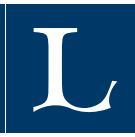
2007:72

LICENTIATE THESIS



Embedded Real-Time Software using TinyTimber - Reactive Objects in C

Johan Eriksson

Luleå University of Technology
Department of Computer Science and Electrical Engineering
EISLAB

2007:72 | ISSN: 1402-1757 | ISRN: LTU-LIC -- 07/72 -- SE

Embedded Real-Time Software using TinyTimber - Reactive Objects in C

Johan Eriksson

EISLAB
Dept. of Computer Science and Electrical Engineering
Luleå University of Technology
Luleå, Sweden

Supervisor:

Per Lindgren

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 licensiate thesis 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. The TinyTimber 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 traditional processes/threads/semaphores/monitors, and side-step the delicate task to encode time-bounds into process/thread priorities.

Moreover, a simulation environment is developed that enables the behaviour of a heterogeous distributed system, consisting of the Timber based embedded software to be observed under a model of the environment.

Furthermore, pedagogic issues of reactive objects have been studied in the context of higher education. First results indicate that the use of TinyTimber give students an increased ability to understand and solve embedded programming assignments.

Finally, the TinyTimber kernel implementation is discussed. Performance metrics are given for a number of representative platforms, showing the applicability of TinyTimber to small embedded systems. A comparison to a traditional system tick driven, thread based, real-time kernel shows that TinyTimber provides tighter timing and a simpler (yet comprehensive) API.

In conclution we find that the use of Reactive Objects in C, realized through TinyTimber is a viable alternative for Embedded Real-Time Programming.

Contents

Chapter 1 – Thesis Introduction 1
1.1 Introduction . . . 1
1.2 Conclusions . . . 2
1.3 Future work . . . 2

Chapter 2 – The Timber kernel 3
2.1 The TinyTimber kernel . . . 3

Chapter 3 – Summary of Contributions 11
3.1 Summary of Contributions . . . 11

Paper A 19

Paper B 31

Paper C 43

Paper D 53

Paper E 57

Paper F 67

Paper G 79

Preface

Most of the work described in this thesis started august 2006, when I got a short project employment at Luleå University of Technology in the CASTT project. I was then given the opportunity to continue my studies as a PhD student in February 2007.

Thanks to Per Lindgren and Jan van Deventer for their excellent guidance and support, and all my fellow work colleagues at EISLAB for creating an inspiring environment at work. I would also like to thank Mikael Nybacka for the great cooperation.

The work in this thesis is founded by Center for Automotive System Technologies and Testing (CASTT).

Part I

Chapter 1

Thesis Introduction

1.1 Introduction

The ever increasing complexity of embedded systems operating under hard real-time constraints, sets new demands on time efficient, rigorous system design and validation methodologies for embedded software. In many cases, such embedded systems are naturally described as time-bound reactions to external events, which may execute in parallel. 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 [1].

This raises the first question of this thesis, **How can the timely behavior of a reactive system be modeled and realized?** The parallelism leads to problems with resource sharing, this leads to yet another question, **How can the state integrity of the system be guaranteed?** And finally, the industry demands for time efficient validation methodologies, **How can the system design be efficiently validated?**

To address these questions we have identified some key challenges in the design of a distributed real-time automotive engine management system (Paper A). The distributed system developed has been used throughout this thesis for experimental setups, and can be considered a reference design where the embedded software is implemented in a traditional thread based manner [2].

To capture the reactive behavior of an embedded system we have undertaken the concept of reactive objects in Timber [3, 4, 5, 6, 7, 8]. Paper B and 2.1 introduces the reader to Timber and TinyTimber.

Furthermore, to demonstrate the suitability of TinyTimber to embedded real-time programming we highlight different aspects of real-time programming and demonstrate possible solutions utilizing reactive objects in papers B, C and D. In paper B we demonstrate how a typical embedded controller in a automotive setting can be modeled, simulated and implemented under Timber semantics. Moreover we develop a simulation environment that enables us to observe the behavior of a heterogeneous distributed system consisting of embedded software, hardware, and a model of the environment. In

paper C and D we illustrate the suitability of reactive objects in the development of real-time communication protocols. We have designed, realized and tested the software in experimental setups, and validated the functionality.

Paper E extends the work done in paper B by a multi-body simulation environment, aiming to provide a time efficient design flow for embedded automotive systems.

We have studied the Timber methodology in the context of higher education, the findings of those studies can be found in paper F.

And to round things up, paper G gives a quick overview of the TinyTimber kernel, and presents performance metrics comparing TinyTimber to a traditional thread based kernel.

1.2 Conclusions

Timber allows the timely behavior of embedded systems to be modeled as time-constrained *reactive objects*, that represent a unification of the object-oriented and parallel programming paradigms. In this thesis we have demonstrated how the Timber semantics for reactive objects translates to embedded real-time programming in C. This is realized through the implementation of TinyTimber, a C API to a minimalistic Timber Run-Time system. The TinyTimber kernel has been presented in section 2.1, performance metrics have been presented for a number of representative embedded platforms (Paper G), the use of Timber/TinyTimber in a simulated environment has been discussed/presented in papers B and E. Papers B, C and D shows that the Timber methodology can be used in embedded real-time systems. In paper F we show that the use of TinyTimber in higher education gives students an increased ability to understand and solve embedded programming assignments. In conclusion, we claim that TinyTimber offers a viable alternative for embedded real-time programming in C.

1.3 Future work

- To further improve the performance of the kernel, we are looking into reducing the critical sections where the kernel has to be protected by disabling interrupts; this will improve the response time and timing performance of the system implementation.
- Using the SRP protocol [9] instead of our current approach (online priority inheritance) will reduce ram memory usage and run-time overhead since we can eliminate the need for context switches and hence run all messages using a single stack.
- Methods for scheduling analysis using WCET [10].
- Documentation, example code, commodity libraries, and design patterns.
- Formal proof of correctness for the kernel.
- Further development of the simulation environment, for instance to support more realistic timing behavior using WCET estimations.

Chapter 2

The Timber kernel

2.1 The TinyTimber kernel

The primary job of the TinyTimber kernel is to manage the messages queues and schedule messages onto execution threads such that (if possible) all message deadlines are met. The TinyTimber 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, or to perform background tasks such as garbage collection [11].

The TinyTimber kernel presented is based on a prototype implementation designed at Luleå University of Technology by Dr. Johan Norlander [12].

2.1.1 Data structures and kernel startup

The TinyTimber kernel comprises a predefined set of statically allocated messages ($M_1 ... M_{NMSGS}$) and execution threads (`thread0`, $T_1 ... T_{NTHREADS}$), `thread0` being the idle thread. Their corresponding datastructures are defined in `kernel.h` (Listing 2.1) and their instances defined in `kernel.c` (Listing 2.2) .

The `RESET(stmt)` (Listing 2.6) macro renders the C `int main(void) {init(); stmt idle();}` hooked to the reset vector executed at system startup. `init()`, (Listing 2.2),

Listing 2.1: kernel.h - messages and threads

```
struct msg_block {
    Msg next;               // for use in linked lists
    Time baseline;          // event time reference point
    Time deadline;          // absolute deadline (=priority)
    OBJECT *to;             // receiving object
    METHOD method;          // code to run
    int arg;                // argument to the above
};

struct thread_block {
    Thread next;            // for use in linked lists
    Msg msg;                // message under execution
    OBJECT *waitsFor;       // deadlock detection link
    CONTEXT context;        // machine state
};
```

Listing 2.2: kernel.c - data structures and initialization

```
struct msg_block     messages[NMSGS];
struct thread_block threads[NTHREADS], thread0;

/* kernel state variables */
Msg msgPool          = messages;
Msg msgQ             = NULL;
Msg timerQ           = NULL;

Thread threadPool    = threads;
Thread activeStack   = &thread0;
Thread current       = &thread0;

void init(void) {
    int i;
    for (i = 0; i < NMSGS - 1; i++)
        messages[i].next = &messages[i + 1];
    messages[NMSGS - 1].next = NULL;

    for (i = 0; i < NTHREADS; i++) {
        threads[i].next = &threads[i + 1];
        CONTEXTINIT_MACRO(&threads[i]); /* Architecture dependent */
    }
    threads[NTHREADS - 1].next = NULL;
    thread0.next = NULL;
    thread0.waitsFor = NULL;
    thread0.msg = NULL;
    INIT();
}

void idle(void) {
    STARTTIMER();
    SETENABLE(TRUE);
    ENABLE();
    while (1) {
        SLEEP();
    }
}
```

saves a unique context to each thread. The global variable `current = &thread0`, used to indicate the currently running thread. Before the `thread0` enters `idle()`, platform specific initiation code is executed, `INIT();`.

Architecture dependencies of the TinyTimber kernel is kept to a minimum, i.e., timer implementation, code for context switch, and idle operation.

The complete source code of TinyTimber can be obtained from the authors under an open-source license.

Kernel data structures

The kernel holds a set of preallocated messages (with base- and deadlines) and threads (execution contexts and pointers to executing message). Messages are circulated between

- the `msgQ`, holding released messages, ordered by increasing deadline, and
- the `timerQ`, holding messages not yet released, ordered by increasing baseline;
- the `msgPool`, holding unallocated messages,

while threads are circulated between

- the `threadPool`, holding unallocated threads, and
- the `activeStack`, holding a stack of pre-empted threads topped by the highest priority thread.

Listing 2.3: kernel.c - queue primitives

```
INLINE_2 BOOL enqueueByDeadline(Msg p, Msg *queue) {
    Msg prev = NULL, q = *queue;
    while (q && (q->deadline - p->deadline <= 0)) {
        prev = q;
        q = q->next;
    }
    p->next = q;
    if (prev == NULL) {
        *queue = p;
        return TRUE;
    } else {
        prev->next = p;
        return FALSE;
    }
}

INLINE_2 BOOL enqueueByBaseline(Msg p, Msg *queue) {
    Msg prev = NULL, q = *queue;
    while (q && (q->baseline - p->baseline <= 0)) {
        prev = q;
        q = q->next;
    }
    p->next = q;
    if (prev == NULL) {
        *queue = p;
        return TRUE;
    } else {
        prev->next = p;
        return FALSE;
    }
}

INLINE_2 Msg dequeue(Msg *queue) {
    Msg m = *queue;
    if (m)
        *queue = m->next;
    else
        PANIC("dequeue:empty queue");
    return m;
}

INLINE_2 void enqueue(Msg m, Msg *queue) {
    m->next = *queue;
    *queue = m;
}

INLINE_2 void push(Thread t, Thread *stack) {
    t->next = *stack;
    *stack = t;
}

INLINE_2 Thread pop(Thread *stack) {
    Thread t = *stack;
    *stack = t->next;
    return t;
}
```

2.1.2 Informal description

The process layer of Timber gives[4];

- scheduling should be deadline based (in accordance to the Timber semantics), and
- object states must be protected at all times.

In the following we will discuss how the TinyTimber kernel addresses the above criteria, with respect to safety, liveness, and real-time properties [1].

These criteria are met by the TinyTimber 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.

Listing 2.4: kernel.c - kernel primitives

```
void dispatch(Thread next) {
    DISPATCH(next);
}

void run(void) {
    while (1) {
        Msg this = current->msg = dequeue(&msgQ);
        request(this->to, this->method, this->arg);
        enqueue(this, &msgPool);
        if ((msgQ == NULL) || (
                (activeStack->next != &thread0) &&
                (msgQ->deadline - activeStack->next->msg->deadline > 0)
                )
            ) {
            Thread t;
            push(pop(&activeStack), &threadPool);
            t = activeStack;  // can't be NULL, may be &thread0
            while (t->waitsFor)
                t = t->waitsFor->ownedBy;
            dispatch(t);
        }
    }
}

void schedule() {
    if (msgQ) {
        if (((activeStack == &thread0) ||
                (msgQ->deadline - activeStack->msg->deadline < 0))) {
            if (threadPool) {
                push(pop(&threadPool), &activeStack);
                dispatch(activeStack);
            } else
                PANIC("schedule:out_of_threads");
        }
    }
}

void timerInterrupt(void) {
    while (timerQ && (timerQ->baseline - TIMERGET() <= 0))
        enqueueByDeadline(dequeue(&timerQ), &msgQ);
    TIMERSET(timerQ);
    schedule();
}
```

During run-time, the TinyTimber kernel will report such hazards. Applications free of such circular references will be executed in a *safe* (with respect to deadlock) manner by the TinyTimber 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.

The actual implementation of the TinyTimber kernel is given in Listings 2.3, 2.4, 2.5, and its call graph is sketched in Figure 2.1. In the following, we will consider the C code implementation as a formal specification of the TinyTimber kernel.

Kernel invariants

State integrity must be preserved for the kernel, the implicit root object, and all TinyTimber object instances. Executing kernel primitives with all interrupts disabled ensures state integrity of the kernel data structures

Conceptually, the external interrupts are bound to the application root object and

Listing 2.5: kernel.c - message primitives

```
void action(Time bl, Time dl, OBJECT *to, METHOD meth, int arg) {
    int wasEnabled = ISENABLED();
    DISABLE();
    Msg m;
    m = dequeue(&msgPool);
    m->to = to;
    m->method = meth;
    m->arg = arg;
    if (wasEnabled) { // action from current message
        m->baseline = current->msg->baseline + bl;
        if (m->baseline < TIMERGET())
            m->baseline = TIMERGET();
        m->deadline = m->baseline + dl;
    } else { // action from interrupt or reset
        m->baseline = TIMESTAMP() + bl;
        m->deadline = TIMESTAMP() + bl + dl;
    }

    if (m->baseline - TIMERGET() <= 0)
        enqueueByDeadline(m, &msgQ);
    else
        if (enqueueByBaseline(m, &timerQ))
            TIMERSET(timerQ);
    ENABLE();
}



int request(OBJECT *to, METHOD meth, int arg) {
    DISABLE();
    Thread t;
    int result;
    t = to->ownedBy;
    if (t) {   // to is already locked
        while (t->waitsFor)
            t = t->waitsFor->ownedBy;
        if (t == current)
            return -1; // Deadlock
        Thread oldTo = to->wantedBy;
        to->wantedBy = current;
        current->waitsFor = to;
        dispatch(t);
        if (oldTo)
            oldTo = NULL;
    }
    to->ownedBy = current;

    ENABLE();
    result = meth(to, arg);
    DISABLE();

    to->ownedBy = NULL;
    t = to->wantedBy;
    if (t) {  // we have run on someone's behalf
        to->wantedBy = NULL;
        t->waitsFor = NULL;
        dispatch(t);
    }
    ENABLE();
    return result;
}
```

are executed with a relative deadline zero. The interrupt handlers are scheduled by the hardware (IRQ controller). State integrity of the root object is ensured by executing interrupt handlers under mutual exclusion, i.e. with interupts disabled.

Finally, the ordinary methods execute on TinyTimber object instances under mutual exclusion. This is upheld by the kernel primitives, assuming that the application never makes any direct function calls or data accesses across object boundaries.

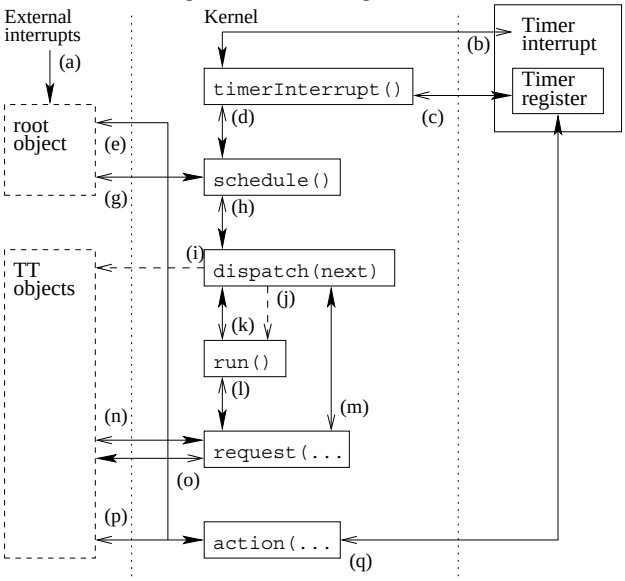
Interrupts

External interrupts are bound to (and managed by) methods on the application root object Figure 2.1(a). These methods may issue asynchronous (e) messages to TinyTimber

Listing 2.6: tt.h - TinyTimber API

```
typedef struct {
    Thread ownedBy;
    Thread wantedBy;
} OBJECT;

// initializes an OBJECT
#define initOBJECT()   {NULL, NULL}

typedef int (*METHOD)(OBJECT*, int);

#define ASYNC(bl, dl, to, meth, arg) \
    action(bl, dl, (OBJECT*)to, (METHOD)meth, (int)arg)

#define SYNC(to, meth, arg) \
    request((OBJECT*)to, (METHOD)meth, (int)arg)

#define RESET(stmt) \
    int main(void) {init(); stmt; idle(); return 0;}

#define BASELINE() \
    current->baseline

#define INTERRUPT(vector, stmt) \
    /* Platform dependent */

#define MSEC(x) \
    /* Platform dependent */

#define SEC(x) \
    /* Platform dependent */
```

Figure 2.1: Kernel primitives.



object instances.

The timer interrupt invokes `timerInterrupt()` (b) when (at least) one message is due for scheduling.

In case of a timer interrupt (b), messages due for release are moved from the `timerQ` to the `msgQ` and the hardware timer is set according to the next absolute baseline (c). In case the application interrupt handler posts an asynchronous message that is due for release, it will be put directly into the `msgQ`.

Scheduling of a new thread is carried out when a released message `msgQ->deadline` has higher priority (i.e., earlier deadline) than currently highest prioritized released message `activeStack->msg->deadline`. The new thread is pushed onto the active stack, and dispatched (h), leading to the context switch (j). Dashed lines indicate context switches.

run(), the life of a thread

The current thread pops the `msgQ` for the scheduled message, and executed by `request(...)`. On return, the message is put back in the `msgPool`. If there are no more released messages of higher priority than the message running on the next level of the thread stack, the current thread will be popped from the `activeStack` and reclaimed. The `activeStack` now holds the next message in order of priority. Assuming that this thread is *not* blocked, we can continue its execution by a dispatch to `activeStack` (k, i). If, on the other hand, the thread is blocked, we dispatch the thread at the end of the transitive blocking chain (k, i). Because scheduling decisions are based on `activeStack->msg->deadline` (which is invariant to the currently executing thread, priority inheritance is accomplished).

Message execution by request

If the object instance is blocked, we search for the end of the transitive blocking chain. This allows on-the-fly deadlock detection. If this message execution was on behalf of a higher priority thread, we resume execution of that thread (m), if not we just return to `run()` (l).

2.1.3 The TinyTimber API

Summary of the TinyTimber interface

- `#include "tt.h"`
  Provides access to the TinyTimber primitives.

- `typedef struct {...} Object;`
  Base class of reactive objects. Every reactive object in a TinyTimber system must be of a class that inherits this class.

- `#define initObject() {...}`
  Initialization macro for class `Object`.

- `typedef ... Time;`
  Abstract type of time values.

- `Time SEC( int seconds );`

- `Time MILLISEC( int milliseconds );`

- `Time MICROSEC( int microseconds );`
  Constructs a `Time` value from an argument given in `seconds / milliseconds / microseconds`.

- `int SYNC( T *obj, int (*m)(T*,A), A a );`
  Synchronously invokes method `m` on object `obj` with argument `a`. Type `T` must be a struct type that inherits from `Object`, while `A` can be any int-sized type. If completion of the call would result in deadlock, -1 is returned; otherwise the result is the value returned by `m`.

- `void ASYNC(Time bl, Time dl, T *obj, int (*m)(T*,A), A a);`
  Asynchronously invokes method `m` on object `obj` with argument `a`, baseline offset `b`, and relative deadline `d`. Type `T` must be a struct type that inherits from `Object`, while `A` can be any int-sized type. Offsets bl and dl allow a new execution window for the asynchronous call to be defined on basis of the current one:

```
new baseline = current baseline,                    bl < 0
             = max(current baseline + bl, current time), otherwise
new deadline = current deadline,                    dl < 0
             = new baseline + dl,                   otherwise .
```

- `Time BASELINE( void );`
  Returns the baseline of the currently executing method call.

- `INTERRUPT( vector, stmt );`
  Declares an interrupt handler for vector whose body is `stmt`.

- `RESET( stmt );`
  Declares a system startup handler whose body is `stmt`.

Chapter 3

Summary of Contributions

3.1 Summary of Contributions

In this chapter, my personal contributions in all the included papers are pointed out.

3.1.1 Paper A

A Distributed Engine Management System for Formula SAE

Authors: Johan Eriksson, Per Lindgren and Jan Van Deventer

This paper is to be considered as background work for this thesis, it gives an overview of a network of ECU's used in a Formula SAE car. Furthermore it decribes the methodology used, and some of the pitfalls with embedded real-time programming. The work descibed in this paper is done by me and most of the writing.

3.1.2 Paper B

A Comprehensive Approach to Design of Embedded Real-time Software for Controlling Mechanical Systems

Authors: Johan Eriksson, Per Lindgren

A clutch and gearshift controller for a Formula SAE car is presented, the focus of this paper is the use of the TinyTimber kernel and it is shown that the use of the TinyTimber kernel in embedded real-time programming is possible and could reduce the development time of the embedded software. All the work and writing is done by me, except for the section on Timber.

3.1.3 Paper C

FIFO WiDOM : Timely Control Over Wireless Links.

Authors: Viktor Leijon, Per Lindgren and Johan Eriksson

We present the idea of scheduling a control network in a first-come first-serve manner, and demonstrate a way to implement this over a wireless link. This achieves real-time guarantees and tightly bounded jitter for control applications. Further, how to design systems using this network is discussed, and the benefits compared to a standard priority MAC protocol are examined through simulations and a prototype implementation. The prototype is implemented using the reactive Timber programming model on a lightweight 8-bit platform.

My contribution in this paper is creating the test platform, and implementing the software using TinyTimber

3.1.4 Paper D

IP Over CAN : Transparent Vehicular to Infrastructure Access.

Authors: Per Lindgren, Simon Aittamaa and Johan Eriksson

In this paper an working IP implementation is presented that runs on to of a can network. The software is realized with the Timber programming model, using the TinyTimber kernel. My primary contribution to this paper is creation of the testplatform, coding of the MAC layer (for the CAN bus).

3.1.5 Paper E

Using Timber in a multi-body design environment to develop reliable embedded software.

Authors: Johan Eriksson, Mikael Nybacka, Tobias Larsson and Per Lindgren

In this paper we extend previous work on Timber with a multi-paradigm design environment, aiming to bridge the gap between engineering disciplines by multi-body co-simulation of vehicle dynamics, embedded electronics, and embedded executable models. Its feasibility is demonstrated on a case study of a typical automotive application (traction control), and its potential advantages are discussed.

I have contributed with the co-simulation setup and the traction control software in TinyTimber.

3.1.6 Paper F

Comprehensive Reactive Real-Time Programming

Authors: Per Lindgren, Johan Nordlander, Kalevi Hyyppä, Simon Aittamaa and Johan Eriksson

Focus on the educational aspects of the use of the Timber methodolody and the TinyTimber kernel. My primary contribution is in chapter III, the TinyTimber and FreeRTOS comparison.

3.1.7 Paper G

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

Authors: Per Lindgren, Johan Eriksson, Simon Aittamaa and Johan Norlander

In this paper, the TinyTimber 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 TinyTimber kernel is also compared to a thread based OS.

My primary contributions is programming and discussion.

References

[1] H. Klapuri, J. Takala, and J. Saarinen, "Safety, liveness and real-time in embedded system design," *Journal of Network and Computer Applications*, vol. 22, no. 2, pp. 69–89, 1999.

[2] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages*. Addison-Wesley, 2001.

[3] 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.

[4] M. Carlsson, J. Nordlander, and D. Kieburtz, "The semantic layers of Timber," in *Programming Languages and Systems, First Asian Symposium, APLAS 2003, Beijing, China* (A. Ohori, ed.), vol. 2895 of *Lecture Notes in Computer Science*, Springer, November 2003.

[5] M. Kero, P. Lindgren, and J. Nordlander, "Timber as an RTOS for Small Embedded Devices," in *Workshop on Real-World Wireless Sensor Networks*, 2005.

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

[7] 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.

[8] P. Lindgren, J. Nordlander, L. Svensson, and J. Eriksson, "Time for Timber," Tech. Rep. ISSN 1402-1528, EISLAB, Luleå University of Technology, Sweden, 2005.

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

[10] B. Lisper, "Fully automatic, parametric worst-case execution time analysis," in *WCET*, pp. 99–102, 2003.

[11] M. Kero, J. Nordlander, and P. Lindgren, "A Correct and Useful Incremental Copying Garbage Collector," in *Published in ISMM: International Symposium on Memory Management*, 2007.

[12] J. Norlander, "Programming with the TinyTimber kernel," Tech. Rep. ISSN 1402-1536, EISLAB, Luleå University of Technology, Sweden, Nov. 2007.

Part II

Paper A

A Distributed Engine Management System for Formula SAE

Authors:
Johan Eriksson, Per Lindgren and Jan Van Deventer

Reformatted version of paper originally published in:
SAE World Congress, 2007

SAE TECHNICAL PAPER SERIES

2007-01-1602

A Distributed Engine Management System for Formula SAE

Johan Eriksson, Per Lindgren and Jan van Deventer
Luleå University of Technology, Sweden

Reprinted From: Electronic Engine Controls, 2007 (SP-2087)

ISBN 0-7680-1636-3

9 780768 016369

**2007 World Congress
Detroit, Michigan
April 16-19, 2007**

SAE International™

400 Commonwealth Drive, Warrendale, PA 15096-0001 U.S.A. Tel: (724) 776-4841 Fax: (724) 776-0790 Web: www.sae.org

ISSN 0148-7191

2007-01-1602

A Distributed Engine Management System for Formula SAE

Johan Eriksson, Per Lindgren and Jan van Deventer
Luleå University of Technology, Sweden

ABSTRACT

In this paper a distributed system for engine management is presented. The system is in use on the 2006 and 2005 Formula SAE cars from Luleå University of Technology.

The purpose of building such a system from scratch is to have a comprehensive, predictable and easily extendable platform, giving the possibility to add extra features even at the racetrack. This allows the system to serve as a research platform for embedded real-time systems and vehicle dynamics. Another motivation is to get low weight on the complete system, and to integrate the electronics in such a way that the total cabling required will be minimal.

The initial requirements are that the system should implement launch control, traction control, electric gear shift and clutch control. To control the engine the system must implement sequential fuel injection, direct fire ignition and closed loop lambda control. Moreover to remotely tune and monitor the system parameters in real-time - even on the racetrack, the system should facilitate wireless communication.

To achieve these goals a system consisting of five units communicating over a standard automotive bus (CAN[1]) was developed.

In this paper we will describe the systems functionality and the units developed.

INTRODUCTION

After successfully participating in the Formula Student competition for a number of years, the goals and ambition for the car made at Luleå University of Technology have been raised. This lead to new demands on the embedded electronics in the car, and the conclusion that there is simply not one „off the shelf" system that would have all the features that we felt were needed, especially not with low weight, and at acceptable cost. The cost is of extra importance, since one part of the competition is about the cost of producing 1000 cars - and with a in house developed engine management system the fictional cost for producing 1000 units will be very low.

Figure 1: The car at 2006 Formula Student competition.

The initial requirements were:

- Advanced engine control, including:
  - Sequential injection.
  - Direct fire ignition.
  - Closed loop wideband lambda control.
- Electronic control of gear shifting and clutch.
- Driver aids, including:
  - Traction control.
  - Launch control.
  - Automatic shifting.

[1] Controller Area Network

- A Driver interface on the steering wheel.

All these features should be integrated into a single system, open for future extensions.

PARTITIONING THE PROBLEM

Looking at the system requirements as a specification we seek a partitioning into different units with respect to; interdependencies (in order to reduce the total amount of communication between units) and interfaces (in order to reduce wiring between units and sensors/actuators). Another aspect is EMI[2], such that highly emitting components could be isolated from the sensor interfaces.

From the above criteria we found a suitable partitioning. The following units were developed:

- Processing unit, does all calculations, handles remote communication and will implement most of the high level functionality.
- Sensor unit, connects to the engine sensors, and has some auxiliary inputs and output.
- Output unit, connects to the sensors used for sensing the position of the crankshaft and drives the injectors and ignition coils.
- Gearshift unit, measures the position of the gearshift and clutch position, and controls the gearshift solenoid and clutch engine.
- Steering wheel unit, acts as the drivers interface to the onboard electronics.

Figure 2 shows how all the units are connected.

The units communicate over two CAN busses; one bus is used for critical systems and the other system is used for less critical data. A critical system is defined as a system required for the car to run and pass the finish line.

PROCESSING UNIT

The processing unit connects the two CAN busses together and has an Ethernet connector used for the wireless link. This unit also features a SD[3] memory connector that is used for data logging. Figures 3 and 4 show the placement of the unit.

For the calculations to be done in parallel with the other tasks this unit is running a multi threaded RTOS[4]. FreeRTOS[4] was chosen because of its open source nature, its small code size and low memory usage.

[2]Electro Magnetic Interference
[3]Secure digital
[4]Real-Time Operation System

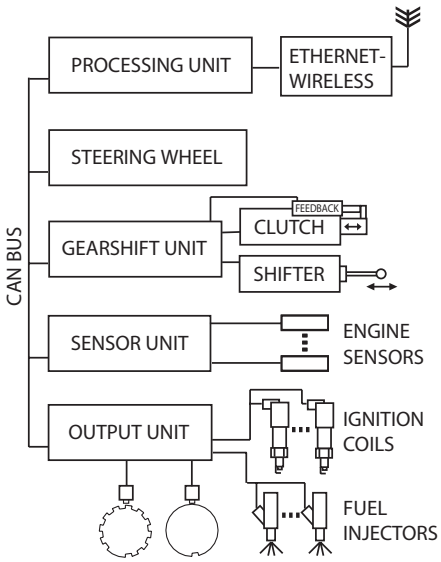


Figure 2: Modular view of the complete system.

Since the processing unit will be responsible for most of the calculations, handle events on the CAN bus, process Ethernet data and log data on the memory card, a powerful 32bit ARM micro controller from PHILIPS was chosen(LPC2119[8]). It has the memory needed and features dual embedded CAN controllers. The micro controller is clocked at 50Mhz.

All the other units run a 8bit ATMEL AVR AT90CAN128[1] clocked at 16Mhz, and the software is implemented without any operating system or kernel.

The code for all units are written in C using the freely available GCC[5] compiler.

The software for the processing unit is written using one thread for each function, the threads are:

- Calculation thread.
- CAN thread.
- Ethernet thread.

These threads communicate by using message queues. In this section we will give a short description of the tasks that the different threads do and refer to figure 5 for a block view of the embedded software.

CALCULATION THREAD The primary purpose of the processing unit is to control the other units on the car. This task is implemented by the calculation thread. The calculation thread also synchronizes the clocks on the other

Figure 3: The rear mounted electronics. Showing from left: Fuse box and charging circuit, Gearchange unit, and Processing unit.

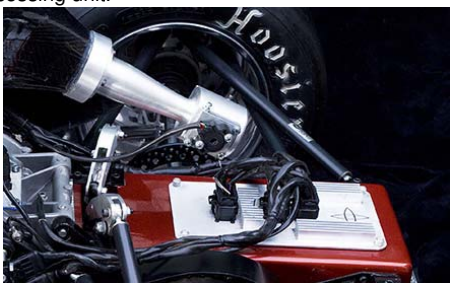


Figure 4: The mounting of the electronics in figure 3.

units by sending out a clock sync message on both CAN busses every 10ms. The other units are then required to reply with the periodic data within 3ms. The processing unit will then perform calculations based on the received data and send out the newly calculated parameters. The operation of the calculation thread can be described by the state machine below:

1. Block, waiting for time to be a multiple of 10ms.
2. Send out clock synchronization message, done by posting a „CAN SEND" message on the CAN queue.
3. Block thread for 3ms.
4. Check that the CAN thread has received the data needed, otherwise assume some „safe" values.
5. Calculate desired output & update internal variables.
6. Send the new parameters by posting messages on the CAN transmit queue.
7. Post a message on the Ethernet queue to broadcast the new data.
8. Go back to (1).

The periodic data used in the calculations are listed below:

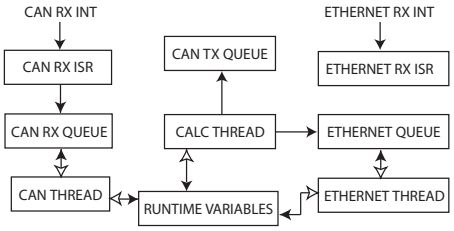


Figure 5: View of the threads and queues used in the processing unit.

- Data from sensor unit:
  - Intake pressure.
  - Intake temperature.
  - Throttle position.
  - Lambda value.
  - Coolant temperature.
- Data from output unit:
  - Current RPM[5].
- Data from gear change unit:
  - Current gear.
  - Ignition cut.
- Internally measured data:
  - Front wheels speeds.
  - Back wheels speeds.
- Data from steering wheel unit:
  - Buttons.

To save memory, all data that is received is placed in a global data structure by the CAN-thread. This means that care must be taken to make sure that no race conditions[6] may occur.

ETHERNET THREAD This thread blocks, waiting for messages on the Ethernet queue, and takes different actions depending on the message received. If a broadcast message is received from the calculation thread, a UDP[7] broadcast containing all the runtime data will be sent out. If a new data interrupt message is received containing new parameter settings, the thread will verify the data and then update settings accordingly.

CAN THREAD This thread handles reception of CAN frames. The thread blocks while waiting for CAN RX interrupts, and for each interrupt it will process the data and update the internal variables.

[5] Revolutions per minute

[6] For example when the same variable is written from two routines simultaneously

[7] User Datagram Protocol

SENSOR UNIT

This unit connects to all engine sensors:

- 4 NTC thermistor inputs for measuring temperatures including:
  - Coolant temperature.
  - Intake temperature.
- 6 K-type thermocouple sensors for measuring exhaust gas temperature.
- Direct interface to a Bosch Planar Wide Band Lambda sensor (Bosch LSU 4.x)[9], figure 6 shows a schematic of how the sensor is connected to the mcu.
- 3 onboard absolute pressure sensors that measures the:
  - Intake pressure.
  - Exhaust pressure (turbo car).
  - Choke pressure after the restrictor (turbo car).
- 8 Analog 0-5V analog inputs for measuring throttle position, oil pressure and auxiliary inputs.

The unit also has these outputs:

- One high current (44A) PWM[8] output used for fan and electric water pump control.
- Three medium current (3A) PWM outputs used for controlling:
  - Fuel pump relay.
  - Boost control valve.
- One logic level PWM output used for controlling a servo. It is primarily used for controlling a turbocharger with variable nozzles, such as a garret GT15V that is commonly used on Formula SAE cars.
- One analog output with 12bits of precision.

This unit is mounted in the headrest, as seen in figure 7.

OUTPUT UNIT

This unit connects to the injectors and the ignition coils. To keep the wiring short to the engine this unit is mounted in the headrest, together with the sensor unit, as shown on figure 7. The inputs to this unit are the sensor on the crank shaft and the sensor of the camshaft. The crank shaft sensor is used to sense the absolute position of the crank shaft. The toothed wheel is of missing tooth type.

[8]Pulse width modulation

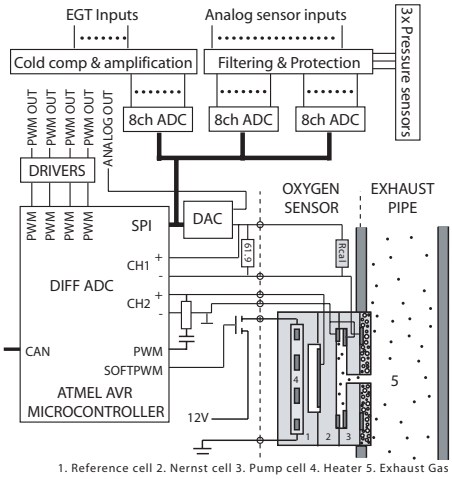


Figure 6: Schematic of the sensor unit.

Figure 7: Picture showing the mounting of the sensor and output units in the headrest.

The cam shaft sensor is used to sense which stroke the engine is in. This is required for sequential injection, and to allow for separate ignition coils for each sparkplug.

The code runs without a kernel and all communication between the different interrupt handlers are done by sharing a set of global variables. This means that extreme care must be taken to make sure that race conditions do not occur. This is done by the masking of interrupts when critical variables are accessed. Identifying when to mask interrupts are done through offline analysis.

STEERING WHEEL

The steering wheel communicates with the driver using a graphical PLED[9] display and some status and warning diodes. Driver input is implemented by six buttons, two

[9]Polymer Light-Emitting Diode

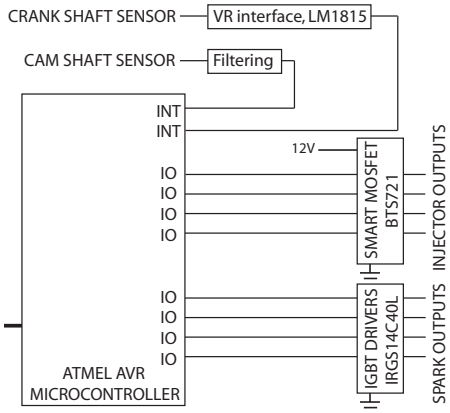


Figure 8: Schematic of the output unit.

gearshift paddles and a clutch lever. Figure 9 shows the steering wheel with the integrated electronics. The driver has the option to choose between different pages on the display and to choose what data to be displayed.

If some critical condition should occur the steering wheel will light up a warning diode to indicate to the driver that something is wrong. The warning diodes on the steering wheel are:

- Oil pressure low.
- Water temperature High.

There is also some status diodes:

- Traction control status - lights up if activated, flashes if actively controlling traction.
- Launch control status - lights up if activated.
- Automatic shift status - lights up if activated.

GEARSHIFT AND CLUTCH CONTROL UNIT

This unit controls the dual acting solenoid that actuates the sequential gearbox, and the clutch actuator. The inputs to this unit are:

- Gear drum position.
- Clutch actuator position.

GEAR SHIFTING The gear drum position sensor is a 360° potentiometer that measures the absolute position of the gear drum; this gives us the means to determine the current gear and to tell when the gear has engaged.

When shifting the operation is as follows:

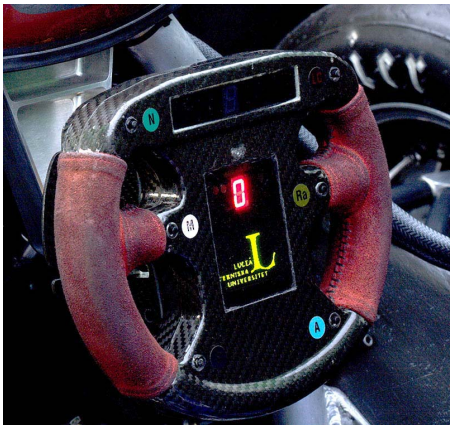


Figure 9: Picture of the steering wheel.

- Send an „ignition cut on" message.
- If shifting down and throttle is released, actuate the clutch.
- Actuate the shifting solenoid in the desired direction.
- Monitor the Gear drum position signal, waiting for the gear to engage. The maximum waiting time is 300ms, if this time elapses there has been some error.
- Deactuate the solenoid.
- Send a „ignition cut off" message.
- If the clutch was actuated then deactuate it.

CLUTCH CONTROL To control the clutch a linear actuator was constructed. The actuator is powered by a DC engine, with a planetary gear box, operating a ball screw[10]. To power the DC engine a H-bridge is used. Feedback is used to measure the position of the ball screw, this is done using a linear potentiometer. A PID[10] controller is embedded into the unit that controls the engine. The target for the controller is generated either by the steering wheel unit, or when shifting by the state machine described above.

DIFFERENT APPLICATIONS

On the two different Formula SAE cars, developed at Luleå University of Technology, that uses this system there is some small differences in the configuration. This is mainly due to the fact that the 2006 car is naturally aspirated and that the 2005 car is turbo charged. There are also some differences in the usage of the cars. The 2006 car is used mainly for driver training and racing, while the 2005 car is used for performing research and testing new ideas.

[10] Proportional Integral Derivative

PC CONFIGURATION AND LOGGING TOOL

To tune and monitor the car a logging and mapping software was developed using LABVIEW. This is a graphical environment which allows a very fast way of producing graphical interfaces. Figure 10 shows a screenshot of the Labview program used for tuning.

The software allows real-time logging and tuning of all the engine parameters, for example all engine tables, regulator parameters and constants. The software also features logging capabilities, that can be easily adjusted to allow for logging only the parameters that are needed. It is also possible to log any data that is available on the CAN bus. One may also send any data from the PC to the CAN network. These features allow real-time monitoring and adjustment of the equipment that is connected to the CAN bus.

TESTING

The system has been extensively and systematically tested to verify correct operation and good performance. Since the system is modular, each unit can first be tested and verified for correct operation before the systems are connected together.

INITIAL TESTING IN A LAB ENVIRONMENT Each of the units were tested and verified in a laboratory setting using a separate micro controller to simulate the primary engine outputs: the crank sensor with the missing teeths and the cam sensor.

FIRST ENGINE TESTING To test the engine management software the system was hooked up to a turbo charged Volvo engine that was mounted on an engine dynamometer. During the first phase of the testing, only the sensor unit and the output unit were tested, and the calculations were done in real-time using National Instruments LABVIEW. A CAN to PC interface was used to handle the communication between the PC and the micro controllers. This approach not only reduced the number of errors that could occur, but also gave a very fast way of developing the control algorithms, by first testing them in LABVIEW on a real engine before implementing them in C on the embedded system.

When the algorithms had been tested and verified the LABVIEW code was implemented in C on the ARM micro controller. And the PC was taken out of the control loop. Now the PC was used only for adjusting the control parameters, engine tables and to monitor the data.

DYNAMOMETER TESTING OF THE 600CC ENGINE IN THE FORMULA SAE CAR Testing and development of the engine was initially performed with a DTA S60 PRO ECU[2]. During this phase of development the primary objective was to optimize the shape, size and length of the intake manifold, intake runners, restrictor, exhaust headers and muffler - to get a good power output and a good drivable characteristic of the engine. After completing this phase, the engine management system was replaced and the system was fine tuned. This also gave us the opportunity to validate the system by comparing it with the DTA unit.

REAL CAR TESTING When the car was ready to drive the system could be verified for drivability and the controllers for the traction control, launch control and automatic shifting could be validated. The system was found to be reliable and to have a very good performance. The car and the complete electrical system also proved itself to be reliable during the 2006 Formula Student competition, with good scores on all events.

FORMULA STUDENT 2006 RESULTS

At the 2006 Formula Student our team finished on an overall 17th place, with the following placement in the individual events:

- Presentation first place, 75 points.
- Fuel Economy 7th place, 31points.
- Cost 11th place, 92 points.
- Acceleration 14th place, 51 points.
- Skidpad 18th place, 32 points.
- Endurance 20th place, 134points.
- Sprint 25th place, 54 points.
- Design 37th place, 63 points.

CONCLUSIONS AND FUTURE WORK

In this paper a distributed system for engine control has been presented.

The benefits are numerous when compared to standard „off the shelf" units, for example: reduced budgeted cost, low weight and reduced wiring.

The system successfully implements the following functionality; launch control, traction control, electric gear shift and clutch control, and engine control consisting of sequential fuel injection, direct fire ignition and closed loop lambda control. Moreover, using wireless technology, system parameters can be remotely monitored and tuned in real-time - even on the racetrack.

In this paper we have described the overall system architecture and discussed the processing-, sensor-, output-, gearshift-, and steering wheel units in detail.

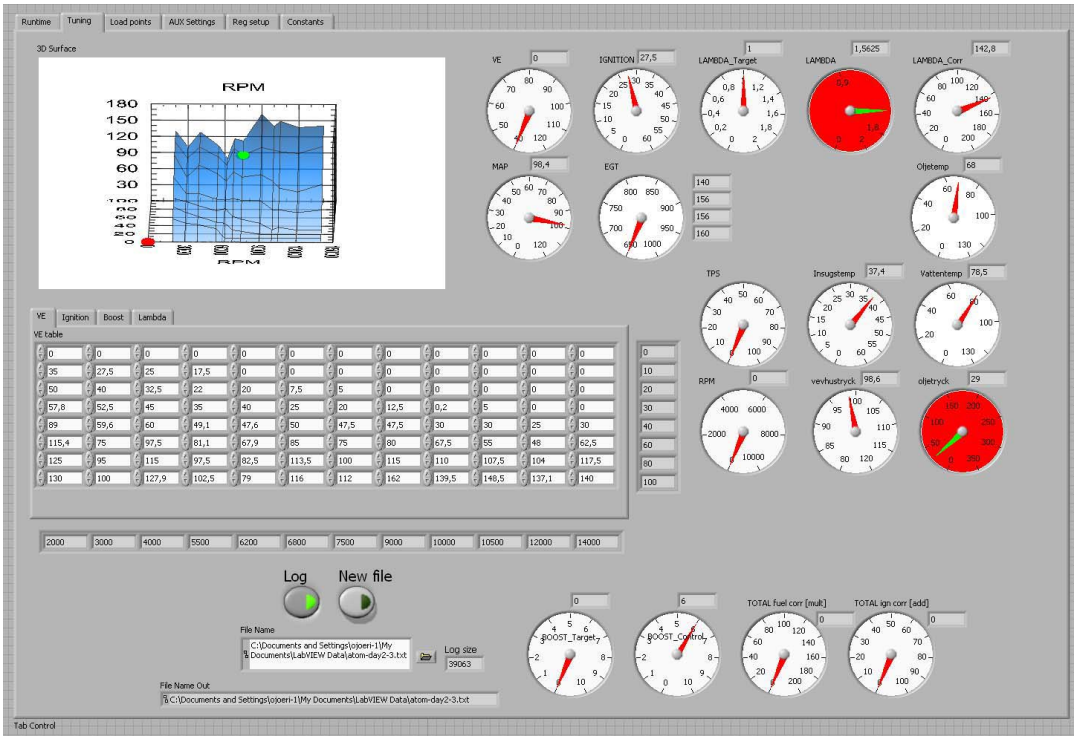


Figure 10: The GUI developed in LABVIEW. Shown is the view used during mapping of the engine.

The chosen partitioning of the system into units has shown to be successful, mainly because the partitioned system could be straightforwardly tested and verified. The system has been deployed onto the 2005 (turbo charged) and 2006 (naturally aspirated) Formula SAE cars developed at Luleå University of Technology, with minimal effort for configuration needed to cope with the different setups.

Our drivers have felt more comfortable driving the car with the help of the driver assisting systems, especially the traction control.

This system currently serves as a research platform for embedded real-time systems and vehicle dynamics. The proposed comprehensive, modular, and distributed system architecture provides the necessary means to straightforwardly implement both additional hardware and software functionality.

ACKNOWLEDGMENT

This work has been funded by CASTT - Center for Automotive System Technologies and Testing at Luleå.

REFERENCES

[1] Atmel. AT90CAN128. http://www.atmel.com/dyn/products /product_card.asp?PN=AT90CAN128.

[2] DTA Fast. S60 PRO. . http://www.dtafast.co.uk/S_60_PRO.htm.

[3] J. Eriksson. An Engine Management System for the LTU Formula SAE race car. *Master thesis at Luleå Univeristy of technology, Sweden*, 2006.

[4] FreeRTOS. -A Free RTOS for ARM7,ARM9,Cortex-M3,... http://www.freertos.org/.

[5] GNU Project. GNU Compiler Collection. http://gcc.gnu.org/.

[6] D. Malmgren. Automotive Electronics and their implementation in a race car. *Bachelor thesis at Luleå Univeristy of technology, Sweden*, 2006.

[7] Mario Farrugia, Michael Farrugia and Brian Sangeorzan. ECU Development for a Formula SAE Engine. *SAE World Congress. Detroit, Michigan*, (2005-01-0027), April 2005.

[8] Philips. LPC2119; Single-chip 16/32-bit microcontroller; 128 kB ISP/IAP flash with 10-bit ADC and CAN. http://www.nxp.com/pip/LPC2119FBD64.html.

[9] Robert Bosch GmbH. *Planar Wide Band Lambda Sensor, Y 258 K01 005-000e*, March 2001.

[10] SKF. Linear Motion. http://www.linearmotion.skf.com/.

CONTACT

This paper is mainly written by Johan Eriksson (johan.eriksson@ltu.se), the work described here was done as a master thesis. Johan Eriksson is currently a PHD student at Luleå University of Technology, Sweden.

Paper B

A Comprehensive Approach to Design of Embedded Real-time Software for Controlling Mechanical Systems

Authors:
Johan Eriksson and Per Lindgren

Reformatted version of paper originally published in:
14th Asia Pacific Automotive Engineering Conference Hollywood, California, USA August 5-8, 2007

SAE TECHNICAL PAPER SERIES

2007-01-3744

A Comprehensive Approach to Design of Embedded Real-Time Software for Controlling Mechanical Systems

Johan Eriksson and Per Lindgren
Luleå University of Technology

14th Asia Pacific Automotive Engineering Conference
Hollywood, California, USA
August 5-8, 2007

SAE International™

400 Commonwealth Drive, Warrendale, PA 15096-0001 U.S.A. Tel: (724) 776-4841 Fax: (724) 776-0790 Web: www.sae.org

ISSN 0148-7191

Printed in USA

2007-01-3744

A Comprehensive Approach to Design of Embedded Real-Time Software for Controlling Mechanical Systems

Johan Eriksson and Per Lindgren
Luleå University of Technology

ABSTRACT

In this paper, we present a comprehensive approach to design of embedded real-time software for electrically controlled mechanical systems in automotive applications. As a case study, we implement a Gear change and Clutch controller for a Formula SAE car. This includes a generic communication interface and protocol for CAN bus communication, I/O interfaces for A/D conversion and PWM output, together with a PID controller for clutch actuation.

Under our framework, the embedded software is developed using Timber, a programming language and formalism that provides executable models for embedded real-time systems.

The case study shows how a complete control system can be straightforwardly modeled, simulated and transformed into executable code. The system has been realized and tested onto a lightweight, 8-bit AVR-5, embedded platform.

Compared to the raw C code design flow, the proposed framework has in our case study showed increased efficiency with respect to development time. We boldly conclude that our Timber based framework offers true **"work with the work".**

INTRODUCTION

Implementing real-time embedded systems is known to be a hard and time-consuming task, where guru-like knowledge of extensive RTOS libraries, and expert programming skills [8] seems absolutely mandatory. However, from an external point of view, the desirable behavior of a real-time system is actually quite simple: for each handled event, a specified response should be produced within a specified deadline (e.g., reacting to incoming CAN bus data). This view is manifested by the high-level programming and systems modelling language Timber.

Timber adopts the reactive paradigms proven successful to hardware description languages (such as Verilog and VHDL), and offers abstractions of modern functional programming languages, e.g., higher order functions, type inference and objects.

The timely behavior for each software component is given directly in the Timber specification, much like what you expect to find in the data-sheets of some physical component (its transfer function/in-to-out delay/response time, or similar).

Timber objects execute in parallel, (much like you would expect hardware components to live their own life). However, operations on a specific object execute exclusively (forcing other operations on the same object to wait). In this way the state of the object is protected at all times. Traditionally, this is a major obstacle in reactive (real-time) programming, as manually accounting for state protection utilizing RTOS primitives such as locks/semaphores/monitors etc.

In the paper we demonstrate a multi-paradigm design environment, aiming to bridge the gap between engineering disciplines by co-simulation of mechanical system dynamics, embedded electronics, and embedded Timber software. We undertake the programming model of Timber, through a C interface to a minimalistic, robust, and portable Timber run-time kernel. This allows for comprehensive and efficient implementation with predictable memory and timing behaviour.

The case study shows how a complete control system can be straightforwardly modeled and transformed into executable code. The system has been realized onto a lightweight, 8-bit AVR-5, embedded platform.

TIMBER

Embedded real-time systems are naturally described as time-bound reactions to external events, a view supported natively in the high-level programming and systems modelling language Timber [9]. The engineering

perspectives of the Timber design paradigm are further elaborated in [10].

In this paper, we undertake the programming model of Timber, through a C interface to a minimalistic, robust, and portable Timber run-time kernel. This allows for comprehensive and efficient implementation with predictable memory and timing behaviour. The C application interface features the following subset of Timber:

- Concurrent reactive objects
- State protection
- Synchronous and asynchronous messages
- Deadline scheduling

The Timber run-time model utilizes deadline scheduling directly on basis of programmer declared event information. In short:

• Objects and concurrency: The concurrent 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 protection 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 timing constraints on the reaction to an event can be directly reused as run-time parameters for scheduling the corresponding method. The permissible window for method execution is defined by the baseline (the absolute point in time for event release), together with the relative deadline. Chain reactions are expressed through relating future events to the locally fixed baseline, and will be free from jitter induced by the actual scheduling.

On the top level, a Timber-C application consists of the implicit root object and its methods, which are the functions installed for handling interrupts and the reset signal (main in C). 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.

A Timber-C application is compiled and linked with the Timber 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 any additional libraries, although libraries may be incorporated as long as they do not violate the run-to-end assumption.

In the design of the Timber kernel, utmost care has been taken in order to offer bounded memory and timing behaviour that is controllable by compile time parameters. The Timber 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.

In the context of other minimalistic operating systems, the Timber kernel stands out with its deadline-driven scheduling and the concept of concurrently executing reactive objects. The Timber kernel is truly lightweight, matching the resource requirements and performance metrics of popular kernels such as TinyOS [13], Contiki [13], FreeRTOS [12] and AmbientRT[15].

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, the Timber-C kernel differs fundamentally to AmbientRT in terms of our object-based locking mechanism, which allows true multithreading and hence improves on schedulability.

CASE STUDY

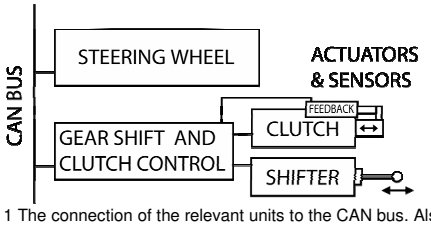


Figure 1 The connection of the relevant units to the CAN bus. Also illustrated is the shifter and clutch actuator's connection to the gear shift and clutch controller.

FUNCTIONALITY OVERVIEW

The functionality required of the embedded software described later in this paper are mainly:

- Means to operate a dual acting solenoid that moves the gear shifting arm on the sequential gearbox.
- Send ignition cut messages to the Engine Management System when shifting - see [5] and [6]
- Control of the linear actuator that operates the clutch.

As seen in Figure 1 the gear change and clutch controller is connected with the rest of the systems on the car through a CAN bus.

To control the gear shift and clutch controller the steering wheel sends out a CAN message containing:

- The target for the clutch actuators closed loop controller.
- A optional command that can be either “shift up” or “shift down”

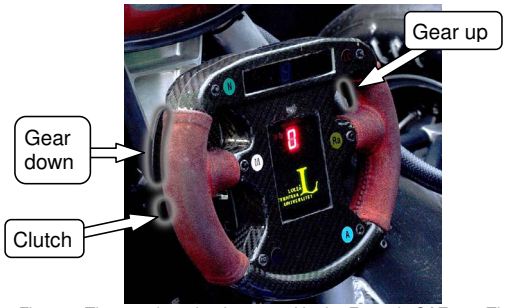


Figure 2 The steering wheel mounted in the Formula SAE car. The location of the gear shifting and clutch control paddles are marked in the picture.

The operation of the embedded code on receiving a message is in short:

- For each CAN message that matches the specified CAN id the `“CAN Object”` calls the `newData()` method with the new message as argument. Then the new clutch target is sent to the PID object by calling the `setTarget()` method with the new target as argument.
  If there is a shift command embedded into the message the actuate method is called, with the direction (up or down) as argument.
- When the `actuate()` method is run the busy flag is first checked to make sure that there is not already a shift operation in progress. If the flag is false the following takes place:
  - The busy flag is set to true.
  - The Soleniod is actuated by calling `setBit()`.
  - A “shift cut on” CAN message is sent out on the CAN bus by calling `canSend()`
  - A new message is scheduled after 50ms to `deActuate()`.

  `deActuate()` does almost the same as `actuate()`. Its behavior is summarized below:
  - The Soleniod is de-actuated by calling `clrBit()`.
  - A “shift cut off” CAN message is sent out on the CAN bus by calling `canSend()`.
  - The busy flag is cleared to false.

The closed loop control of the clutch actuator is done by `periodicFunc()` and for each execution the operation is:

- Fetch the current position by calling `getADValue()`.
- Call `pidCalc()` with the current position as argument.
- Check the sign of the result, and is the sign is negative then call `setBit()` otherwise call `clrBit()`.
- Call the `setDuty()` method with the absolute value of `pidCalc()` as argument.
- Schedule a message to `periodicFunc()` after 5ms (This is how a periodic process is realized.)

To further improve performance on the real car the software also has functions for measuring the position of the gearbox drum, and when a gear has been determined to have engaged the `deActuate()` method is run, this technique also makes it possible to shift into reverse. The position is also used for determining which gear that is actuated. And when down shifting the clutch is automatically actuated. Paper [5], M.Sc. thesis [6] and Bachelor thesis [7] goes into more details on this subject. The reason not to include the full functionality is just to improve the readability in this paper.

THE EXECUTABLE EMBEDDED SOFTWARE MODEL

The embedded C code presented in this paper can be considered as an executable model, as it complies with the Timber semantics, clear of side effects. Figure 3 illustrates how the same software model is used both in simulation and on the real system. Figure 4 shows an object and method view of the embedded software. In the figure a single arrow represents an asynchronous message, while a double arrow represents a synchronous request. Figure 5 depicts the source code for the “App” and “Gear change” object. The other modules are written in a similar manner. A complete implementation can be obtained from the authors.

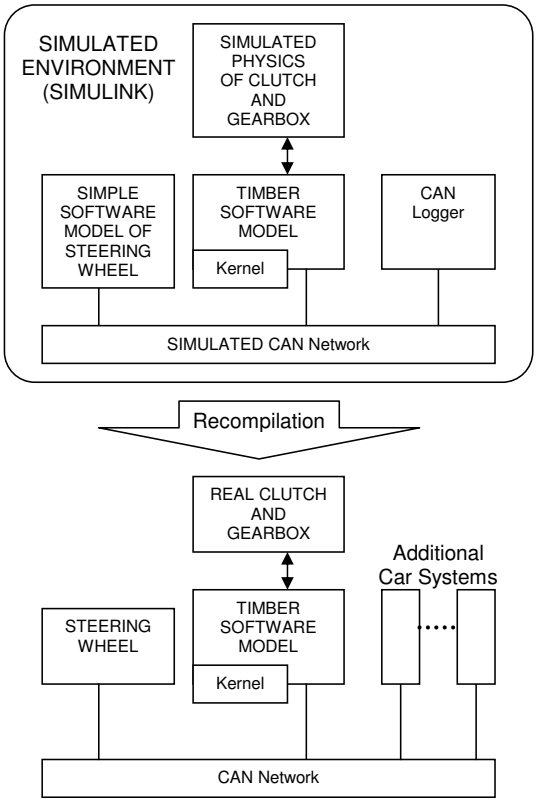


Figure 3 Model, showing the same software model is used in both simulation and in the real implementation

There are only two kernel primitives visible to the programmer. The synchronous request "SYNC(obj,meth,arg)" with the corresponding arguments:

- obj; Pointer to the called object.
- meth; Function pointer to the method to run.
- arg; Argument to (meth)

The baseline and deadline is inherited from the sender (the currently executing message). A "SYNC" message also returns with a value.

The other primitive is an asynchronous action "ASYNC(bl,dl,obj,meth,arg)" with the arguments:

- bl; The new baseline, relative to the current baseline.
- dl; The new deadline, relative to (bl).

- obj; Pointer to the called object.
- meth; Function pointer to the method to run.
- arg; Argument to (meth).

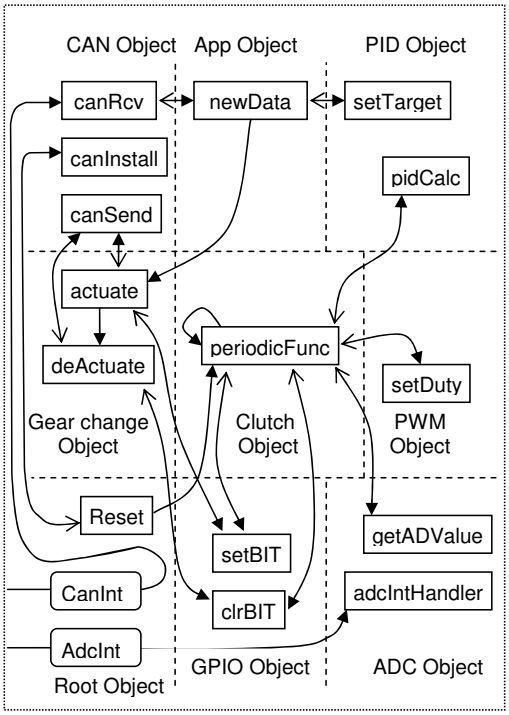


Figure 4 Object and method view of the executable Timber model

```
-----app.h-----
#include "tT.h"
#include "can.h"
#include "gear.h"
#include "pwm.h"

typedef struct{
  OBJECT obj;
  PwmObj *clutchpid;
  GearObj *gear;
} AppObj

void newData(AppObj *self,CanMess mess);

-----app.c-----
void newData(AppObj *self,CanMess mess){
  int clutchtarget = mess->clutchtarget;
  SYNC(self->clutchpid,&(mess->clutchtarget));

  switch(mess->shiftcmd){
  case SHIFTUP:
    ASYNC(MS(0),MS(1),self->gear,actuate,SHIFTUP);
  break;
  case SHIFTDOWN:
    ASYNC(MS(0),MS(1),self->gear,actuate,SHIFTDOWN);
  default:
```

```
  return;
  }
}

-----gear.h-----
#include "tT.h"
#include "can.h"
#include "gpio.h"

typedef struct{
  OBJECT obj;
  CanObj *can;
  GPIOObj *gpio;
  Bool busy;
} GearObj

void actuate(GearObj *self,int arg);

-----gear.c-----
void actuate(GearObj *self,int arg){
  if (self->busy)
    return;
  self->busy = 1;

  switch(arg){
  case SHIFTUP:
    SYNC(self->gpio setBIT,SHIFTUP_PORT);
  break;
  case SHIFTDOWN:
    SYNC(self->gpio,setBIT,SHIFTDOWN_PORT);
  default:
  return;
  }

  CanMess igncut = {CANID_IGNCUT,IGNCUT_ON};
  SYNC(self->can,canSend,&igncut);

  ASYNC(MS(50),MS(2),self->gear,deActuate,0);
}

void deActuate(GearObj *self,int arg){
  SYNC(self->gpio,clrBIT,SHIFTDOWN_PORT);
  SYNC(self->gpio,clrBIT,SHIFTUP_PORT);

  CanMess igncut = {CANID_IGNCUT,IGNCUT_OFF};
  SYNC(self->can,canSend,&igncut);
  self->busy = 0;
}

-----from root.c-----

..
..
// AppObj instantiation
AppObj app = {INITOBJECT(),&clutchpid,&can,&gear};
// Canlistener instantiation
CanListener steeringwheel =
{newData,&app,STEERINGWHEEL_ID}
..
..
void Reset(void){
..
..
  canInstall(&can,&steeringwheel);
..
..
```

Figure 5 Source code for the executable Timber model

VERIFICATION FRAMEWORK

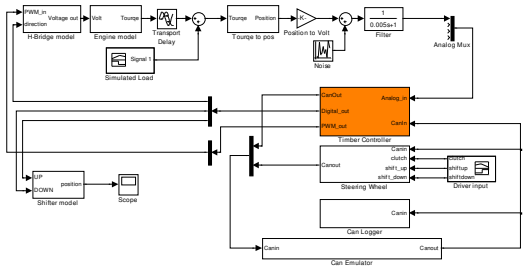


Figure 6 The Simulink co-simulation framework used in this case study

MATLAB/SIMULINK

Simulink® is used as a backbone to co-simulate mechanical system dynamics, embedded electronics, and embedded software. In our case study the mechanical system is represented by a continuous Simulink model, the embedded electronics under consideration is the CAN bus (modelled by an embedded Matlab function), a set of sensors and actuators (modelled by standard Simulink components), and a simplified model of the user interface (clutch and gearshift control modelled by standard Simulink components) together with an executable model of the embedded Timber software. During simulation we can observe and account for the effects of "permissible" timing behaviour specified for the software components. Figure 6 shows the simulation backbone together with all its components. This is a unique method for timing simulations within simulink based on formal semantics which cannot be directly compared to ordinary (informal) simulink modelling and autocoding, as disussed in [20].

COMPILING FOR SIMULINK

In order for the software to be executed under simulink the application code is compiled together with a simulink compatible kernel into a MEX file that can be run from simulink.

IMPLEMENTATION

The same software is implemented on the clutch and gearbox controller on the 2005 and 2006/2007 LTU formula SAE car. Figure 7 shows the embedded electronics platform as mounted in the car.

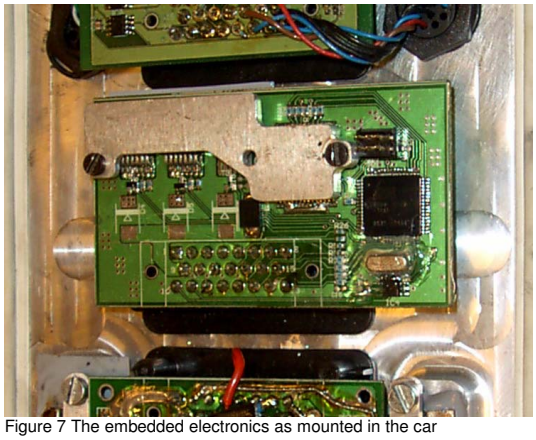


Figure 7 The embedded electronics as mounted in the car

The hardware platform is using a Atmel AT90CAN128 microcontroller, this microcontroller embeddes an AVR5 core and has 4k SRAM, 128k flash. The complete hardware platform with the entire car electronic is described in detail in Paper [5], and M.Sc. thesis [6]. However the embedded software has been reworked, using the Timber programming paradigm.

EVALUATION

The raw C-code implementation described in [7] took approximately three man months including background research, design and implementation, testing, and tuning. Approximately 2/3 of the work was directly related to code implementation.

The corresponding Timber C implementation took two work days including design and implementation, testing, and tuning. However some of this efficiency improvement was due to familiarity to the problem achieved during the raw C development.

In an accompanying paper [19] a traction controller is implemented and simulated, the coding of the embedded controller took just a few hours.

Furthermore, the Distributed Engine management system described in [5,6] have been reworked in Timber C and also in these cases the implementation effort was found to be greatly reduced.

FUTURE WORK

Ongoing and future Timber related research includes;

- finalizing the Timber compiler tool [11]
- implementation and proof of correctness for the real-time garbage collection algorithm[18], and
- formal methods [16,17] and model checking for Timber model analyses, and
- commodity libraries (with examples) for typical vehicular applications, such as controllers, transfer functions, i/o interfaces, loggers etc., and
- visualization of Timber models for specification, simulation and verification

CONCLUSION

In this paper we have demonstrated a design methodology for real-time embedded software that does not require “guru” like knowledge of RTOS primitives and programming advanced patterns [8].

In the presented framework, the embedded software is developed using Timber, a programming language and formalism that provides executable models for embedded real-time systems.

We have shown how a complete control system can be straightforwardly modeled, simulated and transformed into executable code using the Timber programming paradigm.

Through utilizing the proposed co-simulation verification framework, the complete Gear shift and Clutch controller was designed, verified, and deployed on the target in just a few work hours. Compared to the raw C code design flow undertaken in [6] and [7], the proposed framework was found to offer magnitudes in increased efficiency, this with respect to the implementation time. In comparison to raw C code, our case study has shown the framework to:

- eliminate hard-to-find errors, such as race-conditions, and to
- easy capture the timing behaviour, avoiding explicit coding of hardware timers, and to
- provide implicit parallelism by the object structure, eliminating hard coded interrupt based constructs to achieve parallel behaviour.

We boldly conclude that our Timber based framework offers true **“work with the work”.**

ACKNOWLEDGMENTS

Funding from Center for Automotive System Technologies and Testing through Norrbottens Forskningsråd are greatly acknowledged.

REFERENCES

1. Stensson A., Larsson T., “Industry demands on vehicle development – methods and tools”. *Vehicle System Dynamics Supplement 33 (1999), pp. 202-213.*

2. Schäuffele J., Zurawaka T., “Automotive Software Engineering – Principles, Processes, Methods, and Tools”, *SAE Publications, Warrendale, PA, 2005.*

3. Müller-Glaser K. D., Frick G., ”Multiparadigm Modeling in Embedded Systems Design”, *IEEE transaction on control systems technology, VOL. 12, March 2004.*

4. Amory A., Moraes F., Oliveira L., Calazans N., Hessel F., “A Heterogeneous and Distributed Co-Simulation Environment”, *Proceedings of the 15 th Symposium on Integrated Circuits and Systems Design, 2002.*

5. Eriksson J., Lindgren P., Deventer J., ”A Distributed Engine management system for formula SAE”, SAE International Detroit 2007

6. Eriksson J. “An Engine management system for Formula SAE”, M.Sc thesis at Luleå University of technology, 2006.

7. Malmgren D. “Automotive electronics and their implementation in a race car” Bachelor thesis at Luleå University of technology, 2006.

8. Kolnick F., The QNX 4 Real-time Operating System, Basis Computer Systems Inc. September, 1998 ISBN 0-921960-01-8.

9. Black A., Carlsson M., Jones M., Kieburtz R., and Nordlander J.” 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.

10. Lindgren P., Nordlander J., and Eriksson J. “Robust Real-Time Applications in Timber”. In Sixth IEEE International Conference on Electro, Information Tech, EIT, 2006.

11. “the Timber Developer Wiki”, http://hackage.haskell.org/trac/timber/wiki.

12. freeRTOS http://www.freertos.org/

13. Contiki http://www.sics.se/contiki/

14. TinyOS http://www.tinyos.net/

15. AmbientRT http://www.ambient-systems.net/ambient/technology-rtos.htm

16. Baker T. P. “A Stack-Based Resource Allocation Policy for Realtime Processes”, IEEE Real-Time Systems Symposium, pages 191–200, 1990.

17. Liu Y. A. and Gomez G. “Automatic Accurate Cost-Bound Analysis for High-Level Languages”, IEEE Transactions on Computers, 2001.

18. Kero M., Nordlander J., and Lindgren P. ” A Correct and Useful Incremental Copying Garbage Collector”, The 2007 Internatinal Symposium On Memory Management

19. Eriksson J., Nybacka M., Larsson T., and Lindgren P. ”Using Multi-body Simulation to design reliable embedded software – Evaluation and discussion of work method” Accedpted to 14th Asia Pacific Automotive Engineering Conference (APAC-14)

20. Tripakis S., Sofronis C., Caspi P.,and Curic A. “Translating Discrete-Time Simulink to Lustre” ACM Transactions on Embedded Computing Systems, Vol. 4, No. 4, November 2005, Pages 779–818.

CONTACT

Johan Eriksson, Ph. D. Student, Luleå University of Technology, EISLAB, SE-97187, Luleå, Sweden. Phone: +46920 491743, Email: johan.eriksson@ltu.se

Per Lindgren, Associate Professor, Luleå University of Technology, EISLAB, SE-97187, Luleå, Sweden. Phone: +46920 491092, Email: per.lindgren@ltu.se

DEFINITIONS, ACRONYMS, ABBREVIATIONS

HIL: Hardware-In-Loop

SIL: Software-In-Loop

PID: Proportional Integral Derivative

TIMBER: TIMe reactive emBEdded Real-time

GPIO: General Purpose Input/Output

Paper C

FIFO WiDOM : Timely Control Over Wireless Links.

Authors:
Viktor Leijon, Per Lindgren and Johan Eriksson

Reformatted version of paper originally published in:
16th IEEE International Conference on Control Applications Part of IEEE Multi-conference on Systems and Control Singapore, 1-3 October 2007

16th IEEE International Conference on Control Applications
Part of IEEE Multi-conference on Systems and Control
Singapore, 1-3 October 2007

WeA02.4

FIFO WiDOM: Timely control over wireless links

Viktor Leijon, Per Lindgren and Johan Eriksson

Luleå University of Technology
Department of Computer Science and Electrical Engineering
S-97187 Luleå, Sweden
{leijon,pln,johan.eriksson}@csee.ltu.se

Abstract

We present the idea of scheduling a control network in a first-come first-serve manner, and demonstrate a way to implement this over a wireless link. This achieves real-time guarantees and tightly bounded jitter for control applications. Further, how to design systems using this network is discussed, and the benefits compared to a standard priority MAC protocol are examined through simulations and a prototype implementation. The prototype is implemented using the reactive Timber programming model on a lightweight 8-bit platform.

I. Introduction

Control over wireless links is a rapidly developing field, and presents many new problems in control theory. From a computer engineering point of view, getting a stable communication channel with reliable timing properties is the main complication.

One of the challenges is that wireless nodes are often considered to be more loosely coupled than wired systems; adding and removing nodes may be done more dynamically. If predictable real-time properties are desired in a system such as this, temporal composability [1], as well as considerable flexibility, becomes important.

In previous work we have presented a protocol for FIFO networking [2], and outlined how it could be implemented on-top of the CAN protocol. In this paper we extend it to control a wireless link by combining it with work from Pereira *et al.* [3] on the wireless dominance protocol (WiDOM), creating the FIFO WiDOM MAC protocol.

Our approach gains flexibility and implementation agility compared to what is seen in TDMA systems, and is easy to implement on existing embedded systems with wireless capabilities, with only a modest cost.

Our contributions in this paper are:

- We review the basics of a FIFO based MAC protocol and describe the adaptations needed when implementing it on top of the Wireless dominance (WiDOM) MAC (Section II).
- Simulations of the system, comparing it to a priority based protocol are presented, in Section III.
- In Section IV we give a short introduction to reactive objects and the Timber programming model.
- Finally, we present a prototype implementation of FIFO WiDom for the AVR platform using IR communication. We utilize the programming model of Timber to capture the specification, and a complete protocol implementation is presented. (Section V).

II. Wireless dominance FIFO

A. An overview of FIFO networking

There are many approaches to deciding who gets to use a network at any given time. In a time-triggered (TDMA) network it is decided a priori (or at the set up of a communication session), and at runtime the static schedule decides when a node is allowed to transmit. In an event-triggered system some other method is needed; in the standard Aloha-like protocol chance decides who gets to transmit, and in a priority based system a priority is used to arbitrate between concurrent senders. Many systems also function as hybrids of two or more of the methods.

Our approach is to see the network as a FIFO (Figure 1), and decide who gets to transmit on the network next on the basis of how long a message has been waiting. The message with the longest accumulated waiting time is always transmitted next. If there is a fixed speed at which items are removed from the queue (δ time units per item), and a maximum length (N) of the queue, there will also be a maximum time to send a message.

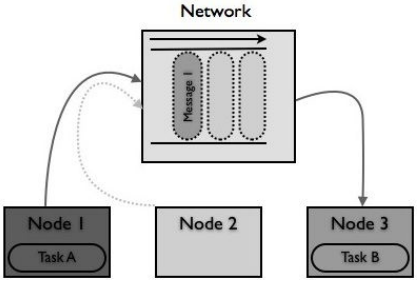


Fig. 1. A FIFO for messages

More details on the principles can be found in our original article [2], which also includes reasoning about how to design systems with the paradigm, calculating the queue length N, and how to obtain composability. That work carries over with little or no modification to FIFO WiDOM.

B. Using WiDom for FIFO networking

Dominance protocols in general have become very popular in wired control networks, in particular in the the CAN network [4], which is particularly common for automotive applications. More recently, Pereira *et al.* [3] have shown how to extend this to wireless networks in the form of the WiDOM protocol.

Analogous to our adaption of the FIFO principle to the CAN bus [2], we can extend WiDOM so that the network behaves like a FIFO as described in section II-A. We explain the principles and parameters of the FIFO WiDom adaptation.

In summary, dominance protocols are based on each message type having a static priority, and any collisions being resolved non-destructively, by an arbitration phase using the priorities of the colliding messages.

In our examples we will assume that 6 priority bits are used in the WiDOM implementation, but this can be extended if more nodes are required. In our prototype implementation (Section V) we have $\delta = 29.12$ ms.

We choose to divide the priority into two parts (Figure 2):

a) Waiting time: is the number of arbitration rounds that the node has lost. This value is reset whenever the node is allowed to transmit a message by winning an arbitration.

b) Node id: must be unique within each system and is used to break ties when two or more nodes have the same waiting time.

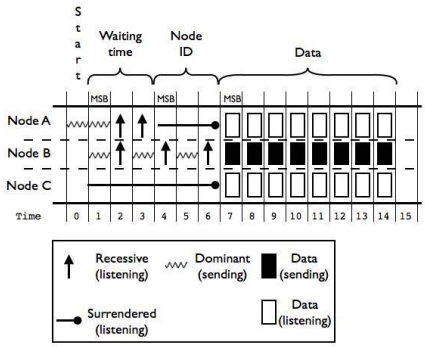


Fig. 2. Overview of a protocol frame.

C. Designing with FIFO WiDOM

The FIFO WiDOM behaves like a FIFO in the sense we explored in Section II-A, therefore the methodology outlined in our previous work [2] can be used to design a system which meets a given set of time constraints.

The distribution of bits between node id and waiting time depends on the distribution between queue slots and nodes. For instance, we may assign 3 bits for node identifier and 3 for waiting time, and thus get $2^3 = 8$ as the maximum number of slots in the queue. When using the maximum queue length the delay is $d = N\delta = 8\delta$ (which is 233 ms in our prototype).

D. Controlled jitter

Jitter control is considered to be especially important for control loops [5], where use of a time invariant control law is generally preferred. We propose a solution for situations where low jitter is preferable over early arrival of messages.

There is a maximum send time, δ, and we design our system with a pre-defined maximum queue size N. In our implementation the actual waiting time (the number of arbitration cycles the message has lost) is a part of the message. This means that if the waiting time portion of the message priority is p, we can always wait $(N-p)\delta$ to give all messages similar delivery times. The worst case jitter is bound by δ plus the maximum deviation in packet send times.

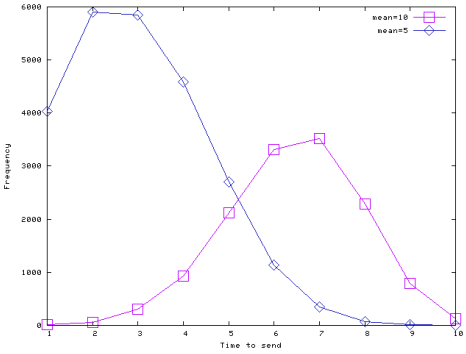


Fig. 3. Send time distributions for two different traffic intensities

Priority	Average delay	Max delay	std. dev
0	5.19	72	6.55
5	3.22	39	3.29
10	2.05	16	1.51
15	1.35	8	0.68
20	1.00	2	0.02

Fig. 4. Priority MAC, inter-arrival time 10

Priority	Average delay	Max delay	std. dev
0	225.06	1434	225.88
5	25.77	231	30.40
10	4.98	46	4.37
15	1.80	14	1.16
20	1.00	1	0.0

Fig. 5. Priority MAC, inter-arrival time 5

III. Simulations

To compare the FIFO approach to a more traditional priority based communication scheme we set up a series of simulations.

In the simulations we created a system with two communicating nodes to simulate a simple event loop, and a number of nodes that produce background traffic. The communicating nodes wait for a message to arrive, and when they have received a message they wait one packet time (to simulate performing some kind of computation) before sending it back. The background nodes each send messages, using an exponential distribution with varying mean intervals between sending. The system was simulated for 100000 packet times.

The number of background nodes was varied, and the case for ten generators of background traffic is studied in particular.

A. Simulation of the FIFO Mac

Setting the mean send time for the background traffic to 10, the average time to send was 3.06 units with a standard deviation of 1.50. Decreasing the interval for the background servers to 5, the average delay increases to 6.43 and the standard deviation is 1.47.

It is worth noting that the maximum delay is 10 in all cases, and that the jitter minimization technique outlined in Section II-D could be used to decrease the jitter even further. For instance, by delaying all packets until 6 packet times have passed in the first example decreases the standard deviation to 0.17.

A plot of the waiting time distribution is shown in Figure 3. We can see that the delays are highly dependent on the traffic load on the network, with increasing load the response time grows. The delay never goes over the maximum queue length.

B. Simulation of a priority based MAC

In this simulation the 10 background processes are assigned even priorities from 2 to 20. The processes are given priorities from 0 to 21. The results with inter-arrival time set to 5 is in Figure 5 and for 10 in Figure 4

Figure 6 shows priority versus average delay (with standard deviation as error bars) for the different traffic intensities.

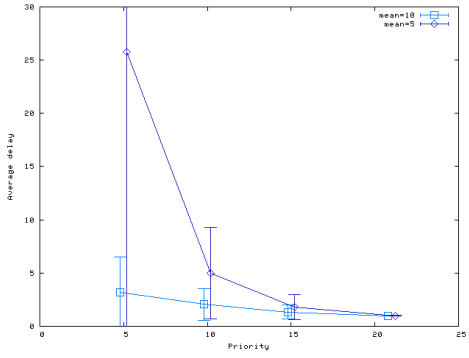


Fig. 6. Average delay as a function of the priority, for two different traffic intensities

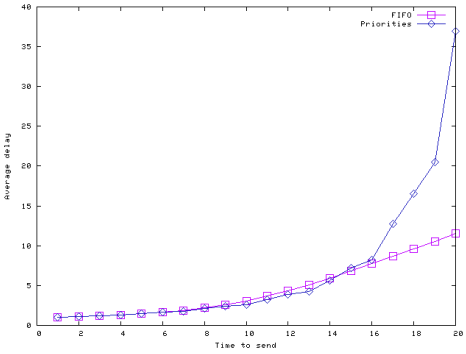


Fig. 7. A comparison of FIFO and priorities for varying traffic intensities.

C. Comparison

We compared the FIFO MAC and the priority mac at various traffic intensities, letting the interval for the background tasks vary from 1 to 20 packet times. The result is in Figure 7. While the FIFO and priority buses behave similarly for reasonable situations, the priority based traffic has unbounded delays and bad average case behavior when there is a lot of higher priority traffic.

We conclude that care is required when choosing priorities in a priority based scheme, and that selecting the wrong priorities can lead to undesirable consequences, in particular for systems with high worst case load. When constructing a system with priorities, it is important to perform a schedulability analysis [6].

The FIFO based approach has a worst case waiting time that is linear to the number of nodes, preventing starvation in the system, but gives the same guarantees to all messages in the system. In the original paper [2] we present extensions to combine priority levels with the FIFO semantics.

IV. Timber

Embedded real-time systems are naturally described as time-bound reactions to external events, a view supported natively in the high-level programming and systems modeling language Timber [7]. The engineering perspectives of the Timber design paradigm are further elaborated in [8].

In this paper, we implement the dominance protocol using the programming model of Timber, through a C interface (*tinyTimber, tT*) to a minimalistic, robust, and portable Timber run-time kernel. This allows for comprehensive and efficient implementation with predictable memory and timing behavior. The application interface of tT features the following subset of Timber:

- Concurrent reactive objects
- State protection
- Synchronous and asynchronous messages
- Deadline scheduling

The Timber run-time model utilizes deadline scheduling directly on basis of programmer declared event information. In short:

- *Objects and concurrency:* The concurrent 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 protection 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 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 timing constraints on the reaction to an event can be directly reused as run-time parameters for scheduling the corresponding method. The permissible window for method execution is defined by the *baseline* (the absolute point in time for event release), together with the relative *deadline*. Chain reactions are expressed through relating future events to the locally fixed baseline, and will be free from jitter induced by the actual scheduling.

On the top level, a tinyTimber application consists of the implicit root object and its methods, which are the functions installed for handling interrupts and the reset signal (main in C). 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.

A tinyTimber 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.

V. Implementation

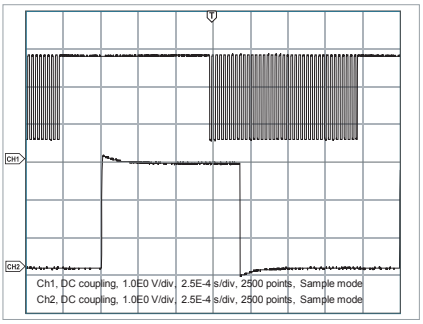


Fig. 8. Measurement of pulse detection delay

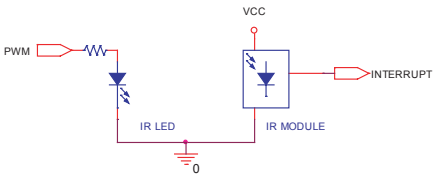


Fig. 9. Schematic of the receiver module

A. Hardware

To implement the protocol a board with an ATmega169 from ATMEL is used. The transmitter on the unit is an infrared diode that emits light at a wavelength of 940nm. The diode is modulated with a 38kHz square wave, generated by the hardware using a 8bit timer. A logic 1 is represented by a carrier and 0 as no carrier. The microcontroller platform is powered from a CR2450 lithium cell.

Receiving the modulated signal is done by using a infrared receiver module that detects the presence of the 38kHz carrier. The module signals a detected carrier by pulling the output signal low.

A measurement of a the 38kHz signal, modulated by 1ms pulses is shown in Figure 8. The time delay from the presence of a carrier to a low level on the receiver module is measured to approximately $250\mu s$.

A schematic showing the connection of the diode and receiver module is shown in Figure 9.

The ATmega169 used has a power consumption of 4 mA in active mode and 1.4 mA in idle. The microcontroller is in idle mode most of the time during both sending and receiving, this brings the average current consumption to 1.5mA. The maximum power consumption of the infrared receiver module is 2 mA. On a single cell, this gives an estimated lifetime for a system in receive mode of approximately one week, or a system sending over 8 million 16-bit frames.

B. Timber objects

The depicted code gives a complete implementation, besides of macros for tT kernel interface (`TT_*`) and IR hardware (`IR_*`). The protocol is implemented by an `irDom` object, (Listings 1, 3), which transmits and receives application data (Listing 2) over the IR interface.

The `irDom` object interface defines object structure, initiation mechanism, and externally accessible methods, (Listing 1). In `appObj` object implements a listener to the `irDomObj` echoing incoming data after one `FRAME`, (Listing 2). `PINCHANGE_INTR(void)` implements the handler for the external interrupt caused by the IR receiver on carrier change, (Nodes B and C, Time 0, Figure 2). `GETTIME()` may utilize available hardware "timestamping" (input capture) of interrupts, to improve timing accuracy. Accordingly, the message `TT_ASYNC(timestamp, SLACK, ...)` will be immediately due for execution, with the eligible execution window of `timestamp....timestamp+SLACK`. `irDomRecSync` will disable carrier change interrupts, and if data pending, skipping the *start* bit and entering the tournament at Time 1 (Node B), or skipping the header bits and start receive at Time 7.5 (Node C).

In `irDomTournament` we first check if we have survived through the complete header (won the tournament), if so enter `irDomTransmit` (Node B, Time 7). We proceed by checking if our header bit is dominant, if so we activate our carrier and proceed with the next bit in order (Nodes A and B, Time 1). On a recessive bit, we deactivate our carrier and issue a listen for carrier halfway into the period (Nodes A and B, Time 2). In `irDomRecessive` we check for incoming carrier, if so we have lost the tournament (Node A, Time 3.5) and skip remaining header bits before start receiving data (Node A, Time 7.5). If we have data pending, we increase our FIFO priority (Node A, Time 3.5). In `irDomReceive` (Nodes A and C, Time 7.5...14.5) and `irDomTransmit` (Node B, Time 7...14) we receive and transmit data correspondingly.

When received finished, data is sent to the installed "callback" (Nodes A and C, Time 14.5). When transmit finished, we deactivate our carrier (Node B, Time 15). At Time 15.5, we clear pending carrier change interrupts and enable new. If data pending we schedule a new packet send (Node A, Time 15.5).

It is worth to note that the `irDomObj` object will be responsive to `irDomSend` events throughout the complete protocol execution. However, the current implementation does not support buffering of more than one data element,

```
#define PERIOD          MSEC(2 * 0.889)
#define SLACK           (0.2 * PERIOD)

#define NRPRIOSTAT      3 // static priority, ID
#define NRPRIOFIFO      3 // 0 for static priorities only
#define NRPRIO          (NRPRIOSTAT + NRPRIOFIFO)
#define NRDATA          8
#define SLOT            ((NRDATA + NRPRIO + 2) * PERIOD)

typedef struct {
    TT_OBJ o;           // Object state
    TT_OBJ* listenObj;  // Listener Object
    TT_METH listenMeth; // Listener Method
    BOOL busy;
    BOOL dataPending;
    short prio;         // [NRPRIOBITS]
    short data;         // [NRDATABITS]
    short rec;          // [NRDATABITS]
} IrDom;

#define initIrDom(obj, meth) {TT_INIT_OBJ(), obj, meth, FALSE, FALSE}

void irDomPrio(IrDom* self, int arg);
BOOL irDomSend(IrDom* self, int arg);
void irDomRecSync(IrDom* self, int arg);
```

Listing 1. irDom.h

```
typedef struct {
    TT_OBJ o;
    idDom* ir;
    ...
} App;

void appListen(appObj* self, int arg) {
    TT_ASYNC(PERIOD, SLACK, self->ir, irDomSend, arg + 1);
    ...
}

extern IrDom irDomObj;
App appObj = {TT_INIT_OBJ(), &irDomObj, ...};
IrDom irDomObj = initIrDom(&appObj, appListen, IR_MAC);

PINCHANGE_INTR(void)Ê{
    TT_SAVECONTEXT();
    Time timestamp = TT_GETTIME();
    ...
    IR_DISABLE_PINCHANGE_INTR();
    TT_ASYNC(timestamp, SLACK, &irDomObj, irDomRecSync, IR_IS_CARRIER());
    TT_SCHEDULE();
    TT_RESTORECONTEXT();
}

int main(void) {
    TT_INIT();      // kernel init
    IR_INIT();      // IR hardware initiation
    IR_CARRIER_OFF();
    IR_ENABLE_PINCHANGE_INTR();
    TT_IDLE();      // enter kernel idle
}
```

Listing 2. root.c

and will return FALSE if data is already pending. Furthermore, due to space limitations, the example code does not implement header information for received packages. A complete implementation is available from the authors.

VI. Related work

Mosley performed some early work on general FIFO networking [9] in the context of the tree-algorithms for retransmitting proposed by Capetanakis [10].

The work on WiDOM is due to Pereira, Andersson and Tovar [3], and includes the extension of response time analysis to that protocol.

There have been several approaches to real time guarantees for CAN, in particular the work of Tindell *et al.* [6] provided a foundation based on scheduling theory for

```
void irDomPrio(IrDom* self, int arg) {
  self->prio = arg;
}

BOOL irDomSend(IrDom* self, int arg) {
  if (self->dataPending) {
    return FALSE;
  }
  self->data = arg;
  self->dataPending = TRUE;
  self->prio &= (1 << NRPRIOSTAT) - 1; // (enter FIFO)
  irDomSendSync(self, 0);
  return TRUE;
}

void irDomSendSync(IrDom* self, int arg) {
  if (self->busy) {
    self->busy = TRUE;
    IR_DISABLE_PINCHANGE_INTR();
    IR_CARRIER_ON();
    TT_ASYNC(PERIOD, SLACK, self, irDomTournament, NRPRIO);
  }
}

void irDomRecSync(IrDom* self, int arg) {
  if (self->busy) {
    return;
  }
  self->busy = TRUE;

  if (self->dataPending) {
    TT_ASYNC(PERIOD, SLACK, self, irDomTournament, NRPRIO);
  } else {
    TT_ASYNC((NRPRIO + 1.5) * PERIOD, SLACK, self, irDomReceive, NRDATA);
  }
}

void irDomTournament(IrDom* self, int arg) {
  if (arg == 0) {
    irDomTransmit(self, NRDATA);
    return;
  }
  arg--;
  if (self->prio & (1 << arg)) { // dominant bit
    IR_CARRIER_ON();
    TT_ASYNC(PERIOD, SLACK, self, irDomTournament, arg);
  } else { // recessive bit
    IR_CARRIER_OFF();
    TT_ASYNC(0.5 * PERIOD, SLACK, self, irDomRecessive, arg);
  }
}

void irDomRecessive(IrDom* self, int arg) {
  if (IR_IS_CARRIER()) { // carrier, tournament lost
    TT_ASYNC((arg + 1) * PERIOD, SLACK, self, irDomReceive, NRDATA);
    if (NRPRIOFIFO) { // FIFO mode
      short fifo = ((self->prio >> NRPRIOSTAT) + 1) & ((1 << NRPRIOFIFO) - 1);
      if (fifo == 0) { // fifo overrun
        TT_PANIC("irDomRecessive",_FIFO_overrun", 0);
      }
      self->prio = (self->prio & ((1 << NRPRIOSTAT) - 1))|(fifo << NRPRIOSTAT);
    }
  } else { // no carrier, survived recessive bit
    TT_ASYNC(0.5 * PERIOD, SLACK, self, irDomTournament, arg);
  }
}

void irDomReceive(IrDom* self, int arg) {
  arg--;
  if (IR_IS_CARRIER()) { // carrier, databit = 1
    self->rec |= 1 << arg;
  }
  if (arg == 0) { // finished
    self->busy = FALSE;
    TT_ASYNC(PERIOD, SLACK, self, irDomInterPackage, arg);
    TT_ASYNC(PERIOD, SLACK, self->listenObj, self->listenMeth, self->rec);
  } else {
    TT_ASYNC(PERIOD, SLACK, self, irDomReceive, arg);
  }
}

void irDomTransmit(IrDom* self, int arg) {
  if (arg == 0) { // finished
    IR_CARRIER_OFF();
    self->busy = FALSE;
    self->dataPending = FALSE;
    TT_ASYNC(PERIOD * 0.5, SLACK, self, irDomInterPackage, 0);
    return;
  }
  arg--;
  if (self->data & (1 << arg)) {
    IR_CARRIER_ON();
  } else {
    IR_CARRIER_OFF();
  }
  TT_ASYNC(PERIOD, SLACK*PERIOD, self, irDomTransmit, arg);
}

void irDomInterPackage(IrDom* self, int arg) {
  IR_CLEAR_PINCHANGE_INTR();
  IR_ENABLE_PINCHANGE_INTR();
  if (self->dataPending) {
    irDomSendSync(self, arg);
  }
}
```

Listing 3. irDom.c

calculatin gworst case response times in a network where each message has a fixed minimum period. There are some recent improvements to this work by Bril, *et al.* [11].

In the time-triggered realm, a lot of work has been done on guaranteeing timing properties. One early time-triggered effort was the time-triggered protocol [12] and the time trigged architecture [13]. Kopetz and Obermaisser has stressed the importance of temporal composability [1]. In trying to understand the differences between CAN and TTA, it is valuable to read his study [14]. There has also been time-triggered variants on CAN, most notably TTCAN [4] and FTT-CAN [15].

The technology which most obviously resembles the FIFO approach is that of the timed token network [16], where a token is sent around the network. If "being first in the queue" is seen as an imaginary token passed around, our approach could be described in terms of a "virtual token"-based network. Observe that our *virtual token* will only be passed to nodes which actually want to transmit something, and will never be lost if a node crashes.

In the context of minimalist operating systems, tiny Timber stands out with its deadline-driven scheduling and the heritage of the reactive object paradigm of Timber. This is achieved while matching the resource requirements and performance metrics of popular kernels such as TinyOS [17], Contiki [18], FreeRTOS [19] and AmbientRT [20].

VII. Conclusions and future work

We have examined the possibility of using a FIFO-based protocol for wireless control. Based on the simulations and theoretical results, we conclude that this MAC has suitable timing properties, such as predictable max delays and controllable jitter.

The implementation also demonstrates the relative simplicity of implementing the FIFO WiDOM MAC on an embedded platform, and measurements show that it is viable for medium term deployment on battery powered nodes. The Timber programming model has been demonstrated to efficiently capture the reactive behavior of the system, and a complete protocol implementation has been presented.

We have not covered fault-tolerance or the effects of errors in general but these issues are largely orthogonal to the time related properties. To what extent this holds true, and how existing fault-tolerance methods could be integrated, should be investigated.

In the future, a larger case study involving a real-world control application would be of interest, to examine how the MAC protocol would work in a more realistic setting.

References

[1] H. Kopetz and R. Obermaisser, "Temporal composability," *Computing and Control Engineering Journal*, vol. 13, no. 4, pp. 156–162, 2002. [Online]. Available: http://ieeexplore.ieee.org/iel5/2218/22129/01029796.pdf

[2] V. Leijon, "FIFO networking: Punctual event-triggered networking," 2006, **Unpublished manuscript:** http://www.csee.ltu.se/~leijon/publications/fifo-06.pdf.

[3] N. Pereira, B. Andersson, and E. Tovar, "Implementation of a dominance protocol for wireless medium access," in *RTCSA '06: Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 162–172.

[4] ISO, *11898 Road Vehicle - Interchange of Digital Information - Controller Area Network (CAN) for High-Speed Communication"*. CENELEC, 2003.

[5] A. Albert, "Comparison of event-triggered and time-triggered concepts with regard to distributed control systems," in *Embedded World*, 2004, pp. 235 – 252.

[6] K. Tindell, A. Burns, and A. Wellings, "Calculating controller area network (CAN) message response times," Feb. 16 1995. [Online]. Available: http://citeseer.ist.psu.edu/291115.html;http://www.cis.ksu.edu/~neilsen/classes/cis721/papers/DCCS_TBW_94.ps

[7] 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.

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

[9] J. Mosely, "An efficient contention resolution algorithm for multiple access channels," Master's thesis, Massachusetts Institute of Technology, May, 1979.

[10] J. Capetanakis, "The multiple access broadcast channel: Protocol and capacity considerations," Ph.D. dissertation, Massachusetts Institute of Technology, 1977.

[11] R. Bril, J. Lukkien, R. Davis, and A. Burns, "Message response time analysis for ideal controller area network (CAN) refuted," in *the 5th International Workshop on Real-Time Networks (RTN 06)*, July 2000.

[12] S. Poledna and G. Kroiss, "The time-triggered communication protocol TTP/C," *Real-Time Magazine*, vol. 4, no. 98, pp. 100–102, 1998.

[13] H. Kopetz and G. Bauer, "The Time-Triggered Architecture," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112 – 126, Jan. 2003.

[14] H. Kopetz, "A comparison of CAN and TTP," in *15th IFAC Workshop on Distributed Computer Control Systems*, 1998. [Online]. Available: citeseer.ist.psu.edu/kopetz98comparison.html

[15] L. Almeida, P. Pedreiras, and J. Fonseca, "The FTT-CAN protocol: why and how," *Industrial Electronics, IEEE Transactions on*, vol. 49, no. 6, pp. 1189–1201, December 2002.

[16] N. Malcolm and W. Zhao, "The timed-token protocol for real-time communications," *Computer*, vol. 27, no. 1, pp. 35–41, 1994.

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

[18] 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.

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

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

Paper D

IP Over CAN : Transparent Vehicular to Infrastructure Access.

Authors:
Per Lindgren, Simon Aittamaa, Johan Eriksson

Accepted for publication in:
Fifth IEEE Consumer Communications & Networking Conference, CCNC, Las Vegas

IP Over CAN, Transparent Vehicular to Infrastructure Access

Per Lindgren
EISLAB
Luleå University of Technology
97187 Luleå
Email: Per.Lindgren@ltu.se

Simon Aittamaa
EISLAB
Luleå University of Technology
97187 Luleå
Email: Simon.Aittamaa@ltu.se

Johan Eriksson
EISLAB
Luleå University of Technology
97187 Luleå
Email: Johan.Eriksson@ltu.se

***Abstract*— For the future we foresee each vehicle to feature wireless communication (to the Internet and/or other vehicles) over various technologies, e.g., UMTS/GPRS, and WLAN/WiFi. In this paper we show how access to such communication resources could be granted to individual components (CAN bus connected ECUs) in the car by allowing transparent data transport using the standardized Internet Protocol (IP). Our experiments show that a complete IP Over CAN implementation, providing both UDP and TCP transport over IP, running on an Atmel AT90CAN128 is capable of transfer speeds up to 200 kbits while using less than 2 kbytes of dynamic RAM.**

I. INTRODUCTION

For the future we foresee each vehicle to feature communication over various technologies, e.g., UMTS/GRPS, WLAN, and other wireless media. The introduction of such technology to vehicular applications will (arguably) be the next big thing, with impact spanning from:

- machine-to-machine issues such as testing during development, maintenance, traffic control, active safety etc.,
- man-machine issues such to enhance the driving experience by context aware information, such as heads up road signs, traffic information (congestion, road-work etc.) all the way to
- man-man issues, such as communication and sharing information with fellow drivers in the nearby.

Such functionality could indeed be accomplished by proprietary solutions, but the history so far points clearly in the direction of standardized and "open" protocols and interfaces. Hence, the Internet Protocol (IP) will be a strong contender for standardized communication in and between vehicles. A major advantage of TCP/IP all-the-way to the end nodes (ECUs), is that the gateway (relaying inter and intra communication) can be designed to operate ignorant from the actual communicating applications. This scenario has been acknowledged e.g. by the upcoming Media Oriented System Transport (MOST [1]) protocol, endorsed e.g. by BMW, Daimler-Chrysler, and Harman/Becker, with its native TCP/IP support. However, the dominant intra vehicle communication technology (for the last decade and for the imminent future), the CAN bus, commonly lacks support for IP transport.

II. IP ALL-THE-WAY

A driving scenario urging for IP all-the-way is end-to-end applications where (at least) one of the endpoints is an in-car unit (ECU). IP-based communication has the advantage of being a mature and well-established standard, for which techniques for authorization, security/encryption and addressing/multi-homing has been developed over the last 30 years. However, the CAN bus, as being the dominant intra vehicle communication technology (for the last decade and for the imminent future), commonly lacks support for IP transport and hence is a hurdle for the development of IP-based end-to-end applications.

Providing IP all-the-way from in-car ECUs to the Internet or to other vehicles, is not solely a question of how to facilitate IP communication over the in-car network(s), but also raises the questions of vehicle-to-infrastructure, and vehicle-to-vehicle communication. These subjects are research topics in their own rights and as such out of scope of this paper.

III. LIGHTWEIGHT IMPLEMENTATION

The work in this paper is based on an existing lightweight modular stack (MODEM-IP) [2] developed at Luleå University of Technology. For the purpose of developing a lightweight IP Over CAN implementation only the IP, TCP, and UDP layers. The Data-Link-Layer and the Physical-Layer is replaced by IP Over Can and the physical CAN-Bus, respectively. The IP Over CAN was based on the proposed protocol in [3].

Each ECU is given a number of predefined static IDs. From the IDs a CAN ID is produced (Figure 1). As an option, to maintain a symmetric priority for reply, the source and destination fields can be swapped (SwapSD bit set '1'). However, this requires two mailboxes, corresponding to the SwapSD = '0'/'1'. The control bit is used to create a control channel for pre-allocating bandwidth/buffer space. It could also be used for various other tasks, such as DHCP etc.

The CAN ID assignment is slightly different than the proposed protocol, the only major difference is the SwapSD. The data layout is slightly changed to exclude the sequence counter, mainly to reduce the overhead and code complexity. Most of the functionality provided by the sequence counter is implemented using the control channel.

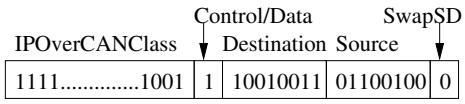


Fig. 1. IP Over CAN ID Layout.

A similar effort to standardize IP Over CAN has been proposed to IETF but is still considered a work in progress [4]. However, the protocol proposed to IETF utilizes almost all of the address bits, making it quite inflexible and could easily interfere with the real-time traffic on the network.

The current implementation uses less than 2 kbytes dynamic RAM and 48 kbytes flash (not optimized for code-size).

IV. Experimental Setup

The IP-Over-CAN protocol implementation has been validated on a set of CAN bus connected ECUs on a Formula SAE car developed at Luleå University of Technology [5]. The ECUs are based on the Atmel AVR AT90CAN128 micro controller clocked at 16MHz, featuring a total of 4 kbyte RAM, and 128 kbyte Flash Memory [6]. The ECUs run a minimalistic real-time Timber kernel featuring [7]:

- Concurrently executing state protected objects
- Time constrained synchronous and asynchronous messages
- Deadline scheduling of message execution

The MODEM-IP and its IP-Over-CAN protocol has been developed to exploit object orientation for modularity, and timed messages to realize internal communication between objects and to capture the timing constraints of the specification.

A set of experiments have been conducted on the reference implementation:

- Setting up and tearing down end-to-end TCP-IP sessions over the CAN network.
- Max throughput tests for IP over CAN has been conducted under a set of different CAN bus load conditions.

The number of connections manageable is limited only by the amount of memory assigned to the TCP layer. The current implementation requires 67 Bytes for storing the per-session TCP state, and another 40 Bytes for an ACK network buffer. As a connection is allowed to be immediately reused after termination, the stack imposes no performance limitation (other than that of the CPUs and network) to setting up and tearing down connections. The three-way-handshake of a TCP connection setup takes roughly 40 CAN-frames, about 50000 bits and 0.05seconds.

The ping test has been conducted using different buffer sizes and periodicities.

The communication is done using UDP, packets are sent in one direction only. Our tests show that the current stack implementation running on the given platform supports transfer speeds up to approximately 200 kbits payload. During the measurements an external bus-load of approximately 400 kbits was present.

Payload (bytes)	Period (ms)	Total Bandwidth Utilization	UDP and IP Overhead
512	20	45%	5.5%
256	10	43%	11%
128	20	37%	22%
128	10	37%	22%
64	20	30%	44%
64	10	30%	44%
64	5	30%	44%

TABLE I
EXPERIMENTAL THROUGHPUT TEST RESULTS.

The CAN bit-rate used in this test is 1Mbit/s. From the test we can see the bandwidth utilization almost reaches 50% as the Payload size is increased (see Table I). The limiting factor is the CAN bus overhead of at least 100%, that comes from the SOF, arbitration, control, CRC, ACK and EOF fields that have a fixed length of 64bits. The UDP and IP overhead is fixed and does not scale with the payload size,

The measurements have been conducted by connecting a PC to the CAN Network, using a USB to CAN interface (Kvaser USBCANII) and a CAN debugging software from ATI (ATI CANLab). The external buss-load is also generated from the PC.

V. Conclusions

In this paper we have shown an IP all-the-way solution for CAN networks. The problem of mapping IP transport over the CAN bus with minimal protocol overhead has been discussed, and the IP-Over-CAN protocol proposal has been presented. Furthermore, we have presented a generic modular, lightweight IP stack implementation suitable for embedded platforms. We have shown how the IP-Over-CAN protocol can be seamlessly integrated in the modular stack architecture. The proposed protocol has been evaluated by experiments conducted on a reference implementation running on a set of CAN bus connected ECUs. Our experiments show that a complete IP-Over-CAN implementation, providing both UDP and TCP transport over IP, can operate efficiently using less than 2 kbytes of dynamic RAM.

References

[1] "MOST Cooperation Homepage." [Online]. Available: http://www.mostcooperation.com/
[2] Isak Rova, Simon Aittamaa, "A modular TCP/IP stack for embedded systems with a tinyTimber interface," Mar. 2007.
[3] Michael Ditze, Reinhard Bernhardi, Guido Kamper, Peter Altenbernd, "Porting the Internet Protocol to the Controller Area Network," in *RTLIA*, 2003.
[4] Petr Cach, Petr Fiedler, "IP Over CAN," Mar. 2001.
[5] J. Eriksson, P. Lindgren, and J. can Deventer, "A Distributed Engine Management System for Formula SAE," in *SAE Journal, 2007, document number 2007-01-1602*, 2007.
[6] "Atmel product web:CAN Networking." [Online]. Available: http://www.atmel.com/products/CAN/
[7] P. Lindgren, J. Nordlander, and J. Eriksson, "Robust Real-Time Applications in Timber," in *Sixth IEEE International Conference on Electro,Information Tech, EIT*, 2006.

Paper E

Using Timber in a multi-body design environment to develop reliable embedded software.

Authors:
Johan Eriksson, Mikael Nybacka, Tobias Larsson, Per Lindgren

Accepted for publication in:
SAE World Congress 2008, Detroit

Using Timber in a multi-body design environment to develop reliable embedded software

Johan Eriksson, Mikael Nybacka, Tobias Larsson and Per Lindgren
Luleå University of Technology, Center for Automotive System Technologies and Testing

ABSTRACT

A major challenge for the automotive industry is to reduce the development time while meeting quality assessments for their products. This calls for new design methodologies and tools that scale with the increasing amount and complexity of embedded systems in today's vehicles.

In this paper we undertake an approach to embedded software design based on executable models expressed in the high-level modelling paradigm of Timber. In this paper we extend previous work on Timber with a multi-paradigm design environment, aiming to bridge the gap between engineering disciplines by multi-body co-simulation of vehicle dynamics, embedded electronics, and embedded executable models. Its feasibility is demonstrated on a case study of a typical automotive application (traction control), and its potential advantages are discussed, as highlighted below:

- shorter time to market through concurrent, co-operative distributed engineering, and
- reduced cost through adequate system design and dimensioning, and
- improved efficiency of the design process through migration and reuse of executable software components, and
- reduced need for hardware testing, by specification verification on the executable model early in the design process, and
- improved quality, by enabling formal methods for verification.

INTRODUCTION

A major challenge for the automotive industry is to reduce the development time while meeting quality assessments for their products. This calls for new design methodologies and tools that scale with the increasing amount and complexity of embedded systems of today's vehicles [14]. To keep and create competitive advantages parts of the embedded software and hardware components have to be developed in-house, which in turn will raise the need for collaboration with suppliers that enables development without disclosing company specific knowledge.

There is a trend in the automotive industry that electronic system development is going from hardware- to software-based solutions [12]. The advantages are, in general, improved flexibility, and lower cost for development, production and maintenance. With this in mind the engineers need tools that can help them in development of their embedded software. Executable models and automatic code generation can be of great help towards an efficient design flow and avoid time consuming and error prone embedded system programming, see for e.g., the Ptolemy project [1], and commercially available UML based approaches such as Rational Rose RT [2].

Design of automotive systems is a truly multi-disciplinary task, spanning from physics, machine elements, and vehicle dynamics, all the way through computer engineering, control theory, computer communication, to man-machine interaction and lifestyle issues. Hence a design methodology embracing a multi-body systems view throughout specification, (system modelling), verification (simulation, resource- and FME-analysis, etc.) and validation (visualization, testing, etc.), is in high demand.

In addition to this, using a common development interface for collaboration and distributed co-simulation could help the engineers at manufacturer and supplier to develop their individual components concurrently. Such a

multi-body co-simulation environment would allow functionality in terms of software to be developed for a model of the physical system dynamics, and vice versa, allow physical parameters to be optimized under given software behaviour. Furthermore, in such a distributed environment, functionality in terms of software components could be developed and refined concurrently at manufacturers and suppliers.

Attempts in this direction can be found in Paper [15], where a heterogeneous and distributed co-simulation environment is presented. The simulator backplane reads a file describing how the modules are connected and manages the communication between the modules as messages during simulation. One drawback is however that the modules need to be adapted to the simulator, hence do not directly reflect their true implementation.

In this paper we undertake an approach to embedded software design based on executable models expressed in the high-level modelling paradigm of Timber [3]. Timber with its object orientation and inherent real-time capability has shown to offer a comprehensive approach to the design of embedded real-time software for controlling mechanical systems [21]. In this paper we extend previous work on Timber with a multi-paradigm design environment, aiming to bridge the gap between engineering disciplines by multi-body co-simulation of vehicle dynamics, embedded electronics, and embedded executable models. Its feasibility is demonstrated on a case study of a typical automotive application (traction control).

BACKGROUND

CO-SIMULATION

There are a number of Multi-body System analysis (MBS) software's that are used for, vehicle dynamic analyses, Hardware-In-the-Loop (HIL) and Software-In-the-Loop (SIL). Some of these are listed below:

- VI-Car RealTime, from VI-Grade
- CarSim™, from Mechanical Simulation Corporation
- CarMaker®, from IPG Automotive
- DYNAware, from Tesis Group
- ASM, from dSPACE
- VDMS with MATLAB®/Simulink®, from Milliken Research Associates and Mathworks respectively.
- Etc.

All of these are applicable when performing co-simulations and when vehicle dynamics are needed in the simulation loop. Due to the increasing speed of the solvers for dynamic vehicle simulation it is possible to simulate the dynamics with smaller step sizes which will in turn increase the accuracy of the SIL and HIL simulations. Using SIL and HIL will significantly speed up the development process for embedded software, as testing can be carried out in a real-life like setting prior to vehicle prototype production.

Past work by the authors when it comes to co-simulation and vehicle validation in general are represented in paper [10], where the ideas behind a future Distributed Real-Time Simulation and Validation framework are presented and discussed. Paper [11] addresses the industry demands on methods and tools for mechatronic vehicle development.

FRAMEWORK

MATLAB® / SIMULINK®

Simulink® is used as a backplane to co-simulate vehicle dynamics, embedded electronics, and embedded software. Simulink mediates between simulator components (in our case Simulink native functions, embedded Matlab (M-functions), and S-functions (such as the Timber simulator(s) and the CarSim™ solver). Figure 1 shows the Simulink co-simulation backplane used in this case study. Simulink offers a familiar user interface and commodity functionality for stimuli, logging and visualization.

CARSIM™ SOFTWARE

To represent the vehicle behaviour the CarSim™ dynamic simulation software for four-wheel vehicles is used. It can easily be connected to Simulink® and represented as an S-function in the same. It is also possible to extend the math model of CarSim™. This is useful when the user want to integrate control systems to the model, modify the vehicle model or change the outputs of the model.

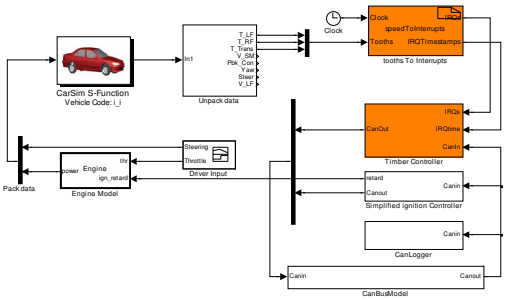


Figure 1. The Simulink framework

TIMBER

Timber adopts the reactive paradigms proven successful for hardware description languages (such as Verilog and VHDL), and offers abstractions of modern functional programming languages, e.g., higher order functions, type inference and objects. The engineering aspects of Timber are further elaborated in [4] and [21]. Up-to-date information is found online at the Timber development Wiki [22].

Real-Time System Modelling

Timber allows the intended timing behaviour to be straightforwardly expressed directly in the model, as time-bound reactions operating on stateful objects. All realizations that meet the specified timing constraints are "permissible" and to be considered correct, much like one regards components complying to their data sheets as being acceptable. Using a traditional real-time operating system based approach, timing behaviour is controlled by setting priority levels, either manually or by utilizing some automated approach [22]. In any case, the actual timing behaviour of such a system will be dependent on the load conditions (other executing tasks) and the performance of the target platform, which both may fluctuate over time. Hence, at this stage, from the software description (with its relative priority levels) only a snapshot view of the system's timing behaviour can be obtained during simulation. However, using Timber, we have the potential to observe and account for the effects of "permissible" timing behaviour of distributed software components, already during simulation.

Object Orientation and Parallelism

Timber objects execute in parallel, (much like you would expect hardware components to live their own life). However, operations on a specific object execute exclusively (forcing other operations on the same object to wait). In this way the state of the object is protected at all times. Traditionally, this is a major obstacle in reactive (real-time) programming, as manually accounting for state protection through RTOS primitives such as locks/semaphores/monitors etc. requires guru-like knowledge [5].

Advanced features of the Timber language

Timber pays heritage to functional languages and comprises a strong type system, featuring type inference, polymorphism, subtyping and higher order functions. Furthermore, the Timber semantics allows for efficient real-time garbage collection[24], thus relieving the programmer from tedious and error prone manual memory management. Altogether, object oriented abstraction, advanced type system, automatic state protection, and real-time garbage collection helps to reduce the risk of hard-to-find programming errors, such as race conditions, dead-locks, memory leakages, and pointer related errors.

System Environment

As in "real life", the embedded code will reside in, and interface to its environment. For simulation purposes this interface can be emulated by a model of the environment, (eventually to be replaced by the actual hardware). In the early design stages of specification verification, we might settle for an abstract model, and later refine it towards accurately reflecting properties of the hardware realization. With increased refinement, the software model comes closer to the actual production code. In such a way, hardware/software co-design can be carried out to the fullest. Throughout this process, we can during simulation observe and account for the use of shared resources (e.g., CAN busses, memories etc.), leading us to a feasible partitioning.

Timber and Formal Methods for Analysis

As mentioned, the observed behaviour during simulation express the "permissible" timing defined by the Timber models. Hence, during the design phase we *assume* that execution of the Timber models will meet the timing requirements of the specification. This leaves us with a separate question, namely whether execution on a *specific* hardware platform will meet the timing requirements under a given load. Traditionally, this requires excessive testing (by HIL or on real vehicle). However, the Timber semantics is defined to underpin formal methods for verification (such as schedulability [6] and resource analysis [7]), which could be exploited to reduce or even circumvent these time consuming and costly testing activities. Furthermore, formal methods have the potential to actually prove that conditions (such as timing constraints or memory sufficiency) are met under *all* given circumstances. Traditional testing can at best provide evidence under a representative set of cases, (as exhaustive methods are clearly unfeasible). This potential use of formal methods may lead to new liability policies, e.g., a sub-contractor who can show that his subsystem has been formally proven correct for usage according to specification should not be held responsible if used outside this specification (which by proof is the only case where it may actually malfunction).

Timber realizations

For a given target architecture, Timber specifications can be compiled into a subset of C, supported by e.g. the gcc tool suit. The resulting C code is compiled and linked with the Timber run-time kernel (which implements the messaging, garbage collection and scheduling mechanisms), see Figure 2. For a bare-chip target, the executable image will not rely on any additional libraries; hence the system is under full control by the Timber kernel.

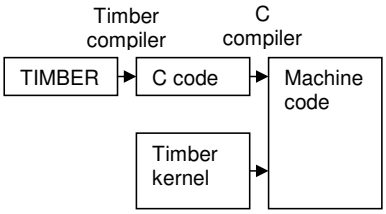


Figure 2. From Timber specification to executable code.

VEHICLE VALIDATION VISUALIZATION

The vehicle validation visualization (v^3) application used in this paper is presented in more detail in [11]. The application is a work in progress to help engineers work and collaborate at distance. The application is written in Java™ and use a Java™ based 3D engine called AgentFX™ [16], it is therefore platform independent and it is also possible to start the application from a web browser via Web Start [17]. Having the possibility to reach the application via the web is a necessity when it comes to distributed collaborative work. We will use the v^3 application in this work as a tool for distributed co-simulation, where the engineers do not have to work at the same physical location to share and view the simulation results, see Figure 3.

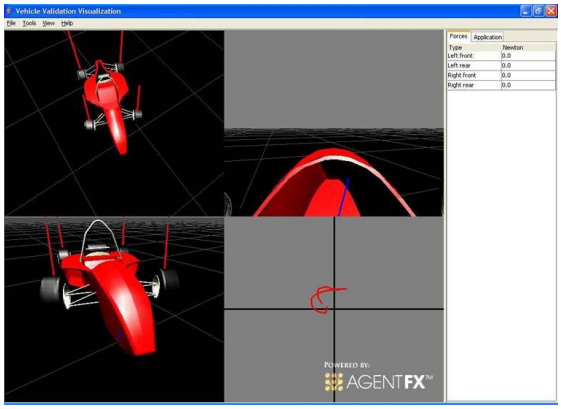


Figure 3. Vehicle Validation Visualization application used during co-simulation.

WORK AND WORK METHODS

CO-SIMULATION

In this paper we propose a design and simulation environment for Timber. By exploiting the unique feature of time-bound reactions, the "permissible" timing behaviour of the system is revealed. Notice that the software description in Timber (with its time-bound reactions) holds a complete view of the system's timing behaviour, completely free of effects caused by system load and platform performance. In this way software components can be reused in a safe manner in different settings, while retaining their intended real-time behaviour.

Throughout the development process co-simulation has been used to simulate the behaviour of the complete system. Figure 4 shows how the simulation model corresponds to the real system.

Vehicle dynamics

Data to build up a CarSim™ model have been gathered from ADAMS/Car™, so the Kinematics and Compliance (K&C) tests normally performed on real life systems are done virtually in ADAMS/Car™.

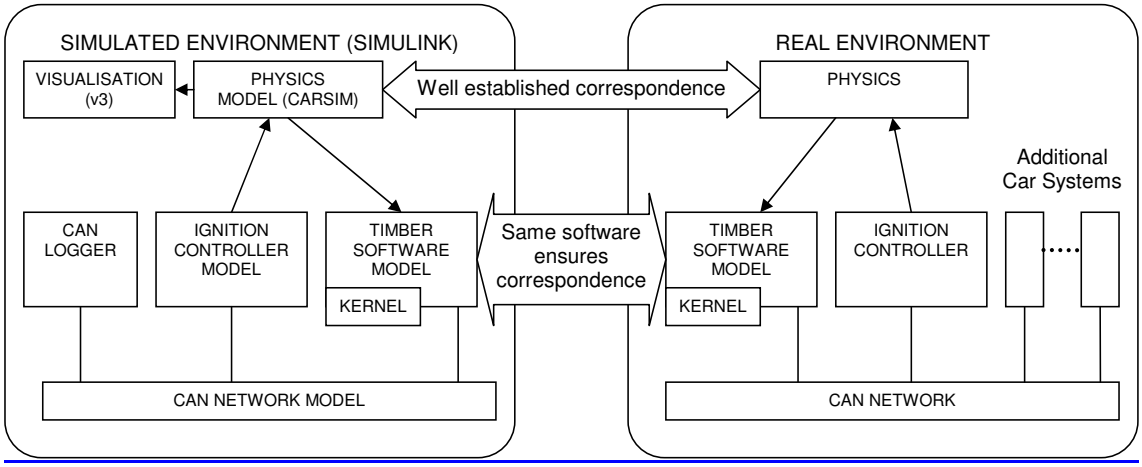


Figure 4. Illustration showing that the same Timber software model is used throughout the design process.

The dynamic model of the vehicle, in this case a Formula SAE car, has been continually developed during years of Formula SAE projects and final thesis works. The moments of inertia have been both measured in real life and obtained from CAD tools.

Development of traction control

The traction controller consists of a number of objects. Figure 5 shows an object view of the system. The operation is in short;

- For every wheel interrupt the `wheelIntHandler()` method is called, if the signal is considered valid it send the new speed to `setWheelSpeed()`.
- The periodical process `calcIgnition()` fetches the slip by calling `getSlip()` and feeds it to `calcPID()`. The result of `calcPID()` is send out on the CAN bus by calling `canSend()`. Then a message is scheduled in 10ms to `calcIgnition()`, (this is how a periodic process is created).

The wheel slip estimator getSlip(),translates the individual wheel speeds into a slip coefficient λ.

$$\lambda = \frac{\left(v_b - \left(v_{fl} + v_{fr}\right)/2\right)}{\left(v_{fl} + v_{fr}\right)/2}, \text{ where } \begin{cases} \lambda : wheelslip \\ v_b : backwheelspeed \\ v_{fl} : leftfrontspeed \\ v_{fr} : rightfrontspeed \end{cases}$$

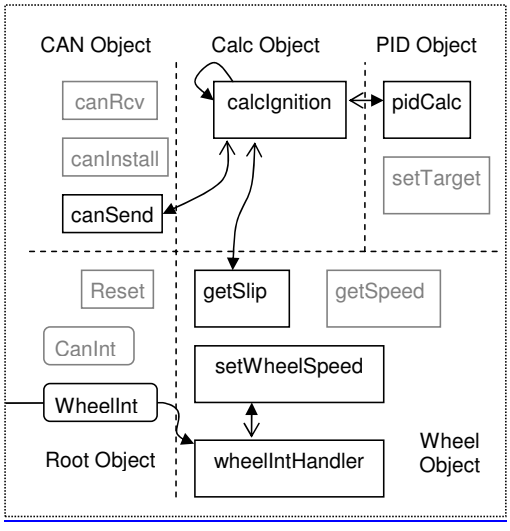


Figure 5. Timber Source

A standard PID module is used to control the ignition retard. The input into this module is the estimated slip and the traction target. The traction target is found by manually looking at a "friction vs. slip rate" (known as $\mu - \lambda$) or "tractive-force vs. slip rate" curve for the tires used. The parameters for the PID controller are found experimentally in the simulated environment.

By examining the curve in Figure 5 it is clearly visible that the optimum slip ratio is around 0.1 regardless of the tire load and temperature. The tire data we have used has been obtained from "The Formula SAE Tire Test Consortium"[18].

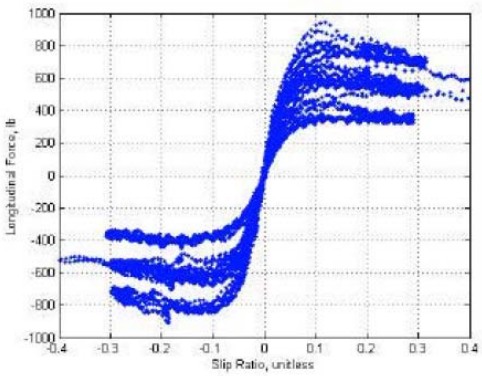


Figure 6. Slip ratio curves in different conditions for a Formula SAE tire [18].

SYSTEM VERIFICATION

The system verification process has been carried out under the proposed framework, (see section FRAMEWORK). Executable models of the embedded Timber specifications have been designed using Timber-C, a direct programming interface to the Timber Run-Time system.

The system behaviour has been verified (showing stable performance) for the extremes and normal distributions of "permissible" execution windows. The 3D visualization has proven useful for real-time feedback during the development process.

IMPLEMENTATION

The executable Timber models have been compiled for the target platform, a Philips LPC2119 microcontroller featuring a 32bit ARM7 core and 16k SRAM, with 128k flash. The complete hardware platform with the entire car

electronic is described in detail in paper [19] and M.Sc. thesis [20]. For details and comparison of the Timber C Run-Time system (TinyTimber) see [25].

FUTURE WORK

Ongoing and future Timber related research includes;

- finalizing the Timber compiler tool, and
- formal methods and model checking for Timber model analyses, and
- commodity libraries (with examples) for typical vehicular applications, such as controllers, transfer functions, i/o interfaces, loggers etc., and
- visualization of Timber models for specification, simulation and verification.

Ongoing and future work on Distributed co-simulation Visualization;

- a common interface supporting video conference, 3D visualization of vehicle simulation, and
- an environment for collaboration and data distribution for co-operative work.

Future work on real-time test-data transport;

- protocol and wireless technology for real-time test-data transport between vehicle and OEM/supplier,
- sensor network for data acquisition

CONCLUSION

We have demonstrated a multi-paradigm design environment to Timber, bridging the gap between engineering disciplines by multi-body co-simulation of vehicle dynamics, embedded electronics, and embedded executable models. Its feasibility has been shown through a case study of a typical automotive application (traction control), and its potential advantages have been discussed in the following topics:

- shorter time to market through concurrent, co-operative distributed engineering, and
- reduced cost through adequate system design and dimensioning, and
- improved efficiency, through migration and reuse of executable software components, and
- reduced need for hardware testing, by specification verification early in the design process, and
- improved quality, by opening up for formal methods for verification.

ACKNOWLEDGMENTS

The funding from Center for Automotive System Technologies and Testing through Norrbottens Forskningsråd, Calspan and the Formula SAE TTC is greatly acknowledged.

CONTACT

Johan Eriksson, Ph. D. Student, Luleå University of Technology, EISLAB, SE-97187, Luleå, Sweden. Phone: +46920 491743, Email: johan.eriksson@ltu.se

Mikael Nybacka, Ph. D. Student, Luleå University of Technology, Division of Computer Aided Design, SE-97187, Luleå, Sweden. Phone: +46920 491698, Email: mikael.nybacka@ltu.se

Tobias Larsson, Associate Professor, Luleå University of Technology, Division of Computer Aided Design, SE-97187, Luleå, Sweden. Phone: +46920 493043, Email: tobias.c.larsson@ltu.se

Per Lindgren, Associate Professor, Luleå University of Technology, EISLAB, SE-97187, Luleå, Sweden. Phone: +46920 491092, Email: per.lindgren@ltu.se

DEFINITIONS, ACRONYMS, ABBREVIATIONS

HIL: Hardware-In-Loop

SIL: Software-In-Loop

PID: Proportional Integral Derivative

TIMBER: TIMe reactive emBEdded Real-time

REFERENCES

1. Edward A. Lee, “Overview of the Ptolemy Project”, Technical Memorandum No. UCB/ERL M03/25, University of California, Berkeley, CA, 94720, USA, July 2, 2003.

2. IBM Software - Rational Rose - Product Overview: IBM Rational Rose Technical Developer [online 2007-02-21] http://www.ibm.com/software/awdtools/developer/technical/

3. 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.

4. P. Lindgren, J. Nordlander, and J. Eriksson. “Robust Real-Time Applications in Timber”. In Sixth IEEE International Conference on Electro,Information Tech, EIT, 2006.

5. Frank Kolnick, The QNX 4 Real-time Operating System, Basis Computer Systems Inc. September, 1998 ISBN 0-921960-01-8.

6. T. P. Baker. “A Stack-Based Resource Allocation Policy for Realtime Processes”, IEEE Real-Time Systems Symposium, pages 191–200, 1990.

7. Y. A. Liu and G. Gomez. “Automatic Accurate Cost-Bound Analysis for High-Level Languages”, IEEE Transactions on Computers, 2001.

8. Liu C-S., Monkaba V., Lee H., Alexander T., Subramanyam V., ”Co-simulation of Driveline Torque Bias Controls”, SAE paper 2001-01-2782, 2001.

9. D’Silva S., Sundaram P., D’Ambroslo J., ”Co-Simulation Platform for Diagnostic Development of a Controlled Chassis System”, SAE paper 2006-01-1058, 2006.

10. Nybacka M., Larsson T., Johanson M., Törlind P. “Distributed Real-Time Vehicle Validation”, DETC2006-99154, Proceedings of ASME IDETC/CIE, 2006.

11. Nybacka M., Larsson T., Karlsson T. “Vehicle Validation Visualization”, VC_InCo2006_P56, Proceedings of Virtual Concepts 2006.

12. Stensson A., Larsson T., Merkt T., Schuller J., Williams R. A., Mauer L., “Industry demands on vehicle development – methods and tools”. Vehicle System Dynamics Supplement, vol. 33, pp. 202-213, Swets & Zeitlinger, 1999.

13. Schäuffele J., Zurawaka T., Automotive Software Engineering – Principles, Processes, Methods, and Tools, SAE Publications, Warrendale, PA, 2005.

14. Müller-Glaser K. D., Frick G., Sax E., Kühl M., ”Multiparadigm Modeling in Embedded Systems Design”, IEEE transaction on control systems technology, vol. 12, March 2004.

15. Amory A., Moraes F., Oliveira L., Calazans N., Hessel F., “A Heterogeneous and Distributed Co-Simulation Environment”, Proceedings of the 15 th Symposium on Integrated Circuits and Systems Design, 2002.

16. AgentFX™, www.agency9.se.

17. Java Web Start, http://java.sun.com/products/javawebstart/index.jsp.

18. Kasprzak E. M., Gentz D., ”The Formula SAE Tire Test Consortium – Tire Testing and Data Handling”, SAE paper 2006-01-3606, 2006.

19. Eriksson J., Lindgren P., Deventer J., ”A Distributed Engine management system for formula SAE”, SAE International Detroit 2007

20. Eriksson J. “An Engine management system for Formula SAE”, M.Sc thesis at Luleå University of technology, 2006.

21. Eriksson J., Lindgren P “A comprehensive approach to design of embedded real-time software for controlling mechanical systems”, 14th Asia Pacific Automotive Engineering Conference (APAC-14)

22. “The Timber Developer Wiki”, http://hackage.haskell.org/trac/timber/wiki

23. Saksena M., Freedman P., Rodziewicz P. “Guidelines for automated implementation of executable object oriented models for real-time embedded control systems”, Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)

24. Kero M., Nordlander J, Lindgren P. ”A Correct and useful incremental copying garbage collector” accepted for publication at The 2007 International Symposium on Memory Management (ISMM 2007)

25. Lindgren P., Eriksson J., Aittamaa S., Norlander J. ” TinyTimber, Reactive Objects in C for Real-Time Embedded Systems” accepted for publication at DATE 2008 - Design, Automation and Test in Europe.

Paper F

Comprehensive Reactive Real-Time Programming

Authors:
Per Lindgren, Johan Nordlander, Kalevi Hyyppä, Simon Aittamaa, Johan Eriksson

Accepted for publication in:
2008 Hawaii International Conference on Education, Hawaii

1

Comprehensive Reactive Real-Time Programming

Per Lindgren, Johan Nordlander, Kalevi Hyyppä, Simon Aittamaa, Johan Eriksson
EISLAB
Luleå University of Technology
97187 Luleå
Email: {per.lindgren, johan.nordlander, kalevi.hyyppa, simon.aittamaa, johan.eriksson}@ltu.se

Abstract

The programming of computer controlled devices (embedded systems) is a common engineering task in ever increasing industrial demand. Hence, courses in reactive, and real-time programming have become mandatory parts of computer science and computer engineering curricula. Their common goal is to teach students how a computer system should be programmed to react to events (internal and external) to the system, under certain timing constraints. However, this fairly simple concept is often clouded by artifacts of today's programming languages and operating systems, which has rendered embedded programing a reputation of being cumbersome, error prone and requiring expert programming skills. In this paper, we dispute this view by presenting an alternative approach to reactive real-time programming based on the notion of timed reactions. Using TinyTimber - a minimal set of comprehensible software constructs together with a portable real-time kernel - the programmer can straightforwardly express the sought system behavior. In this paper we discuss timed reactions in the light of didactical advantages over traditional thread based approaches. Furthermore, we report on first results of applying TinyTimber to computer science education, both in the field of reactive, real-time programming courses and as applied in the field of project based mechatronics education. Results so far are most promising, showing students to have an increased ability to understand and solve embedded programming assignments.

I. Introduction and background

Courses in reactive and real-time programming have become a mandatory part of computer science and computer engineering curricula. Their existence are motivated from the ever growing market of computer controlled devices, such as home and vehicular electronics, and industrial applications. These courses all share the common goal to teach students how a computer system should be programmed such to react to concurrent events (internal and external) to the system, under some given timing constraints. However, this fairly simple concept is often clouded by enforcing the use of today's programming languages and operating systems. The traditional way to realize real-time systems is based on threads, that run concurrently on the system under control of a real-time kernel [1], [2], [3], [4], [5], [6]. The main obstacle of the threaded model is maintaining state integrity of shared resources through blocking [7]. This burden is put onto the programmer, a tedious and error prone task requiring in depth knowledge of the real-time kernel primitives. Furthermore, the by far dominating way to code timing behavior is by assigning priorities to the different threads. A higher priority thread is scheduled in precedence to lower prioritized threads (given that it is not blocked). However, a reaction can involve numerous threads, and each thread can involve numerous resources (and hence risk potential blockings). The malicious effects are twofold; on the one hand, the setting of the priority levels such to meet desired timing requirements of the system reactions is a delicate task, requiring a view of the complete system - and on the other hand, once the system is coded under this thread based model, sole inspection of the resulting code gives little help to understanding the system in terms of its timed reactions.

Didactically, this becomes a tough challenge, as the means to express the system behavior in terms of threads is conceptually far from the intuitive notion of a system reacting to events under certain timing constraints. Moreover, the traditional thread based approach requires the programmer to master a substantial set of complex kernel primitives and their inner workings. In order to help the programmer to understand the complex interaction between threads, several visualization tools have been developed, for example [8], [9]. Furthermore, to give a simplified and protected environment for teaching thread based programming, libraries [10] and tools have been developed [11]. Another approach is taken by the Ada95 langauage [12] and the the Real-Time extension to Java [1]. Their inherent support for (to some extent) real-time concurrent programming, relieves the programmer of a complex kernel interface, at the price of additional language constructs. However the main obstacles of threaded modeling of reactive real-time systems still remains.

From the research community, alternative views of reactive, concurrent and real-time programming have been investigated. Languages such as Estrel, Lustre and Signal have been designed with the outset to express reactive (event driven systems) [13]. Design methodologies and tools such as the Ptolemy framework, HOOD, HRT-HOOD, PAMELA, DARTS, CODARTS as well as UML based approaches (e.g. Rose RT) have been introduced to support the development of embedded systems. (For an excellent overview we refer the reader to [7].) However, from an educational point of view, these languages and tools are highly complex, and falls out of scope for this paper, seeking a comprehensive approach to education in reactive real-time programming.

Altogether, embedded programing has gained a reputation of being cumbersome, error prone and requiring expert programming skills. In this paper, we dispute this view by presenting an alternative approach to reactive real-time programming based on the notion of timed reactions. In Section II, we present TinyTimber - a minimal set of comprehensible software

constructs together with a portable real-time kernel, providing the programmer the means to straightforwardly express the sought system behavior. Furthermore, we discuss timed reactions in the light of didactical advantages over traditional thread based approaches. In Section III, we report on some first results of applying TinyTimber to computer science education, both in the field of reactive, real-time programming courses and as applied in the field of project based mechatronics education. In Section IV the so far promising results are concluded, as students exposed to TinyTimber have shown to have gained an increased ability to understand and solve embedded programming assignments compared to previous classes undertaken a traditional thread based system model.

II. TINYTIMBER

In this paper, we introduce TinyTimber (TT) - a minimalistic, robust, portable real-time kernel with predictable memory and timing behavior - and explore how the Timber semantics for reactive objects can be effectively applied to embedded real-time programming in C. Through the TT implementation, developers are provided a C interface, allowing legacy code to be migrated into the pure reactive design paradigm of Timber. For an in depth description of Timber, we refer to the draft language report [14], the formal semantics definition [15], and previous work on reactive objects [16], [17] and functional languages [18]. The application interface of TT features the following subset of Timber;

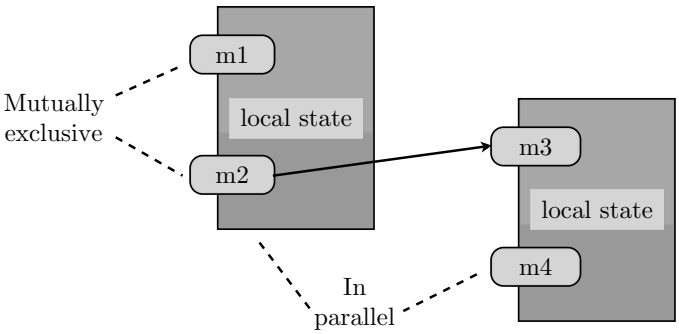
- Concurrent objects
- State protection
- Synchronous and asynchronous messages
- Deadline scheduling

On designing the TT kernel, utter care has been taken to offer bounded memory and timing behavior, controllable by compile time parameters. The kernel in 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 to third party code (even c-lib), which in turn benefits portability and robustness.

In the context of other minimalistic operating systems, TT stands out with its deadline driven scheduling and the heritage paid to the high level modeling paradigm of the Timber language. Another outstanding feature is the implicit state protection mechanism that follows directly from the modularization. A similar implicit locking mechanism for tasks are applied in Ambient-RT [19].

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 concurrency and object orientation into the concept of concurrent state-protected objects (resources). The execution model of Timber ensures mutual exclusion between the methods of an object instance. In this way Timber conveniently captures inherent parallelism of a system without burdening the programmer with the traditional explicit coding using processes/threads/semaphores/monitors, etc [7]. As designed with real-time applications in mind, the language provides the notion of timed reactions, where each method call is given an absolute time-window for execution, expressed as the event *baseline and* deadline. Events are either generated from the environment (typically as interrupts as in the case of software realizations of Timber models) or through synchronous/asynchrnous message sends expressed directly in the language (not as OS primitives). The Timber run-time model assumes deadline scheduling directly based on event information (base- and deadlines), which circumvents the problem of turning deadlines into process priorities. In short;

Fig. 1. Concurrency model for Timber.



- 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, Figure 1. This implicit coding of parallelism and state integrity coincides with the intuition of a reactive object. In a correct Timber program, all methods are non-blocking, hence the system will be responsive to incoming events at all times.
- Events, methods and time: The semantics of Timber conceptually ties events and methods in a way that makes it possible to unify the timely requirements for a reaction to an event, with the run-time demands on the execution of a method. The baseline states the absolute time for the release of an event, i.e. the point in time when the corresponding method becomes eligible for scheduling. Points in time relative to the event baseline will be free of jitter due to the actual scheduling of events.

A. The TinyTimber application interface

TinyTimber (TT) provides a C interface to a real-time kernel. Through the use of TT, the programmer is provided; concurrent reactive objects (`Object`) with state protection, synchronous and asynchronous messages under deadline scheduling. In case of synchronous messages (`SYNC`), the base- and deadlines are inherited from the sending event, while asynchronous messages allow extending the baseline and giving a new relative deadline (`ASYNC`). The run-time model of Timber assumes run-to-end semantics for object methods, hence we may never block for spurious external events. TT requires the programmer to strictly conform to this reactive programming style, and follow the stipulated use of TT primitives. Notice, though, that the type system of C is not powerful enough to ensure these requirements to be met. Furthermore, the programmer will be exposed to some of the "behind the scenes" work carried out by the Timber kernel, details of the implementation that is abstracted away from the Timber programmer.

III. TINYTIMBER IN RESEARCH AND EDUCATION

Programming in TinyTimber requires only a few basic primitives, hence many common yet hard-to-find errors can be avoided. In spite of its simplicity, TinyTimber is capable of expressing concurrent, real-time systems with automatic state protection. In this way, TinyTimber allows the programmer to focus more on the problem solution rather than the actual implementation.

In this paper we focus on how TinyTimber makes it easier for a person without prior knowledge of real-time reactive programming to create a correct real-time reactive program. This does not mean that it can not be utilized by a seasoned real-time programmer as illustrated by [20] applied to research in automotive engineering and computer communication [21].

In the following, we present and discuss first results applying TinyTimber to undergraduate education - in the context of courses in real-time systems and in mechatronics.

A. Real-Time Systems

In the course Real-Time Systems [22] given at Luleå University of Technology, the students (mainly second year) are exposed to reactive, real-time programming and TinyTimber. In their lab assignments, the students step by step implement the core functionality of a TinyTimber kernel. The labs are based on a light weight AVR-5 (8-bit architecture) platform. In the final lab the students control a physical robotic arm by TinyTimber programming.

The following assignment was given as part of the written examination.

> Exam assignment 7
>
> "Implement the class of a reactive object that offers **streaming** functionality. More precisely, the task is to translate a sequence of incoming messages arriving at arbitrary times, into a flow of outgoing messages emitted at a constant rate. To facilitate such behavior, a streaming object will need to contain some buffer storage, whose size we refer to as `N`. The desired period of the outgoing flow should furthermore be `T` milliseconds. Your class should export a method stream to be called each time an incoming message arrives. Outgoing messages should invoke a method `M` of some receiving object `R`.
>
> Define your class so that `N`, `T`, `M` and `R` can be set independently for each instance of your class. You may assume that the data payload for each message is a single integer.
>
> Three special cases to consider:
>
> 1) Any messages arriving when the buffer storage is full should simply be discarded.
> 2) Should the streaming object find the buffer storage empty when it is due to emit a new message, the current message flow should be considered aborted.
> 3) Any message arriving when there is no ongoing flow should immediately start a new flow with period T."

Listing 1. TinyTimber Stream Mediator Solution

```
#include <TinyTimber.h>

struct mediator_t {
    Object self;
    int delay;
    int messages;
    int messages_max;
    Time previous;
    int (*method)(Object *, int);
    Object *object;
};
#define initMediator(N, T, M, R) {initObject(), T, 0, N, MILLISEC(0), M, R}

static void mediator_M(struct mediator_t *self, int data) {
    self->messages--;
    SYNC(self->object, self->method, data);
}

void mediator_stream(struct mediator_t *self, int data) {
    if (self->messages >= self->messages_max)
        return;
    self->messages++;
    if ((BASELINE()-self->previous) > self->delay)
        self->previous = BASELINE();
    else
        self->previous += self->delay;
    ASYNC(self->previous - BASELINE(), MILLISEC(1), self, mediator_M, data);
}

/* Declaration/Initialization. */
struct mediator_t mediator = initMediator(N, T, M, R);

/* Invocation. */
SYNC(&mediator, mediator_stream, 10);
/* Or */
ASYNC(BL, DL, &mediator, mediator_stream, 10);
```

B. Example Solutions

We present two different solutions to the examination assignment in Section III-A, based on TinyTimber and FreeRTOS([5]) respectively. FreeRTOS is chosen as being good representative to the traditional thread based view on reactive real-time programming. As in the case of TinyTimber, FreeRTOS is also targeted towards small embedded systems and portable across multiple platforms. With these solutions we would like to show not only the differences between the two approaches, but more interestingly what we consider to be their strengths and weaknesses.

1) TinyTimber: The TinyTimber solution (listing 1) is very straight forward and does not involve any buffer, only a counter that keeps track of the number of messages that are currently waiting to be sent as well as when the previous message was sent. The only primitives used are `SYNC`, `ASYNC`, and `BASELINE`. Given that you understand these primitives the behaviour of the code is easy to understand and the real-time constraints are clearly visible in the code.

2) FreeRTOS: The FreeRTOS solution (listing 2) to the problem is quite different from the TinyTimber version, and relies on more built-in primitives. Instead of the TinyTimber object we have chosen to use a semaphore to represent the "object" R. Each stream mediator must create a task (thread) to invoke the method M at the appropriate time. The task is given the arbitrary priority of $idle + 1$. Since we do not know the timing constraints of the other threads that might be running, it is impossible to assign a meaningfull priority without a complete system view. Each mediator also requires a queue that acts as a buffer and communication channel. The FreeRTOS queues are thread-safe, thus we do not need to manually ensure their state integrity. We also supply an initialization method, since threads and queues cannot be allocated statically. In this version of the solution we use 8 built-in primitives (and rely heavily on their behavior), we can also see that there is a possible source for jitter between the `xQueueReceive` and `xTaskGeTTickCount` calls. This is impossible to avoid without using a built-in primitive that gives us a timestamp when the an item is received from the queue (currently not supported by FreeRTOS).

Listing 2. FreeRTOS Stream Mediator Solution

```
/* Scheduler includes. */
#include "FreeRTOS.h"
#include "task.h"

#define MEDIATOR_STACK_SIZE 64

struct mediator_t {
    int delay;
    int (*method)(int);
    xSemaphoreHandle send_sem;
    xTaskHandle task;
    xQueueHandle queue;
};

static void mediatorTask(struct mediator_t *mediator) {
    int data;
    porTTickType xLastWakeTime;
    while(1) {
        xQueueReceive(mediator->queue, &data, portMAX_DELAY);
        /* Possible Jitter. */
        xLastWakeTime = xTaskGeTTickCount();
        do {
            xSemaphoreTake(mediator->send_sem, portMAX_DELAY);
            mediator->method(data);
            xSempahoreGive(mediator->send_sem, portMAX_DELAY);
            vTaskDelayUntil(&xLastWakeTime, mediator->delay);
        } while (xQueueReceive(mediator->queue, &data, 0) != pdFALSE);
    }
}

void mediator_init(struct mediator_t *mediator, int N, int T, int (*M)(int),
                   xSempahoreHandle R) {
    mediator->delay = T;
    mediator->method = M;
    mediator->send_sem = R;
    mediator->queue = xQueueCreate(N, sizeof(int));
    xTaskCreate(mediatorTask, "mediatorTask", MEDIATOR_STACK_SIZE, mediator,
                tskIDLE_PRIORITY+1, &mediator->task);
}


void mediator_stream(struct mediator_t *mediator, int data) {
    xQueueSend(mediator->queue, &data, 0);
}

/* Declaration/Initialization. */
struct mediator_t mediator;
mediator_init(&mediator, N, T, M, R);

/* Invocation. */
mediator_stream(&mediator, 10);
```

C. Solution Evaluation

When comparing the two solutions we can see that the size, modularity, and scalabilility of the code are fundamentally different. The real-time constraints were not given in the assignment. However, in the context of real-time systems, the timing constraints are crucial to the system behavior. In the TinyTimber version we can clearly see that the deadline (allowed jitter) is 1 millisecond for each invocation of the method `M`. Given that we have enough system resources we can add any number of objects while still meeting its deadlines. In the FreeRTOS solution this does not hold as adding a higher prioritized task might starve the mediator. To ensure satisfactory timing behavior for the mediator, we would have to make the mediator task priority an external variable that is later calculated with respect to the whole system. To keep the jitter in proportion to the period time, we can define the deadline as a fraction of the delay between each call to the method `M`. In the case of TinyTimber this is expressed simply by assigning the deadline to `self->delay*X` where `X` is the fraction of the delay that should be the deadline. To express this in the FreeRTOS version we would have to make the priority a parameter for the initialization method. This in turn would have to be calculated with respect to the system as a whole. This really shows how the timing constraints of the TinyTimber version is not only easier to understand but also encapsulated within the object itself. As the timing constraints are free from external dependencies to other parts of the system, we can create truly modular, composable real-time systems.

Another issue with the FreeRTOS code is the risk of jitter between the system calls, e.g. in our solution between being unblocked and reading the system timer (tick count). Since TinyTimber is based on the scheduling of events and each event is given a timestamp this is a non-issue in the TinyTimber version.

In the TinyTimber solution we could solve the problem without using a regular buffer, instead we utilized the ability to schedule events that take place at a given time in the future to create the desired constant flow of data. In the FreeRTOS version we had to explicitly create a queue that acts like a buffer. Creating and managing this queue is the main reason why the FreeRTOS code is substantially larger (and more complex) than the TinyTimber version. This is also why the number of primitives used, 8 in FreeRTOS and 3 in TinyTimber, is higher. The number of primitives makes the code harder to understand. As an example we can see that it is not always the same primitive that produces the desired timing behavior, e.g. in our FreeRTOS solution we use `xQueueReceive` to wait for the first item and `vTaskDelayUntil` to create the delay between the subsequent items.

D. Mechatronics

The Mechatronics course at Luleå University of Technology is divided into a traditional part with lectures, problem solving, lab exercises, and a project part. Since 2002 the project has been to design and build autonomous model race cars [23]. The 2007 assignment was designed as follows [24]:

> Project assignment 2007
> "The mobile robot should be equipped with an optical sensor which can detect my IR-beacon at 10m distance in sunlight. The sensor signal should be processed by electronic hardware designed and built by the group. The processed signal should then be feed to the ADC in the PIC microcontroller onboard. The whole sensor unit will be mounted on a standard RC-servo which enables the sensor to be scanned in the horisontal plane. The beacon has two vertical rows of IREDs which emit IR radiation at 900nm. The horizontal distance between the rows is 95 mm. The horizontal angle between the IR emitting rows can be measured by the scanning optical sensor. The scanning servo, the speed and front wheel steering angle of the mobil robot is controlled by the microcontroller.
> The race is done with two robots at a time. The two robots start from a starting line on my sign at 10m distance from the beacon. They should drive to the beacon as fast as possible. The goal position is defined as a circle with 1m diameter placed between the start line and the beacon in such a way that the vertical projection of the emitting rows lies on the circle. The winner is the robot which first enters the circle with a wheel and which stops so that at least one wheel is within the circle. The beacon must not be touched by the robot. A final rule is that the starting direction is not against the beacon but away from it!"

Students were divided into 8 groups of 5 students, based on background, so that at least one student per group should have basic programming knowledge. This year, the students were offered the opportunity to utilize the TinyTimber kernel to solve the programming task. Four groups (including the winning team) chose to use TinyTimber while the remaining 4 groups used bare-bones C programming. Figure 2 is an excerpt from the report of the winning team. To achieve the high precision timings needed on this platform (PIC18F4620 [25]) they were forced to use a hardware timer to generate the pulse length. This could easily be done in software on a faster platform. The feedback from the students have already helped identifying a number of possible improvements. The problems encountered were mainly caused by misuse of TinyTimber due to the shortcomings of the documentation and the lack of example code.

Fig. 2. Excerpt from the report of the winning team.

LTU Systemteknik — Mechatronics SME113

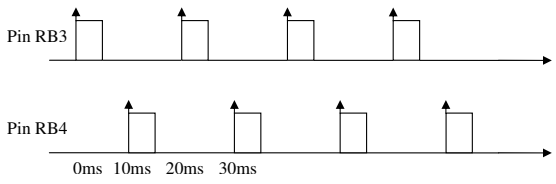
Software

How to control the servos

We use a combination of software and hardware to create the PWM-signal to the servos. The output pin is set high in software and at the same time, timer 3 starts. When timer 3 has reach the value of CCPR2, a CCP2 interrupt is triggered and the output pin is set low.

```
void setPin(Optics *self, int dummy)        //Optic sweep
{
       T3CONbits.TMR3ON = 1;                                //turn on timer3
       PORTBbits.RB3 = 1;                                   //Set pin Hi
       CCPR2 = (self->PW);                                  //set the new pulse width
       ASYNC(MILLISEC(20), SERVO_DL,self,setPin, 0);        //Set Pin again after 20ms


void setPin2(Steering *self, int dummy)     //Steering wheels
{
       T3CONbits.TMR3ON = 1;                                //turn on timer3
       PORTBbits.RB4 = 1;                                   //Set pin Hi
       CCPR2 = (self->PW)+150;              //set the new pulse width (+ software compensation)
       ASYNC(MILLISEC(20), SERVO_DL,self,setPin2, 0);       //Set Pin again after 20ms
}

void interrupt_handler_high(void)
{
       if(PIR2bits.CCP2IF)                  //If this interrupt comes from CCP2
       {
              PORTBbits.RB3 = 0;            //This code is critical, the earlier the better
              PORTBbits.RB4 = 0;

              T3CONbits.TMR3ON = 0;         //turn off timer3
              TMR3L = 0;
              TMR3H = 0;                    //Reset timer3

              PIR2bits.CCP2IF = 0;          //Clear Interrupt Flag
       }
}
```

With this code we can create the PWM needed to control the servos. To avoid interference between the different servos we initialize one setpin method with a delay of 10ms after the other.



Any other method can now easily change the value self->PW to change the pulse width.

Group 1. The Pink Panther 2007-08-16 10

IV. CONCLUSIONS

In this paper we have presented and discussed an alternative approach to comprehensive reactive real-time programming using TinyTimber. Experiences so far are most promising. Results on written examination in an under graduate real-time system's course show clear improvements to the students ability of solving real-time programming problems compared to the previous traditional course. Furthermore, applied to the field of mechatronics, TinyTimber has been successfully applied to implement real-time control systems by students with limited background to the field embedded programming. Future work includes further analysis of course and project results, in order to identify needs for improvements to the kernel and kernel-interface, documentation, example code, commodity libraries, and design patterns.

REFERENCES

[1] "The Java Community Process(SM) Program - JSRs: Java Specification Requests - detail JSR# 1," 2006.
[2] "QNX Realtime operating system (RTOS) software, development tools, and services for embedded applications," http://www.qnx.com/, 2006.
[3] "Home - fsmlabs.com," http://www.fsmlabs.com/, 2006.
[4] "Wind River Home," http://www.windriver.com/, 2006.
[5] "FreeRTOS-A Free RTOS for ARM7,ARM9,Cortex-M3,MSP430,MicroBlaze,AVR,x86,PIC18,H8S,HCS12 and 8051," http://www.freertos.org/, 2006.
[6] "Enea - Homepage," http://www.enea.com/, 2007.
[7] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages, 3rd edition*. Addison Wesley, 2001.
[8] S. Carr, J. Mayo, and C.-K. Shene, "ThreadMentor: A Pedagogical Tool for Multithreaded Programming," in *ACM Journal on Educational Resourses in Computing, Vol. 3, Issue 1*, March 2003.
[9] Q. A. Zhao and J. T. Stasko, "Visualizing the Execution of Threads-based Parallel Programs," Technical Report GIT-GVU-95-01, January 1995.
[10] S. Carr and C. Shene, "A portable class library for teaching multithreaded programming." [Online]. Available: citeseer.ist.psu.edu/carr00portable.html
[11] J. C. Oh and D. Mossé, "Teaching real time oss with doritos," in *SIGCSE '99: The proceedings of the thirtieth SIGCSE technical symposium on Computer science education*. New York, NY, USA: ACM Press, 1999, pp. 68–72.
[12] A. X. M. Team, "Ada 95 Reference Manual," ISO 8652:1995 (E), 1995.
[13] 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*, vol. 91, no. 1, January 2003.
[14] 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.
[15] M. Carlsson, J. Nordlander, and D. Kieburtz, "The semantic layers of Timber," in *Programming Languages and Systems, First Asian Symposium, APLAS 2003, Beijing, China*, ser. Lecture Notes in Computer Science, A. Ohori, Ed., vol. 2895. Springer, November 2003.
[16] J. Nordlander, "Reactive objects and functional programming," PhD thesis, Department of Computer Science, Chalmers University of Technology, Gothenburg, 1999.
[17] 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.
[18] J. Peterson, "The Haskell Home Page," http://haskell.org, 1997.
[19] "Ambient Systems - for low cost, low power, wireless mesh networking solutions," http://www.ambient-systems.net/, 2006.
[20] J. Eriksson and P. Lindgren, "A comprehensive approach to design of embedded real-time software for controlling mechanical systems," in *Asia Pacific Automotive Engineering Conference, APAC-14*, 2007.
[21] V. Leijon, P. Lindgren, and J. Eriksson, "FIFO WiDOM: Timely Control Over Wireless Links," in *IEEE Multi-conference on Systems and Control*, 2007.
[22] "Real-Time Systems," http://www.sm.luth.se/csee/courses/smd/138/, 2007.
[23] K. Hyyppä, "Competition - An Efficient Method to get Students Committed," in *ICALT04*, 2004.
[24] "Mechatronics," sme113, Luleå University of Technology(http://www.ltu.se), course homepage only accessible to students via fronter., 2007.
[25] "Microchip," http://www.microchip.com, 2007.

TINYTIMBER INTERFACE

Summary of the TinyTimber interface

- **#include** "TinyTimber.h"
  Provides access to the TinyTimber primitives.
- **typedef struct** {...} Object;
  Base class of reactive objects. Every reactive object in a TinyTimber system must be of a class that inherits this class.
- **#define** initObject() {...}
  Initialization macro for class Object.
- **typedef** ... Time;
  Abstract type of time values.
- Time SEC(**int** seconds);
- Time MILLISEC(**int** milliseconds);
- Time MICROSEC(**int** microseconds);
  Constructs a Time value from an argument given in seconds / milliseconds / microseconds.
- **int** SYNC(T *obj, **int** (*m)(T*,A), A a);
  Synchronously invokes method m on object obj with argument a. Type T must be a struct type that inherits from Object, while A can be any int-sized type. If completion of the call would result in deadlock, -1 is returned; otherwise the result is the value returned by m.
- **void** ASYNC(Time b, Time d, T *obj, **int** (*m)(T*,A), A a);
  Asynchronously invokes method m on object obj with argument a, baseline offset b, and relative deadline d. Type T must be a struct type that inherits from Object, while A can be any int-sized type. A d value of zero is interpreted as the infinity. If both b and d are zero, the time constraints of the caller are inherited. Otherwise, a new time window for the asynchronous call is computed as:
  new baseline = max(current baseline + b, current time)
  new deadline = new baseline + d.
- Time BASELINE(**void**);
  Returns the baseline of the currently executing method call.
- INTERRUPT(vector, stmt);
  Declares an interrupt handler for vector whose body is stmt.
- STARTUP(stmt);
  Declares a system startup handler whose body is stmt.

Paper G

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

Authors:
Per Lindgren, Johan Eriksson, Simon Aittamaa, Johan Norlander

Accepted for publication in:
DATE - Design, Automation and Test in Europe, 2008

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.

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.