TinyTimber, Reactive Objects in C for Real-Time Embedded Systems

Per Lindgren Johan Eriksson Simon Aittamaa Johan Nordlander
EISLAB, Luleå University of Technology, 97187 Luleå
Email: {Per.Lindgren, Johan.Eriksson, Simon.Aittamaa, Johan.Nordlander}@ltu.se

Abstract

Embedded systems are often operating under hard real-time constraints. Such systems are naturally described as time-bound reactions to external events, a point of view made manifest in the high-level programming and systems modeling language Timber. In this paper we demonstrate how the Timber semantics for parallel reactive objects translates to embedded real-time programming in C. This is accomplished through the use of a minimalistic Timber Run-Time system, TinyTimber (TT). The TT kernel ensures state integrity, and performs scheduling of events based on given time-bounds in compliance with the Timber semantics. In this way, we avoid the volatile task of explicitly coding parallelism in terms of processes/threads/semaphores/monitors, and side-step the delicate task to encode time-bounds into priorities.

In this paper, the TT kernel design is presented and performance metrics are presented for a number of representative embedded platforms, ranging from small 8-bit to more potent 32-bit micro controllers. The resulting system runs on bare metal, completely free of references to external code (even C-lib) which provides a solid basis for further analysis. In comparison to a traditional thread based real-time operating system for embedded applications (FreeRTOS), TT has tighter timing performance and considerably lower code complexity. In conclusion, TinyTimber is a viable alternative for implementing embedded real-time applications in C today.

1 Introduction

The ever increasing complexity of embedded systems operating under hard real-time constraints, sets new demands on rigorous system design and validation methodologies. Furthermore, scheduling for real-time embedded systems is known to be very challenging, mainly because the lack of tools that are able to extract the necessary scheduling information from the specification at different levels of abstraction [12]. However, in many cases, such embedded systems are naturally described as (chains of) time-bound reactions to external events, a view supported natively in the high-level programming and systems modeling language Timber in the form of reactive objects. These time bounds can be used directly as basis for both offline system analysis and during run-time scheduling. In contrast to *synchronous* reactive objects [8, 7] and synchronous languages [5], Timber inherently captures sporadic events, thus provides a more general approach to reactive system modelling. The engineering perspectives of the Timber design paradigm are further elaborated in [14, 11, 13].

In this paper, we present *TinyTimber* (TT) - a minimalistic, portable real-time kernel with predictable memory and timing behavior - and we demonstrate how the Timber semantics translates to embedded real-time programming in C. Through the TT implementation, developers are provided a C interface to a minimalistic Timber Run-Time system, allowing C code to be executed under the reactive design paradigm of Timber (section 2). The TT kernel features the following subset of Timber;

- Concurrent, state protected objects
- Synchronous and asynchronous messages
- Deadline scheduling

In the design of the TT kernel, utmost care has been taken in order to offer bounded memory and timing behavior that is controllable by compile time parameters. The kernel itself is minimalistic, consisting solely of an event queue manager together with a real-time scheduler, and does neither rely on dynamic heap based memory accesses nor additional libraries. Thus, the functionality of the kernel can be made free of dependencies on third party code (even C-lib if so wished), which in turn benefits portability and robustness. Furthermore, the core functionality of the kernel is implemented in ANSI-C.

In the context of other minimalistic operating systems and kernels such as TinyOS [3], Contiki [10], FreeRTOS [2], and AmbientRT [1], TT stands out with its deadline-driven scheduling and the heritage to the reactive object paradigm of Timber. While TinyOS and Contiki lack native real-time support, FreeRTOS provides pre-emptive scheduling based on task priorities in a traditional fashion, and AmbientRT undertakes dynamic task scheduling based on most urgent deadlines, similar to our approach. However, TT differs fundamentally to AmbientRT in terms of our object-based locking mechanism, which allows true parallelism and hence may improve on schedulability and scalability to SRP like approaches [4]. Furthermore, TT provides best effort EDF scheduling with online resource management.

Based on experimental measurements carried out on a set of representative embedded platforms (PIC18, AVR5, MSP430 and ARM7), we show that the TT kernel can implement Timber semantics with high timing accuracy and low memory overhead. Furthermore, we compare the TT kernel to a thread based real-time operating system for embedded applications (FreeRTOS), and our experimental results verify that TT provides tighter timing performance, while matching resource requirements and having considerably lower code complexity. In conclusion, this paper shows that TinyTimber is a viable alternative for implementing real-time applications in C on embedded platforms.

978-3-9810801-3-1/DATE08 © 2008 EDAA

2 Reactive Objects in Timber

In this section we briefly overview Timber in order to introduce the concepts that are most relevant to the rest of the paper. For an in depth description, we refer to the draft language report [6], the formal semantics definition [9], and previous work on reactive objects [15, 16] and functional languages [17].

Timber seamlessly integrates the following concepts; concurrent objects with state protection, deadline scheduling of synchronous and asynchronous messages, higher-order functions, referential transparency, automatic memory management, and static type safety, with subtyping, parametric polymorphism and overloading.

However, it is the notion of *reactivity* that gives Timber its characteristic flavor. In effect: Timber methods never *block* for events, they are *invoked* by them. Timber unifies concurrent and object-orientated paradigms by its concept of concurrent state-protected objects (resources). The execution model of Timber ensures mutual exclusion between the methods of an object instance. This way Timber conveniently captures the inherent parallelism of a system without burdening the programmer with the volatile task of explicitly coding up parallelism in terms of traditional processes/threads/semaphores/monitors, etc [14]. Designed with real-time applications in mind, the language provides a notion of timed reactions that associate each event with an absolute time-window for execution. Events are either generated by the environment (typically as interrupts, as in the case of software realizations of Timber models) or through synchronous/asynchronous message sends expressed directly in the language (not as OS primitives).

The Timber run-time model utilizes deadline scheduling directly on basis of programmer declared event information, which avoids the problem of turning deadlines into relative process priorities. In short:

- *Objects and parallelism:* The parallel and object oriented models go hand in hand. An object instance executes in parallel with the rest of the system, while the state encapsulated in the object is protected by forcing the methods of the object instance to execute under mutual exclusion. This implicit coding of parallelism and state integrity coincides with the intuition of a reactive object. Furthermore, all methods in a Timber program are non-blocking, hence a Timber system will never lack responsiveness due to intricate dependencies on events that are yet to occur.
- *Events, methods and time:* The semantics of Timber conceptually unifies events and methods in such a way that the specified constraints on the timely reaction to an event can be directly reused as run-time parameters for scheduling the corresponding method. Event *baseline* (release) and *deadline* define the permissible execution window for the corresponding method. All other points in time will be given relative to the event baseline, and will thus be free from jitter induced by the actual scheduling of methods.

3 Reactive Objects in C using TinyTimber

Over the last decades, C has become the dominating language for programming embedded systems. As the C language lacks native real-time support, concurrency and timing constraints are traditionally implemented through the use of external libraries, executing under some real-time operating system or kernel.

3.1 The TinyTimber application interface

TinyTimber (TT) allows C code to be executed under the pure reactive design paradigm of Timber. TT offers concurrent reactive objects (`Object`) with state protection as well as synchronous and asynchronous messages under deadline scheduling. Each message/event has a permissible interval for method execution (between the absolute points in time *baseline* and *deadline*). In case of synchronous messages (`SYNC(o, m, p)`), base- and deadlines are always inherited from the sender, (object `o`, method `m`, and parameter `p`). For asynchronous messages (`ASYNC(-1/b, -1/d, o, m, p)`), the new baseline can be either inherited (`-1`) or being computed as current *baseline* + `b`. However, if this new *baseline* (absolute point in time) has already passed at the time of posting, the new message *baseline* is set to current time. Respectively, the message *deadline* can be either inherited (`-1`), or set relative to the the new *baseline*. Events from external sources i.e. interrupts are (conceptually) executed with *baseline* and *deadline* set to current time.

Like its higher-level counterpart, TT assumes a run-to-end method semantic, hence no means are offered for synchronization with events by actively postponing execution within a method.

On the top level, a TT application consists of the implicit root object and its methods, i.e., the functions installed for handling interrupts and the reset signal. The state of the root object is the state of the globally declared C variables. To impose more structure, the root state can be further partitioned into objects of their own, whose methods are run under supervision by the scheduler. Method calls crossing object boundaries must never bypass the kernel primitives, otherwise any communication pattern between the objects can be devised.

A TT application is compiled and linked with the TT kernel for a target architecture using a C compiler suite such as gcc. For a bare-chip target, the executable image will not rely on additional libraries, although libraries may be incorporated as long as they do not violate the run-to-end assumption of TT.

4 The TinyTimber kernel

The primary job of the TT kernel is to manage the messages queues and schedule messages onto execution threads such that (if possible) all message deadlines are met. The TT kernel is purely event driven, hence, if there are no messages eligible for scheduling, the system is idle. The idle state may be used to put the system in low power mode. The complete source code of TT can be obtained from the authors under an open-source license.

4.1 Informal description

In the following we will discuss how the TT kernel addresses event scheduling, with respect to safety, liveness, and real-time properties [12].

These criteria are met by the TT kernel by Earliest Deadline First (EDF) scheduling with priority inheritance, together with mutual exclusion between object instance methods. Priority inheritance together with the run-to-end Timber semantics ensures

that priority inversion will be bounded. As object instance methods operate under mutual exclusion the only possible source of deadlock is circular synchronous events. During run-time, the TT kernel will report such hazards. Applications free of such circular references will be executed in a *safe* (with respect to deadlock) manner by the TT kernel. Since the kernel is internally free of blocking operations, and given the run-to-end requirement on object methods, each event will eventually be executed and *liveness* of the system is upheld. This is achieved as the kernel itself will never enter the idle state unless no events are eligible for execution, and each event will eventually become released (as there is no notion of infinite baseline in Timber). The EDF scheduling together with the bounded priority inversion provides best effort *real-time* properties.

5 Performance Measurements

In this section we characterize the current TT implementation v0.3.0 on a number of representative platforms and give a brief comparison to a traditional thread-based open source operating system (FreeRTOS).

5.1 TinyTimber Overhead

Using TT there are two means of passing messages between objects, either synchronous or asynchronous. In the case of a synchronous message the TT kernel will; acquire the object lock (mutex), call the method specified, and release the object lock. The overhead cost of performing a synchronous call to an unlocked object is shown in Table 1 (a). However, if the object is locked the kernel will force the method holding the lock to resume execution (run-to-end) with inherited priority. The priority inversion is bounded by the acyclic chain of synchronous messages needed to be completed. Priority inversion overhead stems from implicit context swithces, (c) and (f).

In the case of an asynchronous message a baseline and a deadline is assigned to a synchronous message, effectively delaying the execution of the message (b). Once the baseline of the message expires a timer interrupt is generated, the context of the current tread will be saved (c), the message will be released into the queue of active messages (d), if the released message has the earliest deadline a new thread will be allocated (e), the context of the current thread will be restored (f), and a synchronous call is performed (a). Note that the cost of releasing the messages into the active queue is dependent on the number of messages to be released and the number of messages in the active queue, Table 1 (d) shows the cost when one message is released into an empty active queue (best case).

To preserve the state integrity of the kernel interrupts are disabled/enabled upon kernel entry/exit. The instruction count for the largest critical section (interrupts disabled) arises when a baseline(s) expires and message(s) is/are released into the active queue (g). The critical section directly affects timing accuracy and responsiveness (with respect to external events).

5.2 Example Application

The desired behavior of the application is as follows; upon some external event a given output should be driven high for three milliseconds. Yet simple, the application exercises mechanisms for context switching, synchronization, and timing.

	PIC18	AVR5	MSP430	ARM7
(a) Synchronous Call	150	63	46	50
(b) Asynchronous Call	406	167	97	74
(c) Save Context	$137 + 7n$	56	18	23
(d) Release Message	228	53	50	37
(e) Allocate Thread	19	14	5	8
(f) Restore Context	$136 + 10n$	50	18	15
(g) Critical Section	884	398	213	224

Table 1. TinyTimber instruction count for a set of kernel mechanisms. All instruction counts are best case for current implementation. For PIC18, n gives the call-stack depth.

5.3 Application Implementation

The TT implementation is shown in Listing 1. The `output` object is created to encapsulate the output (pin 6.7). When the external interrupt is triggered an asynchronous message is sent to the `output` object invoking the `output_high` method. The `output_high` method will drive the output high and post an asynchronous message to the current object instance that will invoke the `output_low` method after three milliseconds.

The FreeRTOS implementation is shown in Listing 2. A single thread and semaphore is used to implement the desired behavior. Once the semaphore is released we drive the output high, delay for three milliseconds and drive the output low.

5.4 Application Comparison

To measure the response time the first channel of an oscilloscope was connected to the output, the second channel to the interrupt status, and the trigger to the external event. All measurements were performed on an idle system i.e., no other other threads/messages were running. The platform used for the measurements was a MSP430 (msp430x1611) clocked at 4.7MHz.

The delay is defined as the time between the triggering of the external event and the output driven high. The jitter is defined with respect to the pulse length, and the critical section is defined as the longest period of time interrupts are disabled. For the memory footprint of the application, flash is defined as the size of the `.text` section and RAM as the size of the `.data` and `.bss` sections.

Table 2 shows that the footprint of TT is significantly smaller, mainly due to its minimalistic API and simple implementation.

Table 3 shows that TT has a slight edge with respect to both delay and critical section. However, when it comes to timing accuracy, jitter measurements are clearly in favor of the TT implementation. FreeRTOS undertakes a system tick based timing scheme while TT uses a free running timer. The resolution of the system timer in FreeRTOS is 1kHz (by default) while the TT timer has a resolution of 32.768kHz (default on MSP430). The CPU overhead of a system tick based scheme is directly proportional to the timing resolution (where 1 kHz is a reasonable tradeoff). In the case of TT with its free running timer, the resolution is approx 30 μs (1/32768), with a measured average of 20 μs. For a system under load, the limiting factor for TT timing will be the critical section (approximately 100 μs). Hence, in practice a 10 kHz timer would be sufficient and TT could be expected to offer a tenfold improvement over FreeRTOS timing accuracy.

	TinyTimber	FreeRTOS
Flash	3544 bytes	5304 bytes
RAM	1178 bytes	1928 bytes

Table 2. Memory footprints of example applications.

	TinyTimber	FreeRTOS
Delay	$150\mu s$	$220\mu s$
Jitter	$20\mu s$	$1ms$
Critical Section	$100\mu s$	$120\mu s$

Table 3. Delay, Jitter, and Critical Section length.

Listing 1. tt.c

```
#define PULSE_WIDTH MSEC(3)

#define initOutput(width, jitter) {initObject(), width}
typedef struct output_t {
    Object obj;
    Time width;
} output_t;
static output_t output = initOutput(PULSE_WIDTH);

env_result_t output_low(output_t *self, int arg) {
    P6OUT &= ~0x80; /* Set pin 6.7 low. */
}

env_result_t output_high(output_t *self, int arg) {
    P6OUT |= 0x80; /* Set pin 6.7 high. */
    ASYNC(self->width, 0, self, output_low, 0);
}

void msp430_port1_vector(void) {
    P1IFG = 0x00;   /* Clear intrrupt flag for pin 1.7. */
    ASYNC(-1, -1, &output, output_high, 0);
}

INTERRUPT(PORT1_VECTOR, msp430_port1_vector);

static void init(void) {
    P1SEL   &= ~0x80; /* Configure pin 1.7 as digital I/O. */
    P1DIR   &= ~0x80; /* Configure pin 1.7 as input. */
    P1IE    |= 0x80;  /* Enable interrupts for pin 1.7. */
    P6DIR   = 0x80;   /* Configure pin 6.7 as output. */
}
STARTUP(init);
```

Listing 2. freertos.c

```
#define PULSE_WIDTH ((3*configTICK_RATE_HZ+999)/1000)

static xSemaphoreHandle output_semaphore;

void output_task(void *params) {
    for (;;) {
        if (xSemaphoreTake(output_semaphore, portMAX_DELAY)) {
            P6OUT |= 0x80; /* Set pin 6.7 high. */
            vTaskDelay(PULSE_WIDTH);
            P6OUT &= ~0x80; /* Set pin 6.7 low. */
        }
    }
}

interrupt(PORT1_VECTOR) msp430_port1_vector(void) {
    P1IFG = 0x00;   /* Clear intrrupt flag for pin 1.7. */
    if (xSemaphoreGiveFromISR(output_semaphore, pdFALSE))
        taskYIELD();
}

int main(void) {
    xTaskHandle output;

    P1SEL   &= ~0x80; /* Configure pin 1.7 as digital I/O. */
    P1DIR   &= ~0x80; /* Configure pin 1.7 as input. */
    P1IE    |= 0x80;  /* Enable interrupts for pin 1.7. */
    P6DIR   = 0x80;   /* Configure pin 6.7 as output. */

    vSemaphoreCreateBinary(output_semaphore);
    xTaskCreate(output_task, "output", 128, NULL, tskIDLE_PRIORITY + 1, &output);
    vTaskStartScheduler();
}
```

6 Conclusions and Future Work

Timber allows the real-time behavior of embedded systems to be modeled by means of *reactive objects*. In this paper we have demonstrated how the Timber semantics translates to embedded real-time programming in C. This is realized through the implementation of TinyTimber (TT), a C API to a minimalistic Timber Run-Time system. The TT kernel has been introduced and performance metrics have been presented for a number of representative embedded platforms. Compared to a traditional thread based real-time OS (FreeRTOS), TT is shown to excel with its simple API and high timing accuracy. Future work includes improvements of responsiveness and timing accuracy by amortizing queue management and shortening the kernel critical section. Furthermore, the applicability of TT to severely memory constrained systems may be broadened by adopting SRP based scheduling. The generation of pre-emption levels directly from the specification (Timber/C using TT) is currently under investigation.

References

[1] Ambient Systems - for low cost, low power, wireless mesh networking solutions. `http://www.ambient-systems.net/`, 2006.

[2] FreeRTOS-A Free RTOS for ARM7,ARM9,Cortex-M3,MSP430,MicroBlaze,AVR,x86,PIC18,H8S,HCS12 and 8051. `http://www.freertos.org/`, 2006.

[3] TinyOS Community Forum —— An open-source OS for the networked sensor regime. `http://www.tinyos.net/`, 2006.

[4] T. P. Baker. A Stack-Based Resource Allocation Policy for Realtime Processes. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 191–200, 1990.

[5] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.

[6] A. Black, M. Carlsson, M. Jones, R. Kieburtz, and J. Nordlander. Timber: A programming language for real-time embedded systems. Technical Report CSE-02-002, Dept. of Computer Science & Engineering, Oregon Health & Science University, April 2002.

[7] F. Boussinot, G. Doumenc, and J. Stefani. Reactive Objects. *Annals of Telecommunications*, 51(9-10):459–473, 1996.

[8] F. Boussinot and J.-F. Susini. The SugarCubes Tool Box: A Reactive Java Framework. *Software – Practice and Experience*, 28(14):1531–1550, Dec. 1998.

[9] M. Carlsson, J. Nordlander, and D. Kieburtz. The semantic layers of Timber. In *APLAS 2003*, volume 2895 of *Lecture Notes in Computer Science*. Springer, November 2003.

[10] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *First IEEE Workshop on Embedded Networked Sensors*, 2004.

[11] J. Eriksson and P. Lindgren. A Comprehensive Approach to Design of Embedded Real-time Software for Controlling Mechanical Systems. In *The 14th Asia Pacific Automotive Engineering Conference , APAC14*, 2007.

[12] H. Klapuri, J. Takala, and J. Saarinen. Safety, liveness and real-time in embedded system design. *Journal of Network and Computer Applications*, 22(2):69–89, 1999.

[13] V. Leijon, P. Lindgren, and J. Eriksson. FIFO WiDOM: Timely Control Over Wireless Links. In *IEEE Multi-conference on Systems and Control, Singapore*, 2007.

[14] P. Lindgren, J. Nordlander, and J. Eriksson. Robust Real-Time Applications in Timber. In *Sixth IEEE International Conference on Electro,Information Tech, EIT*, 2006.

[15] J. Nordlander. *Reactive Objects and Functional Programming*. Phd thesis, Department of Computer Science, Chalmers University of Technology, Gothenburg, 1999.

[16] J. Nordlander, M. Jones, M. Carlsson, D. Kieburtz, and A. Black. Reactive objects. In *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*, Arlington, VA, April 2002.

[17] J. Peterson. The Haskell Home Page. `http://haskell.org`, 1997.