Timber as an RTOS for Small Embedded Devices

Martin Kero[*] , Per Lindgren, Johan Nordlander
Luleå University of Technology
Department of Computer Science and Electrical Engineering
EISLAB
SE-97187 Luleå, Sweden
www.csee.ltu.se

ABSTRACT
Software development for small, real-time and resource constrained, embedded systems is becoming increasingly complex. To be able to guarantee robustness and reliability, the underlying infrastructure should not be based upon ad hoc solutions. In this paper we identify three key features of a minimalistic Real-Time Operating System (RTOS), and presents the run-time system of Timber, a reactive deadline-driven programming language. We scrutinize the functionalities of the run-time system in the light of real-time requirements, and emphasize the importance of integrating an adequate notion of time, both semantically in the programming interface as well as part of the run-time system.

1. BACKGROUND
Software development for small, real-time and resource constrained, embedded systems is becoming increasingly complex. In order to guarantee robustness and reliability, the underlying infrastructure should not be based upon ad hoc solutions. For this reason real-time operating system (RTOS) features for small embedded systems have gained recent interest.

We have identified a set of desired key features. A minimalistic RTOS should at least:

- supply sufficient infrastructure for reactive concurrent programming,
- preserve state integrity, and
- realize real-time constraints.

In addition, it is beneficial if the RTOS and its programming interface provides the ability of formal reasoning about system properties, which would be useful towards safe and minimal system dimensioning. It is also desirable if this can be accomplished without limiting the expressive power of the programming interface.

RTOS's in general are too heavy weighted for small embedded devices, but a few proposals have been presented. Among them are systems like Contiki[10], PicoOS[3], FreeRTOS[2], Nucleus RTOS[1], and TinyOS[12, 11, 5]. All of these systems have their unique characteristics, but we will, in short, only present the last one. TinyOS is a minimal operating system suitable for the smallest embedded devices. It is based upon a component structure and a reactive event-based concurrency model. Its core footprint is only about 500 bytes. Robustness is partly achieved by a static analyzer for data race detection. The major problem that still remains unsolved in TinyOS, as well as in all the others, is the realization of real-time constraints in run-time. To be able to semantically guarantee some degree of robustness, TinyOS has restrictions in terms of expressive power. For instance, dynamic storage allocation is not allowed.

[*]e-mail: `Martin.Kero@csee.ltu.se`

In this paper, we will present the run-time system of Timber [8, 9], a reactive deadline-driven language for embedded programming. The language it self will only be presented briefly and we will focus on the run-time system features. Readers interested in learning more about the language should read [8, 9] or visit [4]. Space does not allow a detailed side-by-side comparison of features in this paper, although such an evaluation is forthcoming. We will show that Timber offers a systematic approach to deal with real-time issues in embedded programming, unique in that the language semantics is fully reflected in the run-time system - in fact the run-time system and the application are one! This allows the executable to be fully tailored to the problem at hand, which is hard (or even impossible) to achieve under the tradional paradigm that separates the application from the operating system.

2. TIMBER - A SHORT INTRODUCTION
Timber, *TIme - eMBEdded - Reactive*, is a reactive, real-time, concurrent, object-oriented, functional programming language. It is based upon O'Haskell which in turn is an extension to Haskell [16, 15]. The development of the language is a joint effort by Luleå University of Technology, Chalmers University of Technology, and Oregon Health and Science University.

In brief, the language is based upon concurrently executing reactive objects [17]. The inter-object communication is message-based by means of synchronous and asynchronous message sends. A message send is equivalent to invoking a method of the recipient object.

Even though Timber is a general purpose language [9], it is primarily designed to target embedded systems, and we will discuss the aspects of the language in the context of embedded programming.

```
sonar (port,alarm) =
   template
      t := baseline
      ping = before (50*us) action
         port.write(beepOn)
         t := baseline
         after (2*ms) stop
         after (1*s) ping
      stop = action
         port.write(beepOff)
      echo = before (5*ms) action
         distance = k*(baseline-t)
         if (distance < limit) then
            alarm.on
      return{
         sonar = echo
         start = ping
      }
main regs =
   template
      s <- sonar ((regs!0xac00) a)
      a <- alarm (regs!0xa3f0)
      return {
         reset = s.start
         irqvector = [
            (sonarIRQ, s.sonar),
            (buttonIRQ, a.off)
             ]
      }
```

Figure 1: Example Timber program, A *Sonar*

2.1 Records and Objects

Besides primitive data types such as integers, floating point numbers, etc., Timber includes user-defined records and primitive types to support object-orientation. Records can either be used to define immutable data or to describe interfaces to objects.

Timber objects basically consists of two parts, an internal state and a communication interface. An object is instantiated by a template construct, which in turn defines the initial state of the object and its communication interface. The template construct can be seen as a module, offering an input interface and requiring an output interface, when instantiated into an object.

The primitive object-oriented types are `Action`, `Request`, and `Template`, which all are subtypes of `Cmd`. The meaning of the `Action` and `Request` types are asynchronous and synchronous message sends, respectively. These actions and requests are collectively called methods. `Template` is the type defining the template command from which objects are created.

A Timber program has to include a specific main template. The communication interface of this template is system dependent. For embedded devices the input interface usually contains a reset method and bindings from interrupts to actions. We will discuss this interface more thoroughly later on. The output interface is the environment in which the Timber program will operate. In the context of embedded devices, it shall at least provide methods to read from and write to ports.

At system startup, the main template command will be executed, creating an instance of the object main and then executing the reset method. The system will supply the main object with its environment.

2.2 Methods

A method is invoked by a message send command, either an asynchronous action or a synchronous request. A Timber program running on an embedded device can be seen as a set of concurrent objects, all awaiting external stimuli initially caused by interrupts. A method can basically do three things, it can update the state of the object, create new objects, and invoke methods of other objects. After an external stimulus, the chain of reactions will eventually fade out and the system will return to the state of waiting for new stimuli. We will refer to the time when the whole system is inactive and passive as the *idle state*, or *idle time*.

Each message in Timber has a corresponding baseline (earliest release time) and deadline attached to it. By default, a message inherits *baseline* and *deadline* from its sender but for asynchronous messages both can be adjusted by `after` and `before` constructs.

2.3 Objects as concurrent reactive processes

Objects in Timber has its own execution context, or thread of control. Inter-object communication is achieved by message-passing. Only one method within an object can be active at a time, and the object state is only accessible through its methods. This results in mutual exclusion of state mutations, usually referred to as state integrity. Furthermore, a method cannot block the object thread indefinitely which leads to a controllable responsiveness of each object.

The input interface of the main template is, as mentioned earlier, a reset method and an interrupt vector. The interrupt vector is a vector of tuples, connecting interrupt numbers to actions. This is how the environment will have the ability to trigger the reactive Timber program.

3. THE RUN-TIME SYSTEM OF TIMBER

The functionalities of the run-time system is directly reflected by the semantics of the language. The key features that needs to be facilitated in terms of functionalities of the run-time system are as follows:

Scheduling: The fundamental functionality of the run-time system to achieve concurrency between Timber objects, with scheduling based on the baselines and deadlines of their methods.

Message-passing: Supplying sufficient infrastructure for the inter-object communication.

Threading: Facilitating the unique execution contexts for Timber objects.

Time: Ability to supply sufficient time information to make baselines and deadlines meaningful.

Interrupt handling: Functionality for receiving and distributing interrupts throughout the system.

Environment interface: Implementation of the interface to the environment.

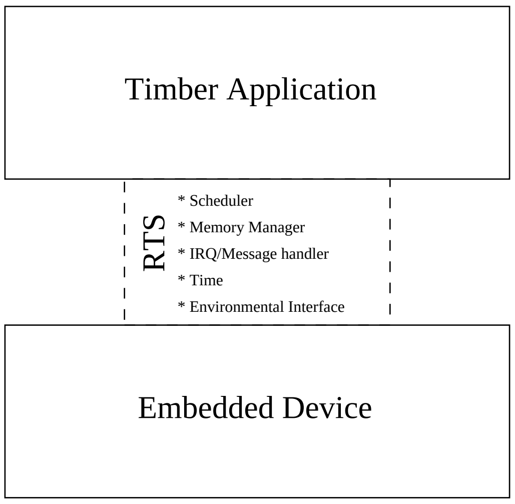

Figure 2: Timber Run-Time System overview

Automatic memory management: Timber does not rely on explicit allocations and deallocations of dynamic data and needs an automatic memory manager to serve with garbage collection.

The semantics of Timber actually does not imply a specific scheduling algorithm. It rather states the following: *Every method invoked by a message send has to be finished within the specified time-line (between its baseline and its deadline).*

The language also requires two levels of scheduling, one for messages (method invocations) within an object, and one between objects. The intra-object scheduling is non-preemptive due to state integrity, and priority inversion is solved by priority inheritance. The inter-object scheduling is preemptive, though, and realized in the run-time system by strict EDF, where the current deadline of an object is equivalent to the deadline of its most urgent message.

The message-passing mechanism in the run-time system is facilitated by message queues. Each object holds a queue of messages sorted by EDF. In addition to the local queues of each object, the run-time system also holds a queue of timed messages. A message send will insert the message into the queue of the recipient object. The corresponding method will be executed when the object has that message first in queue and is scheduled to run. Messages with a baseline ahead in time will be stored in a global queue of timed messages. This queue is sorted by earliest baseline first.

To accomplish concurrency between objects, each object needs to have its own execution context. At least they need a reference to their current execution point in the code. We will further base the context on a non-shared stack environment, adding the current stack-pointer to each context. The ability to use a shared stack environment is under investigation, mostly based on the work by Baker presented in [6, 7]. The context switching and storage is accomplished by means of non-local goto, storing the code- and stack-pointers in jump-buffers attached to each object.

A notion of time is essential for the run-time system in accomplishing correct scheduling. Baselines and deadlines are naturally expressed relatively to the actual occurrence of a message send. A straight-forward way to supply the necessary scheduling information is to calculate an absolute time from a global baseline for all message sends. This will greatly simplify the comparisons needed for scheduling, only dealing with notions of when a method has to be finished and making the time when a message send occurred necessary only locally and temporarily.

The main task of the interrupt handling mechanism is to serve as an interface between the interrupts and the message-passing mechanism. This is solved by a generic interrupt handler for hardware interrupts and a timer interrupt handler. Both of these handlers translate the interrupts into messages and post them as any other message sent by the application. This makes the handling of interrupts transparent to the rest of the system, which treats them as any other message.

The run-time system will allow context-switches to occur at three occasions. First of all, a context-switch may occur after a method is finished. This means that when the method finishes, the object will return the control back to the scheduler, which in turn will do the eventual context switch. The second case is after an interrupt has occurred and the corresponding message is sent. Both the first and the second case may cause a context-switch, but will not necessarily result in that. The third and final case will always cause a context-switch, which occurs on a synchronous request. To be able to receive the requested value, the calling object has to let the recipient object execute the method, and thus a forced context-switch will occur.

The environment interface includes the low-level implementation of time, hardware initializations of interrupts, and environment methods used by the main template. When a Timber program is in its idle state (all objects are inactive) the device is put into proper sleep-mode to lower the power-consumption. The low-level implementation of achieving this is also included in the environment interface.

The semantics of Timber is highly dependent on the ability to allocate memory storage dynamically. Furthermore, as mentioned, the language does not include any explicit allocation/deallocation commands. It is thus crucial to include a garbage collector in the run-time system.

A prototype of a reference counting garbage collector has been implemented for the run-time system of Timber [14] and currently a copying collector is under development. The work has so far shown a set of collector characteristics but due to space limitations they are only addressed in short. A thorough description of the collector is forthcoming.
1 The memory manager will reclaim garbage memory when the system is in its idle state.
2 The time needed for bookkeeping during execution of any method of the program is kept constant and small.
3 The time it takes to perform a whole garbage collection cycle will at worst be proportional to the maximum amount of simultaneously live memory.
4 The amount of contiguous free heap storage has to be

at least the maximum amount of simultaneously live memory. This however should not be misunderstood as a need for twice as much memory than a system based on explicit allocations/deallocations. It is a rather trivial fact that, in a multi threaded system, avoidance of deallocating semantically live data will most of the times result in a lot of semantically dead data to be kept alive. This is almost impossible to measure but if such a measurement would have been accomplished, it would not be surprising if it showed a factor much greater than two between the amount of data kept alive and the semantically live data for a typical embedded application.
5 The collector is incremental, that means, the application can preempt the collector with a very fine time granularity. The needed atomic operations of the collector is constant and small and no extra bookkeeping is needed during a collection cycle.

4. TIMBER IN A REAL-TIME CONTEXT

Timber objects and TinyOS Components are very similar in terms of their design motivations. Both hold the dynamic property of a state, that will be repeatedly updated during the lifetime of the system. The updates will be made by procedures associated to the object or component. Timber objects include methods whereas TinyOS components includes tasks and command/event handlers. Notions of concurrency and reactivity are also supported on both platforms. TinyOS achieves concurrency through enabling event handlers to interrupt the current executing thread of control, either a task or another handler. Timber facilitates concurrency by enabling the scheduler to intervene the execution of a method at the occurrence of an interrupt.

4.1 Semantic Characteristics

Timber is, in contrast to TinyOS, an object-oriented language that not only allows, but rather demands the ability of dynamic storage allocation due to the extensive use of immutable data. Timber allows objects to be created dynamically with very little restrictions. An object may even be created by executing a locally defined template command within a method. TinyOS does not allow dynamic storage allocation at all.

The most unique and significant feature of Timber is its direct real-time support. Timing constraints can be expressed in the source code and will migrate into meaningful scheduling data used by the scheduler at run-time.

Timber eliminates the risk of data races by means of mutual exclusion between methods that may access common mutable data. Instead of relying on the programmer to avoid data races, this protection is induced by the language semantics.

4.2 Run-Time Characteristics

The presence of time is the major characteristic that permeates the entire run-time system. Every scheduling decision is based upon timing constraints originally expressed in the source code.

Another major part of the run-time system is the memory manager. The real-time characteristics of the memory manager is mainly due to the common memory usage behavior of Timber applications. The allocator is totally predictable (incrementing a free pointer) and the collector is transparent due to the fact that it only runs during idle time. The collector is furthermore incremental with a very fine granularity and no extra bookkeeping is needed during a collection cycle.

In contrast to TinyOS, the scheduler can perform scheduling decisions at the occurrence of any interrupt, if desirable. However, in cases where the deadline of an interrupt-handler is extremely short, the run-time system may also be configured to run the handler directly, without passing through the scheduler. The semantics of the language guarantees that only the ability to meet deadlines, not the meaning of the program itself, may be affected by such a choice.

4.3 Analysis and System Dimensioning

We have not discussed how Timber enables analyses for proper system dimensioning and verification. An example of this is shown in [18] by Svensson et al., where a WCET analysis for the schedulable units of Timber programs is presented. This is the first fundamental step in performing whole system schedulability analysis. It has also been shown that the fundamental tools in WCET analysis can be applied in other analyses, such as memory usage analysis [19].

5. CONCLUSION

We have shown how the run-time system of Timber facilitates the main infrastructure for reactive concurrent real-time software development. The integration of time is consistent and meaningful throughout the whole system, from the system specifications in the source code into each scheduling decision made in run-time. The combination of reactive objects and controlled use of mutable state eliminates the risk of data races and preserves state integrity. Dynamic memory storage allocation and garbage collection relieves the programmer from the error-prone task of manual memory management. The graph of objects may also, due to the ability to create objects dynamically, change rapidly over time, resulting in a more agile system than the case where the run-time structure is static. In contrast to TinyOS, Timber does not require any limitations in expressive power to avoid race conditions, this is instead guaranteed by the language semantics. Furthermore, the core of the run-time system has a greater potency (scheduling, memory management, etc.) than the corresponding parts of TinyOS, without introducing any unsafe real-time characteristics..

Timber mainly lacks two things in comparison to other minimalistic RTOS's such as TinyOS. First of all, TinyOS is well adopted, well tested, and well understood by practitioners. It has been available for several years and many practitioners have joined in. Second of all, the heritage of TinyOS is a lot more known by general practitioners due to the well-known imperative language C [13]. We cannot even start to compare C with Haskell in terms of how many well-skilled practitioners each programming paradigm has.

The language Timber includes more features than we have been able to cover within the scope of this paper. Features like polymorphism, sub-typing, inheritance, partial application, etc. may not be fundamental for embedded real-time programming, but may prove useful also in this field. We have throughout this paper discussed the characteristics of the run-time system of Timber in the light of the common idea of how an RTOS should look like. In contrast to this point of view, the paper also points towards a rather new paradigm, to wit the importance of the programming language metaphor. The language Timber, with its concurrent object model, can actually be seen as the most fundamental part of the RTOS presented in this paper, due to the fact that all features of the run-time system is directly induced by the language semantics. This furthermore means that the run-time system is as complex (or simple) as the application needs. The programming language and the run-time system is thus not separate parts.

5.1 Future Work

Due to the fact that both the language Timber and its run-time system is still in the stage of development, many features are still missing. Garbage collection is still not fully implemented and integrated into the run-time system. Many features in the run-time system are still cumbersome and lack some functionalities. Even though the multiple message queues (one for each object) is a straightforward solution to accomplish the two level scheduling, it may not be the most efficient solution. It will be interesting to see if the two level scheduling can be accomplished by one queue solely. Another optimization that will increase the efficiency significantly in terms of memory usage is the use of a shared stack environment [6, 7], either completely shared or a hybrid solution. These optimizations will make the run-time system less unwieldy.

Due to the characteristics of the language, Timber provides ample opportunities to system analysis. It has been shown that WCET analysis can be performed on the schedulable units in Timber [18], which lays the foundation for whole system schedulability analysis. The knowledge of how WCET analysis can be performed will also be useful for the realization of memory usage analysis [19]. It will furthermore be interesting to see how the programming interface and the run-time system characteristics can render possibilities for an analysis framework including a definition of important behavioral attributes of embedded real-time software systems, and realization of tools to measure them.

6. REFERENCES

[1] Accelerated Technology official homepage, http://www.acceleratedtechnology.com/, 2005.

[2] FreeRTOS official homepage, http://www.freertos.org/, 2005.

[3] PicoOS official homepage, http://picoos.sourceforge.net/, 2005.

[4] Timber website at Luleå University of Technology, http://www.csee.ltu.se/index.php?subject=timber, 2005.

[5] TinyOS official homepage, http://www.tinyos.net/, 2005.

[6] 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.

[7] T. P. Baker. Stack-based scheduling of real-time processes. *Advances in Real-Time Systems*, pages 64–96, 1993.

[8] A. Black, M. Carlsson, M. Jones, R. Kieburtz, and J. Nordlander. Timber: A programming language for real-time embedded systems, 2002.

[9] M. Carlsson, J. Nordlander, and D. Kieburtz. The semantic layers of timber. *The First Asian Symposium on Programming Languages and Systems (APLAS), Beijing, 2003. C Springer-Verlag.*, 2003.

[10] A. Dunkels, B. Grönvall, 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, Tampa, Florida, USA*, 2004.

[11] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *In ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[12] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[13] B. W. Kernighan and D. M. Ritchie. *The C programming language*. Prentice-Hall, Inc., 1978.

[14] J. Mattsson. Garbage collection with hard real-time requirements. Master's thesis, Luleå University of Technology, 2004.

[15] J. Nordlander. *Reactive Objects and Functional Programming*. PhD thesis, Chalmers University of Technology, 1999.

[16] J. Nordlander and M. Carlsson. Reactive objects in a functional language – an escape from the evil, 1997.

[17] J. Nordlander, M. Carlsson, M. Jones, and J. Jonsson. Programming with time-constrained reactions, 2005.

[18] L. Svensson, J. Eriksson, P. Lindgren, and J. Nordlander. Language-based WCET analysis of reactive programs, 2005.

[19] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Optimized live heap bound analysis. In *Proc. 4th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 2575 of *Lecture Notes in Computer Science*, pages 70–85. Springer-Verlag, jan 2003.