion introduced by the timer handlers (as they operate operate at the same priority as the tasks they release). A timer handler t_h for a shared timer may preempt a task t_j ($p(t_h) > p(t_j)$), while $p(t_r) \leq p(t_j)$, being t_r the released task.

For vendor specific timers, we typically have the option to set the frequency $f(PT_n)$ (increased frequency gives an improved precision, while at the same time may increase the background load for processing timer overruns). The precision occurs as a jitter parameter to the scheduling. (In case the timer operates at the core clock frequency of the MCU (e.g., for our SysTick implementation), jitter is 0.)

3.7 Run-time verification

The proof of correctness for the implementation is informal. To this end, the `T_ENABLE()/T_DISABLE()` macros and `tq_eng/tq_deq` implementations have been extended to check the invariants. For run-time verification of timing constraints, the code generation for tasks is extended to check on return of each task t_i the condition:

`bl_t_i + dl_t_i > T_CURR()`, where `bl_t_i` is the (dynamic) task release time (baseline) and `dl_t_i` the specified (relative) deadline.

3.8 Assumptions

The general -core assumption on schedulability is that any message can have at most one outstanding instance. This allows the required (safe) queue length to be derived directly as the sum of tasks associated to the queue. In consequence, baseline offsets (`after X ...`) must be less or equal than the sender's inter-arrival time.

4. RELATED AND FUTURE WORK

In the context of light-weight operating systems, neither ChibiOS[2], RIOT[3] nor FreeRTOS[1] provide official characterized queue/timer implementations. TinyOS [10] (TEP 102/108) suggests an HAL virtualization layer. However, timers in TinyOS are outside their model of computation and treated as any other (arbitrary) event source. Contiki [11] provides the Rtimer library for scheduling real-time tasks. However, unlike our approach, their timer tasks are unsafe. Hence, our work presented can be considered as a baseline for future benchmarking.

Future work includes supporting baseline offsets larger than inter-arrival time for the sender. Moreover, as mentioned in Section 3.5, the support for abstract timers is currently limited to a single queue/timer handler. The timebase `T_CURR` is 32-bit, defined by the DWT. This limits the absolute time offsets. Longer offsets can be obtained at application level (manually keeping track of the number of activations until the desired time has expired). Automatic allocation and assignment of (potentially multiple) timer handlers according to the requirements of the application can support arbitrary offsets, as well as reducing priority inversion and overall overhead. Besides temporal properties, issues of energy consumption may be considered for multi-domain optimization. Moreover, the presented abstract timer architecture allows for multiple alternative queue implementations. By analyzing the task set, the compiler could choose the best fit (*linear, heap*, etc.) for each queue according to its characteristics (Section 3.3.3) and overall requirements (w.r.t. timing, memory, etc.).

5. CONCLUSIONS

In this paper, we have introduced abstract timers to the purpose of platform independent support for postponed tasks. The abstraction allows timer tasks (handlers) and queues to be statically allocated and included in system wide compile-time analysis under the task and resource model of RTFM. We have proposed a generic timer implementation that relies solely on the ARM defined Cortex-M core and existing RTFM-kernel primitives and is thus directly applicable to a wide range of commercially available MCUs. Correctness has been argued from invariants for queue and timer task interactions and queue consistency in a concurrent setting. Our experiments validate the feasibility of the abstract timer architecture and the presented characterizations of queuing overhead. Generic/vendor specific timer implementations give concrete bounds, useful as input to further response time and schedulability analysis.

6. REFERENCES

[1] FreeRTOS. (webpage) Last accessed 2015-09-18. [Online]. Available: http://www.freertos.org

[2] ChibiOS/RT. (webpage) Last accessed 2015-09-18. [Online]. Available: http://www.chibios.org

[3] RIOT. (webpage) Last accessed 2015-09-18. [Online]. Available: http://riot-os.org

[4] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, May 2006.

[5] T. Baker, "A stack-based resource allocation policy for realtime processes," in *Real-Time Systems Symposium, 1990. Proceedings., 11th*, Dec. 1990, pp. 191 –200.

[6] P. Lindgren, M. Lindner, and et.al, "RTFM-core: Language and Implementation," in *ESWEEK/CPSArch 2014*, 2014.

[7] P. Lindgren, M. Lindner, A. Lindner, V. Vyatkin, D. Pereira, and L. M. Pinho, "A real-time semantics for the IEC 61499 standard," in *ETFA 2015, September 8-11, 2015, Luxembourg*, 2015.

[8] J. Eriksson, F. Häggstrom, S. Aittamaa, A. Kruglyak, and P. Lindgren, "Real-time for the masses, step 1: Programming API and static priority SRP kernel primitives." in *SIES*. IEEE, 2013, pp. 110–113.

[9] A. Kogan and E. Petrank, "A methodology for creating fast wait-free data structures," ser. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 141–150.

[10] P. Levis, S. Madden, and et. al., "TinyOS: An operating system for sensor networks," in *in Ambient Intelligence*. Springer Verlag, 2004.

[11] A. Dunkels, B. Grönvall, and et. al., "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Emnets-I*, Nov. 2004.