Real-Time For the Masses, Step 1: Programming API and Static Priority SRP Kernel Primitives

Johan Eriksson Fredrik Häggström Simon Aittamaa Andrey Kruglyak Per Lindgren
EISLAB, Luleå University of Technology, 97187 Luleå
Email: {Johan.Eriksson, Fredrik.Haggstrom, Simon.Aittamaa, Andrey.Kruglyak, Per.Lindgren}@ltu.se

***Abstract*—Lightweight Real-Time Operating Systems have gained widespread use in implementing embedded software on lightweight nodes. However, bare metal solutions are chosen, e.g., when the reactive (interrupt-driven) paradigm better matches the programmer's intent, when the OS features are not needed, or when the OS overhead is deemed too large. Moreover, other approaches are used when real-time guarantees are required. Establishing real-time and resource guarantees typically requires expert knowledge in the field, as no turn-key solutions are available to the masses.**

In this paper we set out to bridge the gap between bare metal solutions and traditional Real-Time OS paradigms. Our goal is to meet the intuition of the programmer and at the same time provide a resource-efficient (w.r.t. CPU and memory) implementation with established properties, such as bounded memory usage and guaranteed response times. We outline a roadmap for Real-Time For the Masses (RTFM) and report on the first step: an intuitive, platform-independent programming API backed by an efficient Stack Resource Policy-based scheduler and a tool for kernel configuration and basic resource and timing analysis.

I. Introduction

Resource constrained platforms such as the ARM-7 and ARM Cortex Mx/Rx architectures have gained widespread use for cost, size and power efficient implementations of embedded real-time applications. The Artemis Research Agenda predicts the number of embedded processors to be over 40 billion by 2020 [1]; their functionality is to a large extent dependent on embedded software. Unlike programs for general-purpose processors, embedded software typically expresses interaction with internal and external peripherals and typically operates under resource and timing constraints.

Lightweight Real-Time Operating Systems have gained widespread use in implementing embedded software on such platforms. A prominent example thereof is FreeRTOS [2], whose success is (arguably) due to its familiar API, documentation, wide platform support, mature/stable code base, open source licence etc. However,

1) bare metal solutions are chosen, e.g.,
   - when the reactive (interrupt-driven) paradigm better matches the programmer's intent,
   - when the OS features are not needed, or
   - whenever the OS overhead is deemed too large;
2) other approaches are used when real-time guarantees are required, e.g., [3] and [4].

Co-funded by EU-ERDF, 978-1-4799-0658-1/13/$31.00 2013 IEEE

Establishing real-time and resource guarantees typically requires expert knowledge in the field, as no turn-key solutions are available to the masses.

In this paper we set out to bridge the gap between bare metal solutions and traditional Real-Time OS paradigms. Our goal is to meet the intuition of the programmer and to provide a resource-efficient (w.r.t. CPU and memory) implementation with established properties, such as bounded memory usage and guaranteed response times. We outline a roadmap for Real-Time For the Masses (RTFM) and report on the first step: an intuitive programming API backed by an efficient scheduler suitable for resource and timing analysis.

Our approach is based on the reactive programming paradigm, where run-to-completion jobs are triggered either from the system's environment or programmatically inside the system. Since such systems are inherently concurrent, critical sections are typically introduced to avoid race conditions on shared variables or other system resources. In the context of real-time applications, reactions to events are typically associated with priorities or timing constraints expressed in terms of deadlines.

By associating events to interrupts and implementing the corresponding reaction (job) directly in the respective interrupt handler (ISR), reactive (event/interrupt-driven) systems can be straightforwardly implemented and efficiently scheduled directly by the hardware. However, in order to support shared resources between jobs we need to adopt a resource management protocol. We choose the Stack Resource Policy (SRP) [5], for which we can construct an efficient implementation that exploit commonplace interrupt hardware. Moreover, SRP brings additional benefits of deadlock-free execution on a single stack, bounded priority inversion, etc. SRP has been extensively studied over the last decades and a rich set of methods for system analysis have been developed.

In this paper, the programming model is restricted to SRP with single-unit resources, static priorities and sporadic tasks (deadline $<$ inter-arrival time). These restrictions allow us to develop a set of kernel primitives that utilise the underlying interrupt hardware for static priority preemptive scheduling under SRP. In particular, we show that the job request and admission mechanisms for SRP can be handled with zero overhead on common micro-controllers that support nested, priority-based interrupt handling.

Furthermore, we report on the development of KCC (Kernel Configuration Compiler), a tool to automatically derive resource ceilings and to synthesise a target- and application-

specific kernel configuration from an XML system model (derived from a C program). Additionally, the tool performs basic stack depth analysis and, in order to determine schedulability, response time analysis given inter-arrival and worst-case execution times for jobs and critical sections.

We characterise the kernel primitives for the market-dominating ARM Cortex M0/M3 architectures and find that job and resource requests introduce only a few bytes of memory overhead, and the CPU overhead is in the submicrosecond range even for instep M0-MCUs like NXP/LPC11. In comparison, overhead is less than a tenth of the eventing mechanism in FreeRTOS; in fact, the proposed scheduler is hard to beat even by means of manual coding.

II. Real-Time For the Masses, Roadmap

With the ambition to facilitate real-time programming for the masses outside the relatively small and domain-specific communities, we propose the following steps:

Step 1 Basic Kernel Primitives
- Choosing a suitable task model for RTFM. We choose the task model of SRP with jobs and resources.
- Specifying an intuitive programming API to RTFM kernels allowing both direct application development and the use of RTFM as a micro-kernel for other OS/RTOS. To this end, we propose a platform-independent C code API, compatible with gcc-based tool chains.
- An efficient implementation of kernel primitives for commonplace hardware. Here we present kernel primitives for static priority SRP scheduling for ARM Cortex M0/M3.
- A tool for target- and application-specific kernel configuration with support for a basic resource and schedulability analysis. Here we present the KCC tool, supporting response time and basic stack depth analysis, producing kernel configurations for ARM Cortex M0/M3.

Step 2 Infrastructure
- Automatic generation of XML system models from C programs.
- Implementing virtual interrupt sources necessary to support platforms with insufficient hardware capabilities.
- Implementing support for job requests with arguments (messages) and virtual timers (for postponed messages).
- Extending basic system analysis to exploit additional information, such as task offsets.

Step 3 Other Programming Models and True Parallelism
- Mapping of more elaborate programming models to RTFM. These include TinyTimber [6], the REKO framework [7], and TinyOS [8].
- Investigating other scheduling approaches for RTFM, including dynamic priority SRP, hierarchical SRP, and M-SRP for multicore systems.
- Extending basic program analysis and kernel configuration to support the additional programming models and scheduling approaches.

The presented approach is influenced and motivated by previous work on Concurrent Reactive Objects and their implementation in TinyTimber [6]. The first step covers scheduling primitives with a simple C code API and basic tools. The second step further facilitates program development and increases the applicability of RTFM. Support for virtual timers and job requests with arguments forms the necessary infrastructure for mapping more elaborate programming models to the RTFM kernel, offering further abstractions and implementing OS-like services.

Other lightweight approaches include EDFI [9]. The EDFI approach is similar to ours in that it adopts the notions of SRP. However, it relies on (dynamic) EDF-based scheduling, which precludes exploiting static hardware priorities for efficient scheduling. Another approach, combining non-preempting background tasks with interrupts, is used in TinyOS [8], which allows for a bounded stack depth. However, TinyOS provides no native real-time support, and hence the performance of a system relies entirely on the programmer's ability to split tasks into sufficiently short sections. In [10], a method to preemptively execute native TinyOS tasks is suggested. This method could be deployed using the proposed RTFM SRP kernel and is a subject of future work. Yet another mechanism for single-stack execution is proto-threads, used in e.g. Contiki [11]. However, there is no notion of timing or priorities for proto-threads, hence no native real-time support.

III. Task Model

We adopt the task model used by Baker [5]:

- J A job J is a finite sequence of instructions to be executed on a single processor.
- $\mathcal{J}$ $\mathcal{J}$ denotes both a job execution request and its execution.
- $p(\mathcal{J})$ Defines the (base) priority of $\mathcal{J}$. $p(\mathcal{J}') > p(\mathcal{J})$ indicates that expediting $\mathcal{J}'$ is sufficiently important that completion of $\mathcal{J}$ is permitted to be delayed.
- $\pi(J)$ Defines the preemption level of a job, defined so that a job J' may preempt J only if $\pi(J') > \pi(J)$.
- R A nonpreemptable resource R can be claimed by a job for the execution of a critical section.

We restrict the SRP model from [5] to single-unit resources. Following the abstract resource ceiling definition [5], we define:

- $\lceil R \rceil$ The (static) current ceiling of resource R,

$$\lceil R \rceil \ = max(\{0\} \cup \{\pi(J) | J \in L(R)\})$$

  where $L(R)$ is the set of jobs that (may) request R.
- Π The (dynamic) current system ceiling,

$$\Pi = max(\{0\} \cup \{\pi(J)\} \cup \{\lceil R \rceil | R \in R_{claimed}\})$$

  where J is the currently executing job (if any), and $R_{claimed}$ the set of currently claimed (outstanding) resources.

Under SRP a job execution request for J is blocked until $p(\mathcal{J})$ has the highest priority of all outstanding jobs execution requests and $\Pi < \pi(J)$.

IV. Programmer's API

We provide a basic C code API for RTFM, an example is shown in Listing 2. The task model is captured by:

- J A job J corresponds to a C code function (without arguments and return values), e.g., **void** j1_Code() { ... }. For analysis we require functions to have bound execution time and stack requirement. (Notice, this does not preclude the use of recursive functions as long as the stack depth requirement can be determined.)
- $\mathcal{J}$ Execution requests originate either from the environment (see below) or posted programmatically within the system, e.g., JOB_REQUEST(j1). The priority for all job execution request to J is given by a define, e.g., **#define** j1_Prio 1, Listing 1.
- R Critical sections for the (nested) single-unit resources can achieved by lock/unlock, as shown in j1_Code, or alternatively by CLAIM(r1, { ... }), as shown in j2_Code. The latter enforces the LIFO nesting of resource requests as required by SRP.

Additionally, to enable hardware based scheduling, each task (job), is bound to a corresponding ISR. The API provides:

- $\mathcal{J}$ The mapping from an interrupt source to a job request is given by a define, e.g., **#define** j1_IRQn EINT1_IRQn. The macro BIND_SOURCE_TO_JOB_REQUEST(J, ISR), installs the function (job) as an ISR according to the given mapping.
- $Reset$ Program code run at startup/reset, enabling interrupts, initializing interrupt priorities etc. For purpose of analysis, $Reset$ can be seen as a (separate) one-shot job.

And finally for SRP based scheduling, the API provides:

- $\lceil R \rceil$ The (current) resource ceiling is given by a define, **#define** r1_Ceiling 2.

Listing 2 gives an example for the LPC11x (ARM Cortex M0), defining two jobs (j1 and j2) and their priorities (1, 2), 2 being higher. Since both jobs request $r1$, the resource ceiling is 2.

Listing 1. Example configuration on LPC11x (ARM Cortex M0): config.h

```
#define SOURCE_MASKING
#define H(x) (3-x)

#define j1_IRQn EINT1_IRQn
#define j2_IRQn EINT2_IRQn
#define j1_Prio 1
#define j2_Prio 2
#define r1_Ceiling 2

#define JP1 (1 << IRQn(j1))     /* set of jobs at priority 1 */
#define JP2 (1 << IRQn(j2))     /* set of jobs at priority 2 */
#define JP3 (0)                 /* set of jobs at priority 3 */
#define JP4 (0)                 /* set of jobs at priority 4 */
```

Listing 2. User program: program.c

```
void j1_Code(void) {
        LOCK(r1);
                /* Critical section j1, r1 */
        UNLOCK(r1);
}

void j2_Code(void) {
        JOB_REQUEST(j1); /* job execution request for j1 */
        CLAIM(r1, {
                /* Critical section j2, r1 */
        });
}

BIND_SOURCE_TO_JOB_REQUEST(j1, EINT1_IRQHandler);
BIND_SOURCE_TO_JOB_REQUEST(j2, EINT2_IRQHandler);

void user_reset(void) {
        /* optional user startup code here, run before system started */
        SETPRIO(j1); SETPRIO(j2);       /* Set HW priorities*/
        ENABLE(j1); ENABLE(j2);         /* Enable HW sources  */
}
```

V. Static priority SRP Scheduling

We assign static preemption levels $\pi(J) = p(J)$, hence we may use π and p interchangeably in the following. We assume that the hardware supports prioritised interrupt nesting, which (as described below) allow us to view the interrupt hardware as a preemptive static priority scheduler for our system. For the discussion, we exemplify the kernel primitives (in C/assembler) by notions of the ARM Cortex Mx architectures, (the kernel primitives can be devised similarly for other architectures).

We define a mapping H from priorities p to interrupt priorities $H(p)$, where a higher p gives a higher priority on the underlying hardware (typically 0 is the highest hardware priority). We make the assumption that the number of priorities and ISR vectors (interrupt sources) are sufficient.

Listing 3. Kernel Primitives for ARM Cortex Mx

```
#define IRQn(J)         (J##_IRQn)

#include "config.h"

#define Ceiling(R)              R##_Ceiling
#define Code(J)                 J##_Code
#define BARRIER_LOCK()          { asm volatile("dsb\n" "isb\n" ::: "memory"); }
#define BARRIER_UNLOCK()        { asm volatile("" ::: "memory"); }
#define JOB_REQUEST(J)          { NVIC->ISPR[0] = (1 << IRQn(J)); }

#ifdef SOURCE_MASKING
        int LockMask[5] = {
                ~(JP1 | JP2 | JP3 | JP4),               /* prio 0, idle */
                ~(JP2 | JP3 | JP4 ),                    /* prio 1 */
                ~(JP3 | JP4),                           /* prio 2 */
                ~(JP4),                                 /* prio 3 */
                ~(0)                                    /* prio 4, highest */
        };

        #define LOCK(R) {                               \
                int old_en = NVIC->ICER[0] ;            \
                NVIC->ICER[0] = LockMask[Ceiling(R)];   \
                BARRIER_LOCK();

        #define UNLOCK(R)                               \
                BARRIER_UNLOCK();                       \
                NVIC->ISER[0] = old_en;                 \
        }
#endif

#ifdef GLOBAL_PRIORITY
        #define LOCK(R) {                                                       \
                int sc_old = __get_BASEPRI();                                   \
                __set_BASEPRI_MAX(H(Ceiling(R))<<(8-__NVIC_PRIO_BITS));         \
                BARRIER_LOCK();
        #define UNLOCK(R)                                                       \
                BARRIER_UNLOCK();                                               \
                __set_BASEPRI(sc_old);                                          \
        }
#endif

#define CLAIM(R,CODE) { LOCK(R); {CODE}; UNLOCK(R); }

#define BIND_SOURCE_TO_JOB_REQUEST(J, ISR)  void ISR (void) { J##_Code(); }

#define ENABLE(J)       { NVIC->ISER[0] = (1 << (IRQn(J))); }
static __INLINE void SetPriority(IRQn_Type n, uint32_t p) {
        NVIC->IPR[_IP_IDX(n)] = (NVIC->IPR[_IP_IDX(n)] & ~(0xFF << _BIT_SHIFT(n))) |
        (((p << (8 - __NVIC_PRIO_BITS)) & 0xFF) << _BIT_SHIFT(n)); }
#define SETPRIO(J)      { SetPriority(IRQn(J), H(J##_Prio)); }

#include "program.c"

int main(void) {
#ifdef GLOBAL_PRIORITY
        __set_BASEPRI(H(0)<<(8-__NVIC_PRIO_BITS)); /* set to idle priority */
#endif
        user_reset();   /* user setup */
        while (1) ;     /* busy wait, can be modified for sleep mode */
}
```

A. Job request

A job execution request $\mathcal{J}$ amounts to a pending interrupt on source IRQn(J) associated with job J through the interrupt vector table. If $p(J) > \Pi$ (Π being the system ceiling) and J has the highest priority of all pending sources, the IRQn(J) interrupt is taken, and the interrupt hardware performs the sequence $\Pi_{old} = \Pi$; $\Pi = p(J)$; J executes; and when completed $\Pi = \Pi_{old}$, i.e., job admittance according to static priority SRP is natively supported by the interrupt hardware.

Interference (due to preemption) for a job J_j is defined as $sum(\{E(J_i)|p(J_i) > p(J_j)\})$ [12], where $E(J)$ is the execution time for J, under the assumption that we select the oldest highest priority job first. However, typical interrupt controllers choose the ISR on basis of the vector position, rather than the time of arrival (for reasons of hardware implementation complexity). This has the implication that the interference is in the worst case $sum(\{E(J_i)|p(J_i) \geq p(J_j), J_i \neq J_j\})$.

B. Resource request/release

Under SRP, when claiming a resource R, the system ceiling Π should be set $max(\lceil R \rceil, \Pi)$ for the duration of the critical section. We present two efficient approaches to manipulate the system ceiling.

1) Source Masking, All Cortex Mx: — An interrupt IRQn(J) may be taken only if the corresponding source is enabled. Hence, we can emulate the effect of setting $\Pi = x$ by disabling sources corresponding to jobs J with $p(J) \leq x$. Listing 3: SOURCE_MASKING gives an example implementation for the M0/M1 architecture (having 4 priority levels).

To ensure that interrupt masking has taken effect before entering the critical section, reordering directives [13] and memory and instruction barriers may be required [14]. A safe approach is to enforce barriers after (potentially) raising, and before restoring, the priority.

This approach is highly portable to architectures supporting centralized interrupt mask, (the LockMask needs to be adapted to the number of levels of the target architecture).

2) Global Priority, Cortex M3 and above: For architectures that directly support changing the base priority (global priority level) efficient implementation is straightforward. Only sources with higher priority than the current base priority can be admitted. In effect, the system ceiling for SRP scheduling is implemented directly by the hardware).

VI. RTFM Utilities

The presented RTFM API together with the static priority SRP kernel primitives is sufficient to implement reactive software onto lightweight platforms. However, using the API requires the programmer to manually derive resource ceilings and to establish resource usage and real-time properties. To this end, we have developed a Kernel Configuration Compiler (KCC) that, given an XML defining the target architecture and chosen scheduler, the jobs, their resource requests, and environment bindings, produces a target-/application-specific kernel configuration. Furthermore, given WCET for jobs and critical sections, deadlines and inter-arrival times, the tool performs a schedulabilty test through basic SRP response time analysis and a basic, yet safe, stack memory analysis. The KCC will be described in a forthcoming paper.

TABLE I
Evaluation, SM (Source Masking), GP (Global Priority)

	FreeRTOS M3	SM M0	GP $\geq$ M3
Job Latency/Job OH	650/1522 (best case)	20/40 (incl. barriers)	10/20 (incl. barriers)
Lock OH/Unlock OH	260/170 (best case)	20/20 (incl. barriers)	10/10 (incl. barriesr)
Critical Section ID	40 (constant)	-	-
Memory Job	468 (per task)	4 per nested resource request	4 per nested resource request
Static Mem	76 (per queue)	4+4 per prio	0
Footprint Program	8184	732	720
Native Analysis	None	Basic	Basic

VII. Evaluation and Future Work

The kernel primitives have been implemented for the NXP LPC11c24 (M0) / LPC1769 (M3) platforms and tested with both *gcc* v4.6 and v4.7. The implementation is expected to work for all other M0/M3 MCU's, hence a large portion of the embedded market is already covered. Table I shows a preliminary evaluation of CPU usage (in cycles) and memory overhead (in bytes). Devices operate at maximum speed of 48 and 120 MHz, respectively. The test programs were compiled using *gcc* with the option -Os. Job Latency/Job OH denotes the cost of invoking a higher priority job from a lower, Lock OH/Unlock OH reports the overhead for resource management, while Critical Section ID reports the number of cycles with disabled interrupts used for resource management. The footprint excludes the CMSIS SystemInit (368 bytes). RTFM overhead is constant and hard to beat even with manual coding and amounts only to the necessary and sufficient protection mechanisms presented.

References

[1] "ARTEMIS Strategic Research Agenda 2011," ISBN/EAN 978-90-817213-1-8f, April 2011.

[2] R. Barry, *FreeRTOS reference manual: API functions and configuration options*. Real Time Engineers Limited, 2009.

[3] K. Hänninen, J. Mäki-Turja, M. Bohlin, J. Carlson, and M. Nolin, "Analysing stack usage in preemptive shared stack systems," Mälardalen University, Technical Report ISSN 1404-3041 ISRN MDH-MRTC-202/2006-1-SE, July 2006. [Online]. Available: http://www.mrtc.mdh.se/index.php?choice=publications&id=1136

[4] J. Maki-Turja and M. Nolin, "Fast and tight response-times for tasks with offsets," in *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, ser. ECRTS '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 127–136. [Online]. Available: http://dx.doi.org/10.1109/ECRTS.2005.15

[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] J. Eriksson, "Embedded real-time software using TinyTimber : reactive objects in C," Licentiate Thesis, Luleå University of Technology, 2007. [Online]. Available: http://epubl.ltu.se/1402-1757/2007/72/LTU-LIC-0772-SE.pdf

[7] J. Wiklander, "A reactive approach to component-based design of resource-constrained embedded systems," PhD Thesis, Luleå University of Technology, 2011.

[8] 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.

[9] P. G. Jansen, S. J. Mullender, P. J. Havinga, and H. Scholten, "Lightweight edf scheduling with deadline inheritance," 2003. [Online]. Available: http://doc.utwente.nl/41399/

[10] P. Lindgren, H. Mäkitaavola, J. Eriksson, and J. Eliasson, "Leveraging TinyOS for integration in Process Automation and Control Systems," in *IECON 2012 : 38th Annual Conference of the IEEE Industrial Electronics Society*, 2012.

[11] A. Dunkels, B. Grnvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA, Nov. 2004. [Online]. Available: http://dunkels.com/adam/dunkels04contiki.pdf

[12] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, pp. 284–292, 1993.

[13] GCC,Volatiles. (webpage). [Online]. Available: http://gcc.gnu.org/onlinedocs/gcc/Volatiles.html

[14] "ARM CortexTM-M Programming Guide to Memory Barrier Instructions," ARM, infocenter DAI0321, 2012.