RTFM-core: Language and Implementation

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

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

***Abstract*—Robustness, real-time properties and resource efficiency are key properties to embedded devices of the CPS/IoT era. In this paper we propose a language approach RTFM-core, and show its potential to facilitate the development process and provide highly efficient and statically verifiable implementations. Our programming model is reactive, based on the familiar notions of concurrent tasks and (single-unit) resources. The language is kept minimalistic, capturing the static task, communication and resource structure of the system. Whereas C-source can be arbitrarily embedded in the model, and/or externally referenced, the instep to mainstream development is minimal, and a smooth transition of legacy code is possible. A prototype compiler implementation for RTFM-core is presented. The compiler generates C-code output that compiled together with the RTFM-kernel primitives runs on bare metal. The RTFM-kernel guarantees deadlock-lock free execution and efficiently exploits the underlying interrupt hardware for static priority scheduling and resource management under the Stack Resource Policy. This allows a plethora of well-known methods to static verification (response time analysis, stack memory analysis, etc.) to be readily applied. The proposed language and supporting tool-chain is demonstrated by showing the complete process from RTFM-core source code into bare metal executables for a light-weight ARM-Cortex M3 target.**

I. Introduction

Embedded software development plays an ever increasing role to implement the functionality of embedded devices, and may often dominate the time-to market and development cost. In the mainstream, C-code still remains the predominant means for software development. To facilitate the development, a huge number of light-weight operating systems have been developed, e.g., ChibiOS, RIOT, and FreeRTOS. In common, they typically provide thread based abstractions, under which incorrect resource management may lead to both race conditions and dead-locks. To the end of compile time (static) verification, very little support is given by the programming models and supporting tools, thus at a large, the programmer has to reside to manual test based methods. Moreover, although light-weight in comparison to traditional full-fledged thread libraries and operating systems (e.g. POSIX threads under Linux), threads come with a run-time overhead. To overcome this hurdle, timing critical operations can be performed directly by associated interrupt handlers. However, these are treated non-uniformly (external) to the thread model, and special attention has to be put to the handling of shared resources, which further adds to the complexity of the design.

Alternatively, the programmer turns to a bare metal solution, taking full control of the underlying hardware, and thus, the complete responsibility for correctness of the implementation without any help of abstraction.

In this paper we present an alternative approach, based on a reactive programming model taking the outset of *tasks* and *resources*. The generic task/resource model can be summarized as follows: each *task* is: associated to a *priority* and a triggering *event*; is run-to-completion (without any notion of waiting for additional events); and may *pend* other events arbitrarily (this can be seen as requesting asynchronous execution of the corresponding task). A *resource* may be *claimed* for the duration of a critical section of a task in a LIFO (hierarchical) manner (i.e., enforcing nested critical sections).

This model coincides with the notion of tasks and resources used in the context of the Stack Resource Policy (SRP)[1].

With resource constrained systems in mind, the RTFM-core language and run-time system implementation imposes further restrictions, namely: events are single unit (i.e., each event can be either pending or not); an event is considered to be consumed when the corresponding task starts executing; resources are single unit, i.e., each resource can either claimed or free; priorities are considered as being static (assigned e.g., from the deadlines of the corresponding tasks).

In prior work we have shown how such models can be efficiently executed onto bare metal hardware by the RTFM-kernel primitives [2]. The RTFM-kernel guarantees deadlock-free execution, and efficiently exploits the underlying interrupt hardware for static priority scheduling and resource management under SRP. This allows a plethora of well-known methods to static verification (response time analysis, stack memory analysis, etc.) to be readily applied.

In our prior work, the programmer manually invokes the RTFM-kernel primitives (provided as C-code macros). This approach is of course plagued with all the common problems associated to the use of macros and libraries, since little compile time support can be offered to prevent the user from violating the programming model. Furthermore, in order to perform the necessary analysis (i.e., deriving the resource ceilings) for SRP based scheduling, the programmer had to provide a model (in XML) of the (per task) resource usage pattern to the Kernel Configuration Compiler (KCC) tool.

In this paper, we present a language (RTFM-core), which addresses the shortcomings of prior work. Programs (models) written in RTFM-core are automatically translated into C-code, where invocations of RTFM-kernel primitives are generated by the `rtfm-core` compiler, and thus correct by construction w.r.t. the task and resource model. Furthermore, the necessary analysis for SRP based scheduling is performed by the `rtfm-core` compiler directly on the given program, precluding the need for additional models (XML) and tools (KCC). Thus models in RTFM-core are truly executable!

```
Top   ::=  #> CCode <#
       |   ISR Id Int { Stmt }
       |   Task Id Int { Stmt }
       |   Func Id (CCode) { Stmt }
       |   Reset { Stmt }
       |   Idle { Stmt }
       |   Top Top

Stmt  ::=  #> CCode <#
       |   claim Id { Stmt }
       |   pend Id
       |   sync Id (CCode)
       |   Stmt Stmt
```

Figure 1. Simplified grammar for RTFM-core

II. RTFM-core Language

A. Syntax

The RTFM-core language (grammar is depicted in Figure 1), is designed from the outset of simplicity, with a clear focus on constructs for concurrency and real-time operations. The `CCode` terminal denotes the presence of embedded C-code. Each `ISR`/`Task` is associated with a static integer priority, while `Func`'s merely facilitate modularization. Following the grammar, `claim` is recursively defined (enforcing well formed hierarchical nesting of critical sections).

B. Well-formedness

The well-formedness criteria for the language involves satisfying the following constraints: each `ISR`/`Task`/`Func` should have unique identifier *Id*; each `pend` *Id* should match a `Task` or `ISR` *Id*; each `sync` *Id* should match a `Func` *Id*; transitive `sync` chains should be free of cycles; and `Reset`/`Idle` should be uniquely defined. The well-formedness of RTFM-core is agnostic of the semantics of the embedded C-code. Hence, even if a synchronous chain would never occur in practice (due to the logic of the embedded C-code) it is considered invalid. Moreover, the compiler does no attempt at matching formal parameter definitions (in `Func`) to actual parameters (in `sync`), to this end, we rely on the (external) backend tools for error detection and reporting.

Additionally, in order to uphold well formedness, we assume critical sections (defined in terms of potentially nested claims) to have single entry and exit points. This implies that the programmer must ensure that the embedded C-code does not contain control flow constructs (**`return`**, **`break`**/**`continue`**, **`goto`**) crossing critical section boundaries.

C. Dynamic Semantics

In the following, we give an informal dynamic (run-time) semantics for the RTFM-core language, following the notions of *task*'s and *resource*'s.

Each `ISR`, `Task` and the `Reset`, is bound to a sequence of statements and is referred to as a *task*. Execution of a *task* (a *task instance*) is triggered by the occurrence of a corresponding *event* and should run-to-completion (i.e terminate). (In the following we use *task* and *task instance* interchangeably.) `Reset` is a task, invoked once by the underlying run-time system at startup. A fully deterministic reset behaviour is achieved as other tasks (pended by the `Reset`) are blocked until `Reset` is completed. `Idle` is a (optionally non-terminating) background process in the system running at lower priority than all other tasks in the system. This allows the user to implement target/application specific power save modes, background jobs, or even hierarchical scheduling.

The *task*'s are concurrent in our model (and may execute in parallel). Their associated priorities *may* be used by the underlying scheduler (and corresponding resource and scheduling analysis). However, the execution model does *not* stipulate any single scheduling policy, and hence can be applied in many settings, single-core non-preemptive execution, concurrent/preemptive execution on single-cores, as well as to parallel execution onto multi-cores.

During execution a *task* may request (`claim`) a (single-unit) *resource* for the duration of a *critical section* (limiting concurrency, e.g., as a means to avoid race conditions on shared memory or specific hardware). Following the grammar, resources will be claimed in a nested LIFO manner. Moreover, a *task* may trigger asynchronous execution (`pend`) of other *task*'s. Functions execute synchronously (`sync`) on behalf of the sender, and may in turn emit events (`pend` tasks for execution) and claim resources (which are not the case for native C code).

As such RTFM-core is a generalization of the task/resource model underpinning the RTFM-kernel, and hence, as shown in Section III-A, RTFM-core programs can be straightforwardly targeted for scheduling by the RTFM-kernel primitives. Additionally, we offer alternative targeting to hosted environments (Linux/OSX/Windows) for the scheduling, which allows for parallel execution of RTFM-core programs onto multi-cores. [1]

III. RTFM-core Compiler

The `rtfm-core` compiler implementation follows the grammar defined in and allows C-code stored literally in the AST. (The syntax is `#>` for escaping to C and `<#` returning to RTFM-core). Errors in the model are reported by the compiler on first occurrence, indicating line, column and gives textual feedback.

The backend options provided by the compiler, namely `[-gcc |-ccomp |-clang]`, allows specific C-code generation, while the target options `[-rt|-kernel]` selects targeting the generated code according to type of execution environment. Additionally, `rtfm-core` provides graphical representation of task sets and resource utilization `[-gv_task |-gv_res]`, helpful in the development and documentation process (see Figures 3 and 2 for respective output).

Inclusion of C-code is supported through `#include` as well as external linking. Notice, that included C-code may *not* contain RTFM-core constructs, since includes are processed externally to the `rtfm-core` compiler.

[1] Notice, we discriminate in between `ISR`'s and `Task`'s purely for practical reasons (in order to later bind the RTFM-core entry-points to adequate positions of the underlying interrupt hardware ISR vector for bare metal execution.) To this end, other design choices may be considered, e.g., defining everything as tasks, with the option to (separately) bind specific tasks to the interrupt vector.

A. Targeting to the RTFM-kernel API

The RTFM-kernel, exploits the underlying interrupt hardware for performing scheduling and resource management under SRP.

In short, (assuming a static priority, preemptive, vectorized interrupt controller) each task (entry point) 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. To this end, e.g., the Cortex M3/M4 NVIC controller can be efficiently exploited by use of its `BASEPRI` register. An interrupt is admitted (scheduled) only if its logic priority is higher than `BASEPRI`, thus the `BASEPRI` register of the NVIC can be used to implement priority ceiling based scheduling in hardware. The NVIC supports atomic `SETBASEPRI(c)`/`SETBASEPRI_MAX(c)` (where c correspond to a priority), the latter updates `BASEPRI` only if $c >$ `BASEPRI`. The RTFM-kernel encapsulate the operations required for SRP based scheduling in a minimalistic API implemented as C-code macros. Those of interest are: `RTFM_pend(i)`, which pends `ISR[i]`; `RTFM_lock(c)`, which reads and stores the old ceiling value `old_c = BASEPRI` on the stack and performs `SETBASEPRI_MAX(c)`; finally, `RTFM_unlock()`, which restores the old ceiling value from the stack by performing `SETBASEPRI(old_c)`. For architectures not supporting setting a ceiling (`BASEPRI` or similar) the system ceiling can be implemented in terms of masking the individual interrupts/-tasks. In order to conserve interrupt handlers, a local (FIFO) scheduler may be used for each priority level (out of scope for this presentation).

For further details of the RTFM-kernel primitives and performance advantages, we refer the reader to [2].

In order to generate code for a specific target, the interrupt vector layout, and range of supported hardware priories is given. An example for the LPC1769 Cortex M3 is given in Listing 1. Each entry is associated an option $\{K|R|O|F|U\}$ providing information to the RTFM-core compiler used to automatically generate an appropriate mapping for the target vector table.

```
1   type isr_type =
2     | K (* reserved by the RTFM kernel                          *)
3     | R (* reserved by ARM                                      *)
4     | O (* overridable but has default implementation           *)
5     | F (* free to use by the application                       *)
6     | U (* used by the application                              *)
7
8   let isr_vector = [
9       (* Core Level - CM3 *)
10      (K, "&_vStackTop");             (* The initial stack pointer          *)
11      (K, "ResetISR");                (* The reset handler                  *)
12      (O, "NMI_Handler");             (* The NMI handler                    *)
13    ...
14      (R, "0");                       (* Reserved                           *)
15    ...
16      (O, "SysTick_Handler");         (* The SysTick handler                *)
17
18      (* Chip Level - LPC17 *)
19      (F, "WDT_IRQHandler");          (* 16; 0x40 - WDT                     *)
20      (F, "TIMER0_IRQHandler");       (* 17; 0x44 - TIMER0                  *)
21    ...
22      (F, "CANActivity_IRQHandler");  (* 50; 0xc8 - CAN Activity interrupt  *)
23  ]
24
25  let isr_maxpri = 32
```

Listing 1. `IsrVector.ml` for LPC1769.

In summary the RTFM-core compiler performs preprocessing and code emission.

The preprocessing is responsible for deriving the resource ceiling values by analyzing the (static) resource dependencies (per task), allocating each `ISR` to an index in the `isr_vector` according to name, and (iteratively) allocating each `Task` to a (unique) free index in the `isr_vector`. During the prepossessing the model is checked for consistency (i.e., each named `ISR` matches a `isr_vector` entry; each `ISR` can be assigned a unique `isr_vector` entry; and the uniqueness and existence of `ISR`/`Task`/`Func`/`Reset`/`Idle` definitions). The resource analysis traverse each entry point (transitive `sync` chain) for resource `claim`'s and record for each resource the ceiling (the maximum priority of tasks claiming each resource). Cyclic claims are rejected (hence recursive `sync` operations are not allowed in case any of the involved `Func`'s performs a `claim`).

The code emission includes: a mapping from logic to physical priories of the underlying hardware (`isr_maxpri`); a RTFM-kernel preamble; and the top level definitions. The preamble is composed of a set of adaptations for the hardware interrupt vector, setting task/priority and resource ceiling bindings and for each `ISR`/`Task`, code for setting priorities and enabling interrupts. For each top level definition in RTFM-core, a C function is emitted. Code generated for a `claim` amounts to `RTFM_lock(c)`, the statements of the critical section, followed by a corresponding `RTFM_unlock()`.

B. Backend Support for bare metal execution

The `RTFM-kernel.c` contains the `main()` which first calls the auto generated preamble Section III-A, then executes the `Reset` task, and when finished, enables other tasks and enters the `Idle` task.

The generated application code is included by `RTFM-kernel.c` in order to facilitate whole program optimization by the chosen backend compiler (GCC/CCOMP/LLVM). For bare metal execution, system initialization code must be provided that sets up the processor registers, peripherals, clocks etc. as well as initial heap contents, and calls `main()`.

The backend currently supports GCC and CCOMP (CompCert), while CLANG backend is still experimental. While the C language itself can be considered portable across compilers, we have considered and addressed the following issues: the build process (tool parameters, section naming for linking etc.), the compiler memory consistency model (in order to safely implement critical sections), and the method used for interrupt bindings.

C. Support for managed environments

RTFM-core has been developed from the outset of robustness, real-time performance, resource efficiency and ease of verification. However, the language as such is not limited to bare-metal targets, on the contrary the programming model based on reactivity, tasks and resources, is likewise applicable to embedded systems running (u)Linux, Windows Embedded/WinCE, and even to desktops and portables running Linux/OSX or Windows. Moreover, the latter are likely the environments under which RTFM-core systems will be designed and deployed from, and in many cases also communicate with, in a CPS or IoT setting.

To this end, we have developed a cross platform run-time environment for RTFM-core, allowing models to be compiled to native binaries and executed under Linux/OSX and Win32 hosting environments. A cross platform (eclipse based) setup has been created, which makes the source code for the RTFM-compiler and run-time system uniformly and easy accessible.

1) RTFM-RT Design and Implementation: From the outset of the RTFM-core execution model, `Task`'s and `Isr`'s are treated uniformly by the RTFM-RT.

Each task amounts to a statically allocated thread (implemented by Pthreads under OSX/Linux and windows threads under Win32). During run-time it waits (infinitely) for the release of a (single-unit) [2] named semaphore (identified by the unique task name. When released the corresponding function is invoked. When the task invocation is executed to end, the thread returns to its waiting state.

Each resource amounts to a (statically allocated) mutex, while a resource `claim` amounts to locking the mutex, performing the critical section, followed by unlocking the mutex.

a) Priority inversion and the deadlock problem: Under POSIX (i.e., Linux/OSX environments), scheduling policy and parameters are set such that threads are queued per priority and only higher priority threads my preempt the currently running thread. Mutexes are created as to emulate the priority ceiling protocol with the initial ceiling set to the priority ceiling for the resource. Hence, it implies an immediate ceiling policy, and should as such reduce overall priority inversion. However, as threads run concurrently and potentially in parallel, this does **not** guarantee deadlock free execution.

To this end, the RTFM-core compiler, constructs a multi-graph of resource claims for all tasks in the system, and detects potential deadlocks by performing a topological sort. Given the `-gv_res` option a `.gv` is generated for the multigraph, which facilitates the programmer to identify and address the potential deadlock (see Figure 2).

While, potential deadlocks may be seen as a major limitation of RTFM-core/-RT, we like to stress the fact that deadlocks are a major obstacle to any lock based concurrent execution model, and not unique to RTFM-RT. In contrast to native use of Pthreads, or most other threading APIs, we offer the service to the programmer of statically (at compile time), **analyze** the program for potential deadlocks (and once and for all eliminate them). Hence, there is using RTFM-core no need for complex deadlock detection during run-time, neither with the run-time system itself nor within the user application. If RTFM-RT is seen merely as yet another threading library, this feature alone can be argued as a huge step towards robust thread-based programming!

b) Targeting to the RTFM-RT API: Code generation from RTFM-core models to C-code, follows at a large the compilation process presented for targeting the RTFM-kernel. Code generation for the RTFM-kernel has been carefully designed to infer a minimum of CPU and resource overhead, this is not an issue in any hosted environment (as the overhead of the underlying operating system is likely magnitudes larger than the RT primitives themselves).

This allows us the luxury of some additional book-keeping, which allows to further facilitate debugging and producing visual feedback. Each task is invoked with its identifier (index in the thread pool), and sent as an additional argument to the RTFM-PT primitives, which allows the run-time system to report both resource usage and communication patterns.

c) Debugging facilities: The RTFM-RT primitives are extended to support visual feedback during execution, the `DEBUG` option exposes the following:[3]

Action				Description
Pend	by task	$\rightarrow$	of task	pending of a `Task/ISR`
Sync	by task	$\rightarrow$	of function	call to `Func`
Claim	by task	$\rightarrow$	of resource	(potentially blocking)
Claimed	to task	$\leftarrow$	of resource	when resource locked
Released	by task	$\rightarrow$	of resource	when resource un-locked
End	of task			when run-to end

Additionally the inner workings of the RTFM-RT can be observed by enabling the `DEBUG_RT` option, it exposes the following:

Action				Description
SPost	by task	$\mapsto$	of semaphore	success/saturated
SWait	of task	$\mapsto$	on semaphore	in waiting state
SReleased	of task	$\leftarrow\!\shortmid$	of semaphore	in released state
MLock	by task	$\mapsto$	of mutex	(potentially locked)
MLocked	by task	$\leftarrow\!\shortmid$	of mutex	when locked
MRelease	by task	$\mapsto$	of mutex	when unlocked

Notice, that the `DEBUG_RT` nicely reveals the duality in between the application (mutator) and the run-time system.

The `DEBUG` option largely facilitate application development/debugging, and makes manual instrumentation of task communication and resource utilization obsolete. The `DEBUG_RT` option on the other hand is useful to understand and further develop the run-time system (and as such of less importance to the end-user).

D. Example system

In order to showcase the RTFM-core language, compiler and target environments, we give a small, yet illustrative example, showing the use of tasks, interrupt handlers, functions and resources. The source file `Example.core` is depicted in Listing 2, defining a system a `Task` and an `ISR` and the shared resources `R1/R2`. The `Reset` code pends the `task` twice, and `TIMER0_IRQHandler` (once). Due to the single buffer behaviour of events (in interrupt hardware and our semaphore implementation) only a single event of each type will be outstanding (and additional ones silently dropped). The `task` performs a synchronous call (`sync`) to the function `func`, which in turn, claims (`claim`) the resources `R1` and `R2` in

[2]The single-unit limitation mimics the hardware pend vector (which is implementing a one-sized buffer for pending events). In this way, the run-time semantics comes close (but not equal) to the RTFM-kernel semantics. This is of course a design choice, the implication of larger buffers can be put to further investigation.

[3]Since the `sync` $\rightarrow$ `Func` is performed directly inside the application, there is no way for the run-rime system to directly track (and report) such actions. To this end, the -D option to `rtfm-core` inserts instrumentation code, which invokes the run-time system for (optional) presentation.

a nested fashion. The `TIMER0_IRQHandler` has a similar behaviour while claiming the resources in opposite order (i.e, `R2` followed by `R1`).

The corresponding generated code for RTFM-RT and RTFM-kernel are shown in Listings 3 and 4 respectively.

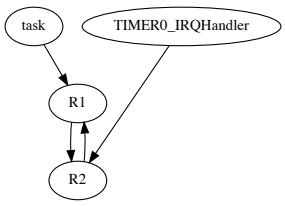

Figure 2. Potential deadlock (shown as cycle), produced by the `[-gv_res]` compiler option for the example application.

```
#>
#include <stdio.h>  // printf, etc.
<#
Reset {
    pend task;
    pend task; // will be discarded due to single buffer
    pend TIMER0_IRQHandler;
}

Func void func() {
    claim R1 {
        #>printf("func_R1_claimed\n");<#
        claim R2 {
            #>printf("func_R2_claimed\n");<#
        }
    }
}

Task task 1 {
    #>printf("task\n");<#
    sync func();
}

ISR TIMER0_IRQHandler 2 {
    #>printf("TIMER0_IRQHandler\n");<#
    claim R2 {
        #>printf("TIMER0_R2_claimed\n");<#
        claim R1 {
            #>printf("TIMER0_R1_claimed\n");<#
        }
    }
}
```

Listing 2. `Example.core` with potential deadlock.

```
// Example.core compiled Thu Jul 24 13:53:32 CEST 2014 for RTFM-RT
enum resources {R2,R1,RES_NR};
char* res_names[] = {"R2","R1"};
int ceilings[RES_NR] = {2, 2};
enum entry_nr {user_reset_nr, task_nr, TIMER0_IRQHandler_nr, ENTRY_NR};
int entry_prio[] = {0, 1, 2};
void task(int);
void TIMER0_IRQHandler(int);
ENTRY_FUNC entry_func[] = {user_reset, task, TIMER0_IRQHandler};
char* entry_names[] = {"user_reset", "task", "TIMER0_IRQHandler"};
// RTFM-Application
void user_reset(int RTFM_id) {
    RTFM_pend(RTFM_id, task_nr);
    RTFM_pend(RTFM_id, task_nr);
    RTFM_pend(RTFM_id, TIMER0_IRQHandler_nr);
}
void func(int RTFM_id) {
    RTFM_lock(RTFM_id, R1);
    printf("func_R1_claimed\n");
    RTFM_lock(RTFM_id, R2);
    printf("func_R2_claimed\n");
    RTFM_unlock(RTFM_id, R2);
    RTFM_unlock(RTFM_id, R1);
}
void task(int RTFM_id) {
    printf("task\n");
    func(RTFM_id);
}
void TIMER0_IRQHandler(int RTFM_id) {
    printf("TIMER0_IRQHandler\n");
    RTFM_lock(RTFM_id, R2);
    printf("TIMER0_IRQHandler_R2_claimed\n");
    RTFM_lock(RTFM_id, R1);
    printf("TIMER0_IRQHandler_R1_claimed\n");
    RTFM_unlock(RTFM_id, R1);
    RTFM_unlock(RTFM_id, R2);
}
```

Listing 3. `Example.c` targeted to RTFM-RT.

```
// Example.core compiled Thu Jul 24 13:59:52 CEST 2014 for RTFM-KERNEL
#define H(x) (32-x)
#define R2 2
#define R1 2
#define IRQ_NR_task 0
#define IRQ_NR_TIMER0_IRQHandler 1
#define IRQ_PRI_task 1
#define IRQ_PRI_TIMER0_IRQHandler 2
// RTFM-Application
void RTFM_init_priorities() {
    RTFM_set_priority(IRQ_NR_task, IRQ_PRI_task);
    RTFM_enable_irq(IRQ_NR_task);
    RTFM_set_priority(IRQ_NR_TIMER0_IRQHandler, IRQ_PRI_TIMER0_IRQHandler);
    RTFM_enable_irq(IRQ_NR_TIMER0_IRQHandler);
}
void user_reset() {
    RTFM_pend(IRQ_NR_task);
    RTFM_pend(IRQ_NR_task);
    RTFM_pend(IRQ_NR_TIMER0_IRQHandler);
}
// Generated code same as for RTFM-RT, besides:
// - handlers for Tasks/IRQs are without arguments,
// - RTFM-kernel macros and maps priorities through H(x), and
// - RTFM-kernel macros does not feature sender id.
```

Listing 4. `Example.c` targeted to RTFM-kernel.

IV. Related Work

In context of light-weight operating systems targeting CPS/IoT, the majority undertakes a thread based approach to concurrency, e.g., FreeRTOS [3], ChibiOS[4], and RIOT[5]. In such settings, real-time requirements, adds to the general problem of thread based programming, hence, the programmer may need to step outside the given concurrency model by manually managing timing critical actions directly in the interrupt handlers. However, utter care has to be taken not to violate the integrity of the system in such cases. In comparison RTFM-core and the RTFM-kernel provides superior performance to the aforementioned thread based approaches, with CPU and memory overhead on par with manual bare metal implementations [2]. Moreover, in general the thread based approaches offer little or no help in verifying real-time/resource properties, forcing the programmer to tedious and error prone testing. RTFM-core and the SRP based scheduling of the RTFM-kernel, provides an excellent outset for formal verification, establishing real-time and resource properties at compile time. The SLOTH kernel [6] undertakes a similar approach to utilizing the interrupt hardware for efficient scheduling. However, it is based on the OSEK programming model targeting mainly the AUTOSAR community with a focus on safety critical applications. RTFM-core and the RTFM-kernel on the other hand strives to bring Real-Time to The Masses (RTFM), and has the goal facilitating embedded development and concurrent programming in general.

In the context of managed environments, we have shown that POSIX Pthreads[7], and Win32 threads[8], can be used as vehicles for implementing RTFM-RT onto multi-core /multi-CPU platforms. Under POSIX, RTFM-RT utilizes Pthreads to implement immediate ceiling protocol, in order to minimize priority inversion. As being a lock based solution, it is plagued with potential deadlocks [9]. In contrast to native Pthreads, and most commonly available threading approaches, e.g. boost [10] , Ada, [11] and Java Threads [12], the `rtfm-core` compiler natively provides design time analysis, and visual feedback to facilitate the programmer in detecting and addressing potential deadlocks, once and for all! This way, both the run-time system and the application, can be made completely free from complex deadlock detection and handling mechanisms.

In contrast to other concurrency extensions of the C/C++ language (e.g., Concurrent C / Real-Time Concurrent C[13], Real-Time C[14]), our model is static and declarative, directly in terms of *task* and *resources* building directly on the notions used in context of static schedulability analysis. TinyOS [15] provides through the NesC programming language a static/declarative model, however, TinyOS tasks are executed

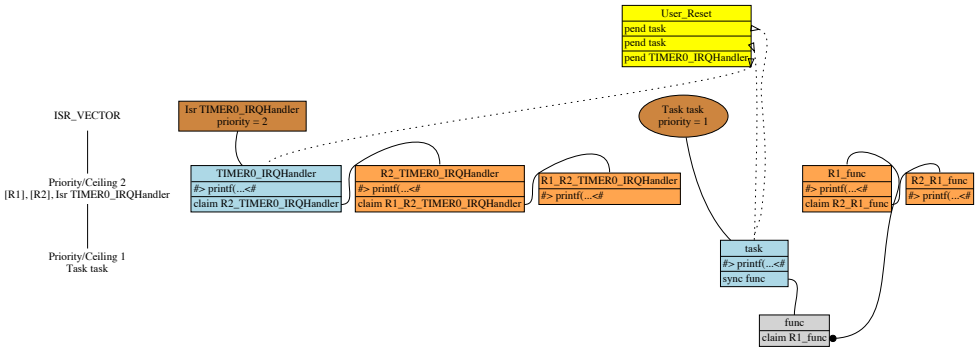

Figure 3. Task and resource view, produced by the `[-gv_task]` compiler option for the example application.

non-preemptively which largely limits real-time scheduling, whereas RTMF tasks are fully concurrent.

In a broader perspective, we find other concurrency and real-time programming models. E.g., with a heritage to functional languages Erlang [16], Atom [17], Tower/Ivory [18] and Timber [19] have been developed. While functional languages brings the benefit of pure computations (being free of side effects) and thus facilitates program analysis, the functional programming paradigm has not reached to the mainstream of embedded programming. Synchronous languages (e.g., Esterel, Signal, Lustre [20]) offers implicit concurrency and resource protection, however the programming paradigm has only reached uptake in safety critical applications.

V. Conclusions

In this paper we presented the RTFM-core language, the `rtfm-core` compiler and a set of supporting run-time systems. The RTFM-kernel offers SRP based, CPU and resource efficient execution onto light-weight platforms. SRP brings benefits of deadlock free scheduling, single stack execution and a plethora of methods for static (offline) analysis (for estimating response times and memory utilisation etc.). Moreover, we have presented the RTFM-RT run-time system, which exploits parallelism of hosting multi-core/multi-cpu systems running OSX/Linux/Win32. Yet simplistic, the presented language and supporting tools offer a complete solution to the primary challenges of real-time and parallel programming for static systems with shared resources. While programs can be written directly in RTFM-core, it offers a succinct and semantically well-grounded outset for being used as an intermediate representation for model based design and high-level languages.

Acknowledgements

This work was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology), and the EU ARTEMIS JU funding, within project ARTEMIS/0001/2013, JU grant nr. 621429 (EMC2) and VINNOVA (Swedish Governmental Agency for Innovation Systems) and Svenska Kraftnät (Swedish national grid).

References

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

[2] J. Eriksson, F. Haggstrom, 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. [Online]. Available: http://dblp.uni-trier.de/db/conf/sies/sies2013.html

[3] FreeRTOS. (webpage) Last accessed 2014-07-28. [Online]. Available: http://www.freertos.org

[4] ChibiOS/RT. (webpage) Last accessed 2014-07-28. [Online]. Available: http://www.chibios.org

[5] RIOT. (webpage) Last accessed 2014-07-28. [Online]. Available: http://riot-os.org

[6] Sloth: A Minimal-Effort Kernel for Embedded Systems. (webpage) Last accessed 2014-07-28. [Online]. Available: https://www4.cs.fau.de/Research/Sloth/

[7] IEEE-The Open Group: POSIX Programmer's Manual. (webpage) Last accessed 2014-07-28. [Online]. Available: http://www.unix.com/man-page/posix/1/man/

[8] Multithreading with C and Win32. (webpage) Last accessed 2014-07-28. [Online]. Available: http://msdn.microsoft.com/en-us/library/y6h8hye8.aspx

[9] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, May 2006. [Online]. Available: http://dx.doi.org/10.1109/MC.2006.180

[10] Boost C++ Libraries. (webpage) Last accessed 2014-07-28. [Online]. Available: http://www.boost.org

[11] Ada Resource Association. (webpage) Last accessed 2014-07-28. [Online]. Available: http://www.adaic.org

[12] Class Thread (Java Platform SE 7). (webpage) Last accessed 2014-07-28. [Online]. Available: http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

[13] N. Gehani and K. Ramamritham, "Real-time concurrent c: A language for programming dynamic real-time systems," *Real-Time Systems*, vol. 3, no. 4, pp. 377–405, 1991. [Online]. Available: http://dx.doi.org/10.1007/BF00365999

[14] V. Wolfe, S. Davidson, and I. Lee, "Rtc: language support for real-time concurrency," in *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, Dec 1991, pp. 43–52.

[15] P. Levis, S. Madden, J. Polastre, R. Szewczyk, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, "TinyOS: An operating system for sensor networks," in *in Ambient Intelligence*. Springer Verlag, 2004.

[16] J. Armstrong, *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[17] H. Zedan, A. Cau, Z. Chen, and H. Yang, "ATOM: an object-based formal method for real-time systems," *Ann. Software Eng.*, vol. 7, pp. 235–256, 1999. [Online]. Available: http://dx.doi.org/10.1023/A:1018942406449

[18] Glois Inc. (2015, Jan.) Ivory Language. [Online]. Available: http://ivorylang.org

[19] The Timber Language. (webpage) Last accessed 2011-04-15. [Online]. Available: http://www.timber-lang.org

[20] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, Robert, and D. Simone, "The synchronous languages 12 years later," in *Proceedings of The IEEE*, 2003, pp. 64–83.