Programming with the TinyTimber kernel

Johan Nordlander

September 13, 2012

1   Introduction

This document describes the C programming interface to *TinyTimber*, a very small and lightweight run-time kernel for event-driven embedded systems.

Prerequisites assumed in this document are familiarity with standard sequential programming in the C language as well as basic knowledge of microprocessor fundamentals, especially concerning interrupt handling and device I/O on the assembly level. Experience with the concepts of an object-oriented language such as Java will also be helpful, but is not an absolute necessity.

2   Design principles

The fundamental idea that underlies the design of TinyTimber is the notion of a *reactive object*. A reactive object is a component that reacts to incoming *events* by updating its internal state and/or emitting outgoing events of its own. Between such activations a reactive object is *idle*; i.e., it simply maintains its state while waiting for future activations.

According to this view, a hardware device such as a serial port controller is a typical reactive object. The device itself is literally a "black box" object that does nothing unless stimulated by external events. In the case of a serial port these events are of two kinds: either a signal change on the incoming communication line, or a read or write command received on the connected data bus. The emitted events are similarly divided; either a generated signal change on the outgoing communication line, or an interrupt signal issued towards some connected microprocessor. The clock pulse driving the shift registers of the port may also be included among the external event sources, if desired. At all times, the current state of a serial port object is the contents of its internal registers. Figure 1 depicts this view.[1]

---

[1]Although the graphical notation used within this document is entirely informal, the two
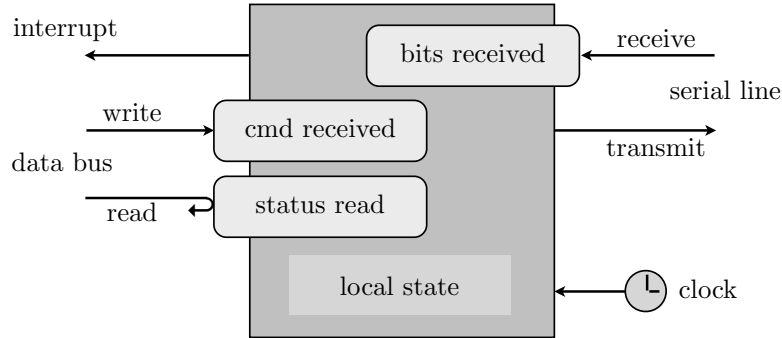
1

Figure 1: A reactive object in the form of a serial port

The microprocessor itself may also be considered a reactive object. It reacts to incoming interrupt events by updating its state (memory and registers) or generating commands (read or write) on the data bus. Here as well a clock pulse may be viewed as an additional source of incoming events. Perhaps less obvious is the fact that a microprocessor which is done emitting all its responses to previous events is also in effect doing "nothing" until a new interrupt occurs. Programmers tend to think of this "nothing" as the repeated execution of some dummy instruction, but it should be noted that most microprocessor architectures provide an alternative in the form of an instruction that literally halts program flow until an interrupt occurs. In any case, the division of a microprocessor's time into active and inactive phases is an important aspect of its behavior. Figure 2 shows the system obtained when a microprocessor object is connected to a serial port and a keyboard object via some common data bus.

Even a compound system can be thought of as a reactive object in its own right. For example, a user of the microprocessor system in Figure 2 will probably not distinguish the individual components from each other, but rather view the system as a single object capable of emitting serial port packets while reacting to incoming port and keyboard events. Conversely, a single reactive object might reveal a more refined structure of interconnected components when looked at from the inside, and this pattern may very well repeat itself at different levels of magnification. The core idea of TinyTimber is to let such a hierarchy of reactive objects extend into

kinds of arrows used are intended to distinguish between one-way signalling (the straight arrow) and request-response communication (the fishhook). Blocks stand for stateful objects, and the operatons that can be performed on these appear as rounded rectangles along the border. The clock symbol is just a generator of periodic events.
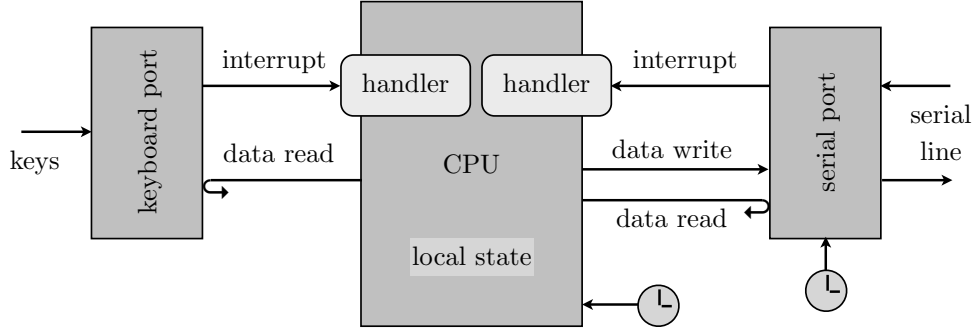
Figure 2: A microprocessor object connected to a keyboard and a serial port

the *programmable* internals of a microprocessor object as well.

So, the microprocessor is a hardware reactive object whose behavior and internal structure is defined by software. The links across this hardware/software boundary are the interrupt handlers, which constitute the externally visible *methods* of the microprocessor object. The persistent state of the microprocessor are all the global variables, which we may choose to partition into an object structure that reflects a conceptual hardware decomposition of the programmed application. Figure 3 illustrates what the software-defined internal structure of a microprocessor object may look like.[2]

An essential aspect of real world objects is that they naturally evolve in parallel with each other, so a key task for the TinyTimber kernel is to simulate concurrent execution of multiple software objects on single processor hardware. Moreover, TinyTimber will also guarantee that the methods of a particular object execute in a strictly sequential fashion, thus avoiding the need for any manual state protection mechanisms (see Figure 4). To make this scheme work, the programmer has to ensure that every method call that crosses an object boundary is handled by the TinyTimber kernel. Apart from that requirement, though, programming with TinyTimber is as simple as partitioning the program state into an appropriate object structure, and defining the behavior of each object in terms of method code.

Details of the TinyTimber method call primitives will be explained in subsequent sections. First, however, some basic techniques for expressing object-

---

[2]This figure shows a clock symbol placed over one of the one-way communication arrows, as an indicator of the TinyTimber way of controlling timing behavior. More on this topic in Section 6.
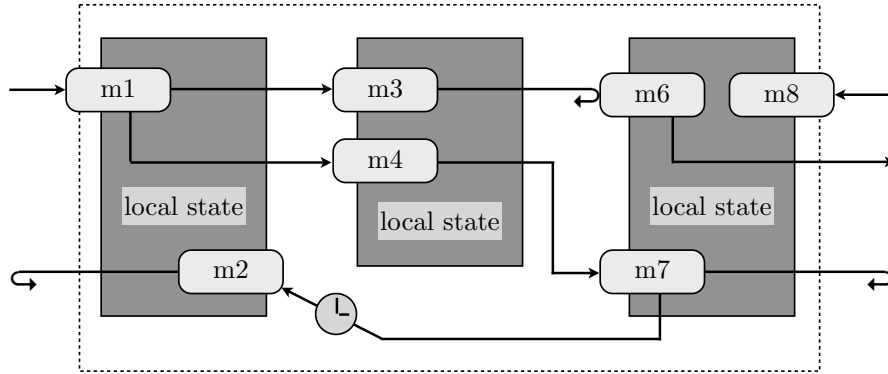
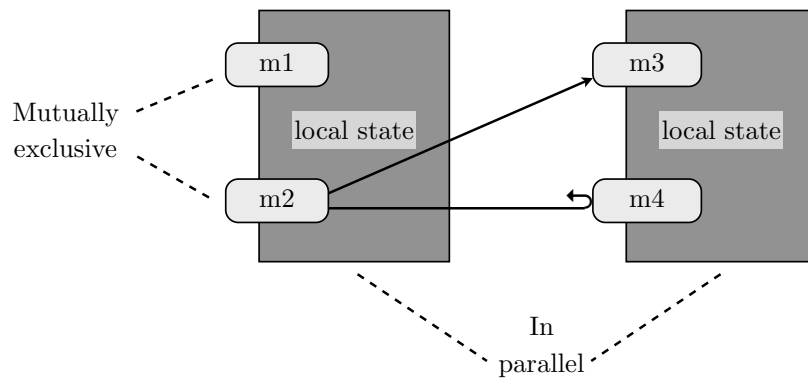Figure 3: Software-defined internals of a microprocessor object

Figure 4: Concurrency and mutual exclusion among reactive objects

oriented concepts in C must be discussed.

3   Basic object-orientation in C

Object-oriented programming means many different things to different people, so
let us start with some terminology.

By an *object* we mean a collection of variables at some unique place in mem-
ory, together with a set of functions that have the exclusive right to access these
variables. The functions are commonly called *methods*, and the variables are often
termed *state* or *instance* variables. Calling, or *invoking* a method of an object is
commonly thought of as *sending a message* to the object. A *class* is a template
for creating objects with a similar variable layout and a similar set of methods.
Inheritance is a mechanism for defining classes on basis of already existing classes,
primarily by adding state variables and methods.

To encode these object-oriented concepts in the C language we will use a com-
bination of a `typedef` and a `struct` for defining the variable structure of classes,
and the standard technique of "self-application" for invoking methods. Here is an
example of a class `Counter` that specifies two instance variables:

```
typedef struct {
    int value;
    int enabled;
} Counter;
```

Creating an object from a class template just means initializing a fresh object
variable of the right type with the proper initial values. We will consistently use a
preprocessor macro $\text{init}X$ to express initialization of an object of class X, which
for the `Counter` class might look like

```
#define initCounter(en)  { 0, en }
```

A fresh `Counter` object is then created as follows:

```
Counter cnt = initCounter(1);
```

Self-application means invoking a method by applying it to the address of the
receiving object – that is, to the *self* of the receiver. A method belonging to a class
X is thus supposed to take a pointer to an X object as an extra first parameter.
The following functions both qualify as `Counter` methods:

```
int inc( Counter *self, int arg ) {
    if (self->enabled)
        self->value = self->value + arg;
```

```
        return self->value;
    }

    int enable( Counter *self, int arg ) {
        self->enable = arg;
        return 0;
    }
```

To call method `inc` on object `cnt` with argument `1`, one essentially just writes

```
    inc( &cnt, 1 );
```

However, as we will see in Section 4, method calls directed towards an object different from the current self will be expressed in a slightly different manner.

Moreover, a method could in principle return any type and take any number and type of arguments after the mandatory first object pointer. For technical reasons, though, TinyTimber restricts the method calls it handles to carry just one integer parameter and return an integer-valued result. This restriction may be loosened somewhat by means of type casts, provided that the actual values can be represented within the space of an integer.

A class C may inherit a class B – i.e., be an extension of class B – by including a B object as the first component of its variable layout. For example, a class `ResetCounter` can be defined as an extension of `Counter` as follows:

```
    typedef struct {
        Counter super;
        int nResets;
    } ResetCounter;

    #define initResetCounter(en)   { initCounter(en), 0 };
```

A `ResetCounter` object created by

```
    ResetCounter rcnt = initResetCounter(1);
```

may now be treated as a `Counter` object by means of a simple type cast, because the address of an object and the address of its first component will always be identical. Hence methods `inc` and `enable` above may be used on `ResetCounter` objects as well:

```
    inc( (Counter*)&rcnt, 4 );
    enable( (Counter*)&rcnt, 0 );
```

The `ResetCounter` class may in addition provide new methods that do not work on `Counter` objects. Here is an example:

```
int reset( ResetCounter *self, int arg ) {
    self->super.value = 0;
    self->nResets++;
    return 0;
}
```

Notice how the instance variables of the *superclass* object are accessible via the field `super`. It is also important to observe that the `super` field must denote a full superclass object, not merely a pointer to such an object; otherwise the type casting trick shown above will not work.

More advanced object-oriented mechanisms such as dynamic binding and method override are certainly possible to encode along this line as well, but this is as far as we need to go in order to utilize the TinyTimber kernel. In fact, even the use of inheritance as described above is probably an overkill in many applications.

However, the TinyTimber kernel places some basic requirements on the layout of every object it manages, and these requirements are most straightforwardly expressed terms of inheritance. Concretely this means that every class in a Tiny-Timber system must inherit the predefined class `Object`, either directly or via some other class that inherits `Object`. To qualify as a correct TinyTimber class, our `Counter` class should thus actually have been written

```
typedef struct {
    Object super;
    int value;
    int enabled;
} Counter;

#define initCounter(en)  { initObject(), 0, en }
```

The definition of `ResetCounter` is ok as it is, because it inherits `Object` via `Counter`. From now on we will silently assume that all classes follow the inheritance requirement one way or another.

The most important aspect of our encoding is that it naturally leads to a software structure where the program state is partitioned into well-defined units of concurrency. Moreover, provided that

1. only the methods of an object are allowed to access its state variables, and

2. method calls not directed to the current self are handled by the kernel,

the TinyTimber kernel is actually able to guarantee that the state integrity of all objects is automatically preserved, even in the presence of overlapping reactions and multiple concurrent events.

This property heavily depends on the way method calls are performed, so let us now take a look at the TinyTimber primitives for invoking methods in a concurrent setting.

4   Concurrent method calls

Recalling Figure 4, every TinyTimber object acts a potential host of concurrent activity. On the other hand, method calls directed to a particular object must execute in a strictly serial fashion, such that each object is executing *at most* one of its methods at a time. These requirements affect the semantics of method calls in two different ways:

1. Each time a method call is made, the caller must actually check that the receiving object is not already executing some method. Should this be the case, the caller must wait.

2. Alternatively, in cases where the caller is not really interested in the result returned by a method, the receiving object could be left to execute the designated method whenever it is ready. This would leave the caller free to continue immediately once the method call is posted, possibly executing in parallel with the called method.

Unfortunately, neither alternative will happen automatically if methods are invoked in the standard self-applicative fashion. For example, suppose some method of an object `x` wishes to call method `reset` of object `rcnt` defined in Section 3. Simply writing

```
reset( &rcnt, 0 );
```

will force execution of `reset` irrespective of whether `rcnt` is already executing another method or not (with possible corruption of the state variables of `rcnt` as a result). Moreover, this simplistic mechanism will not provide any option for letting the current method of `x` continue in parallel with method `reset`.

In the concurrent programming literature, the two invocation alternatives described as bullets 1 and 2 above go under the names *synchronous* and *asynchronous* message passing, respectively. TinyTimber supports them both in the form of two primitive operations: `SYNC` and `ASYNC`.

Primitive `SYNC` takes an object pointer, a method name and an integer argument, and performs a synchronous method call as described under bullet 1. For example, to execute the method call attempted above in a synchronous fashion, one writes

```
SYNC( &rcnt, reset, 0 );
```

Should the method result be of interest, it can be captured as the result of the SYNC operation:

```
some_variable = SYNC( &rcnt, inc, 1 );
```

The primitive ASYNC also takes an object pointer, a method name and an integer argument, but performs an asynchronous method invocation along the lines of bullet 2 instead. An asynchronous variant of the previous example thus looks as follows:

```
ASYNC( &rcnt, reset, 0 );
```

The result of an ASYNC call is always a *message tag*, independently of whether the actual method invoked returns a meaningful value or not (recall that the ASYNC primitive does not wait for any result to be produced). The returned tag is mostly ignored, but can also be used to prematurely abort a pending asynchronous message (see the appendix for details).

Making an asynchronous call to the currently executing object (i.e., to the current self) is perfectly reasonable – that call will be handled *after* the active method has terminated. This would be like sending a letter to one's own mail address, to be picked up at a later time. The usefulness of such an arrangement will perhaps be more evident after Section 6, where some variants of the ASYNC primitive are discussed.

On the other hand, a synchronous call to the current self is an unconditional error that will lead to *deadlock*, because the receiver cannot possibly be ready to accept such a call at any time. This situation would be like calling one's own telephone number and waiting for someone to pick up the phone – clearly not a good strategy for making progress! If the intention is to simply call a method of the current self as if it were an ordinary subroutine, the standard self-application mechanism should be used instead. In fact, this is the *only* scenario where self-application should be used in a TinyTimber system; all other method calls either cross an object boundary and/or need asynchronous behavior.

In general, deadlock may result whenever there is a cycle of objects that call each other synchronously. Should deadlock occur, TinyTimber will abort the closing SYNC call, and return the error code -1. Programmers must therefore take special action to disambiguate SYNC results if -1 can also be expected as a proper return value. The easiest way is to avoid deadlocks altogether, by breaking every synchronous call cycle with an ASYNC call at some point.

A word of caution: Both SYNC and ASYNC make heavy use of type casts to achieve a convenient method call syntax. Unfortunately, the C language is not potent enough to detect some misuses that may result; for example, attempting to call method reset on object cnt (which does not support that method). As

9

is always the case when type casts are involved, the programmer must take full responsibility for the correctness of the resulting program. But this should be no news to a C programmer!

5   The software-hardware boundary

Figure 3 shows the logical structure of a hypothetical microprocessor and its programmed software internals. Three reactive objects can be identified, each one supporting its own methods and some local state. These objects are conveniently described and implemented according to the techniques discussed in Sections 3 and 4.

However, Figure 3 also indicates that methods m1 and m8 are callable from the world outside the microprocessor, and that methods m2, m6 and m7 are able to invoke methods of this outside world in turn. It is now time to explain workings of this software-hardware boundary in more detail.

Concretely, incoming method calls from the hardware environment correspond to interrupt signals received by the microprocessor. Methods m1 and m8 thus have the status of interrupt handlers, that are invoked each time the microprocessor detects a signal on its corresponding interrupt input pins. Apart from this special link to the outside world, interrupt handlers are ordinary methods accepting the same type of parameters as methods invoked with `SYNC` and `ASYNC`. To install method `meth` on object `obj` as an interrupt handler for interrupt source `IRQ_X`, one writes

```
INSTALL(&obj, meth, IRQ_X);
```

This call, which preferably should be performed during system startup, causes `meth` to be subsequently invoked with `&obj` and `IRQ_X` as arguments whenever the interrupt identified by `IRQ_X` occurs. The symbol `IRQ_X` is here used as a place-holder only; the exact set of available interrupt sources is captured in a platform-dependent enumeration type `Vector` defined in the TinyTimber interface.

A few technical points regarding interrupts are worth noticing. Interrupt handlers are effectively scheduled by the processor hardware, which on most platforms means that they are executed non-preemptively (i.e., with further interrupts disabled). To avoid conflicts between the hardware and software schedulers, Tiny-Timber ensures that *all* methods of an object that has interrupt handlers installed execute with interrupts disabled. This poses no restrictions on the way such methods can be invoked (`SYNC` or `ASYNC`), nor on what kind of calls that can be made from within such methods. However, synchronous calls within an interrupt handler will only succeed if the receiving object is inactive at the time of the interrupt.

`SYNC` should therefore not be used by interrupt-handling methods unless it is acceptable that the failure value (`-1`) may occasionally be returned.

System reset is a particular event that most microprocessor hardware treats as yet another interrupt source. To a C programmer, though, a "system reset" handler is already available in the form of the `main` function, so the TinyTimber interface will typically not include a specific system reset identifier in its `Vector` definition.[3] The `main` function nevertheless has a special responsibility in a TinyTimber system, in that it must hand over control to the TinyTimber scheduler; otherwise the system will just terminate prematurely without handling any events. This is achieved by invoking the non-terminating primitive `TINYTIMBER` as the last `main` statement. Here is an example:

```
int main() {
    INSTALL(&obj1, meth1, IRQ_1);
    INSTALL(&obj2, meth2, IRQ_2);
    return TINYTIMBER(&obj3, meth3, val);
}
```

This system will install two interrupt handlers before invoking TinyTimber proper, thereby firing off the whole event-handling mechanism. The arguments given to `TINYTIMBER` identify a method call that will act as the startup event — the TinyTimber kernel will schedule this call as its first event-handling operation. Should no specific startup event be desired, `NULL` can be given in place of all three arguments.

Returning to Figure 3, the concrete representation of outgoing method calls directed towards the hardware environment are *read* or *write* commands issued by the processor to devices on the external data bus – i.e., events generated in software to which hardware objects are supposed to react. Such I/O operations can be conveniently expressed in the C language as either pointer operations (memory-mapped architectures) or inline assembly code (separate I/O bus), and need no involvement by the TinyTimber kernel. Still, I/O operations form an important conceptual link in the reactive object model on which TinyTimber is built, and they can preferably be understood as yet another form of method call in a reactive system.

Here is an example of what a set of outgoing method calls to hardware objects might look like on a memory-mapped architecture.

```
char *port = (char*) 0x1234;   // "self" of the hardware object
...
```

---

[3]TinyTimber will reserve a few other interrupt sources for its own internal use as well, which will also be absent from the definition of `Vector`.
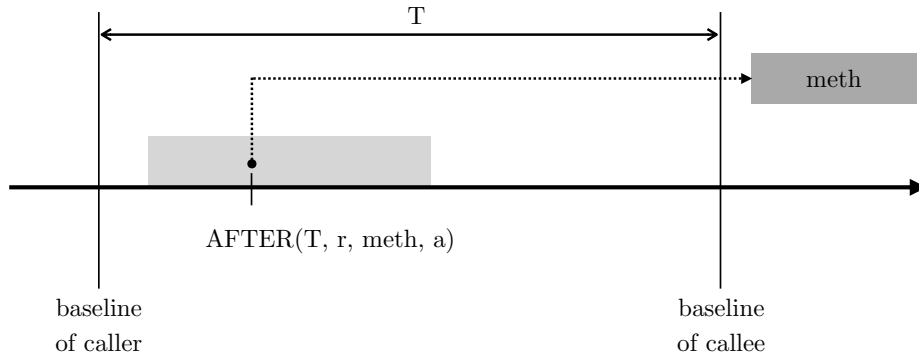
Figure 5: Baseline adjustment using `AFTER` with an offset T

```
unsigned char = *port;           // A "read" method call
...
*port = expr;                    // A "write" method call
```

For concrete information on the hardware objects available in a particular context, platform specific C documentation should be consulted.

6 Managing time

Implied in the presentation this far has been the assumption that all methods terminate after a relatively brief outburst of activity. This might contradict many people's perception that concurrent and event-driven software inevitably must involve infinite loops. In a TinyTimber system this is far from the truth, though. There are no means to "block" for an event in TinyTimber, and when there is no method activity, a reactive object simply rests. That characteristic naturally applies to microprocessor hardware as well, and thanks to the TinyTimber scheduler, it also holds for the whole compound object that includes the microprocessor and its programmed internals.

Still, what is lacking in the TinyTimber picture painted so far is the ability to express *periodic* activity. Or more generally put: to prescribe that a certain activity must be triggered *at a certain point in time.*

The TinyTimber primitive for managing the passage of time is called `AFTER`. Unlike time primitives in many other concurrent systems, `AFTER` does not stop the program flow; instead it just sets up an event at a future point in time that some chosen object will react to. The following code issues an asynchronous call to method `meth` of object `obj`, to be be delivered after X seconds.

```
AFTER( SEC(X), &obj, meth, 123 );
```

Function `SEC` is a preprocessor macro that converts its argument to the platform dependent units of time used by `AFTER`. Other useful macros with a similar purpose are `MSEC` and `USEC`.

One important TinyTimber chracteristic is that the time parameter in an `AFTER` call is not mesured from the time of the call, but from the current *baseline*, which is a time-stamp TinyTimber sets for each interrupt and then subsequently maintains for each message. The baseline of a message can be seen as a lower timebound on its execution, and what `AFTER` actually does is that it lets the programmer adjust the baseline of a called method with an offset relative to the current one (see Figure 5). An `AFTER` call with a baseline offset of `0` thus means that the called method runs with the same baseline as the caller. In fact, writing

```
ASYNC( &obj, meth, 123 );
```

is just a shorthand for the equivalent call

```
AFTER( 0, &obj, meth, 123 );
```

Counting time offsets from the stable reference point of a baseline makes the actual time an `AFTER` call is made irrelevant. For example, the following methods are equivalent, both performing some immediate work and triggerering `more_work` to be run `T` seconds after the current baseline.

```
int work1( MyObject *self, int arg ) {
    // do immediate work
    AFTER( SEC(T), &obj, more_work, 0 );
}

int work2( MyObject *self, int arg ) {
    AFTER( SEC(T), &obj, more_work, 0 );
    // do immediate work
}
```

Below follows a method that will react to an initial invocation by repeatedly calling itself every `T` milliseconds.

```
int tick( MyObject *self, int arg ) {
    // do something useful
    AFTER( MSEC(T), self, tick, arg );
}
```
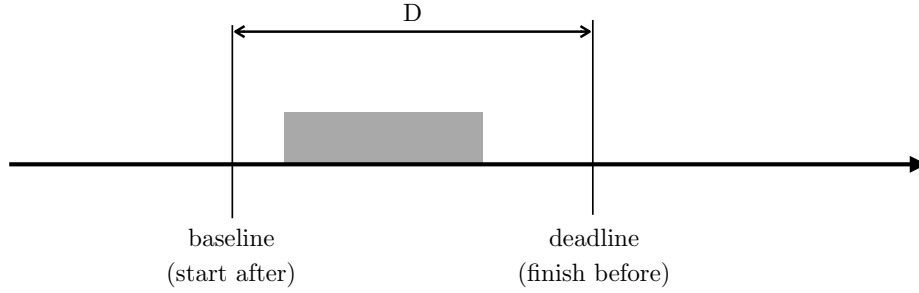
Figure 6: Window of legal method execution (relative deadline is D)

This example succinctly captures the way periodic computations are expressed in TinyTimber. That is, a recursive asynchronous method call with a time offset replaces the loops and blocking operations found in many other systems. An immediate benefit of the TinyTimber formulation is that every object utilizing the pattern is free to handle any other events in the space between periodic activations. A TinyTimber object simply treats the ticking of time as any other event – i.e., as some unknown source of an ordinary method call. Moreover, the new baseline computed by an AFTER call is indifferent to the actual speed at which methods execute. Periodic computations in particular are thus free from accumulating drift.

For applications where hard real-time performance is required, TinyTimber also provides an option for specifying the *deadlines* of asynchronous messages. A deadline marks a point in time when a method must be done, and together with the baseline of a method call, it defines a time window of legal method execution (see Figure 6).

To perform an asynchronous call with an explicit deadline D, one writes

```
BEFORE( D, &obj, meth, 123 );
```

C.f. Figure 7. To do the same with an additional baseline offset T, a combination of the AFTER and BEFORE primitives is used (Figure 8):

```
SEND( T, D, &obj, meth, 123 );
```

In both cases, the specified deadline parameter is a time-span measured relative the baseline of the generated message. As a special case, a relative deadline value of 0 is interpreted as infinity; i.e., specifying an infinitely wide window of legal
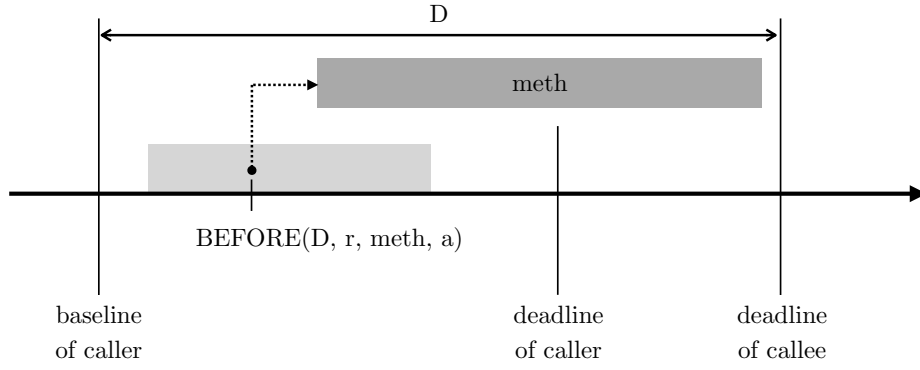
14

Figure 7: Deadline adjustment using `BEFORE` with an offset D
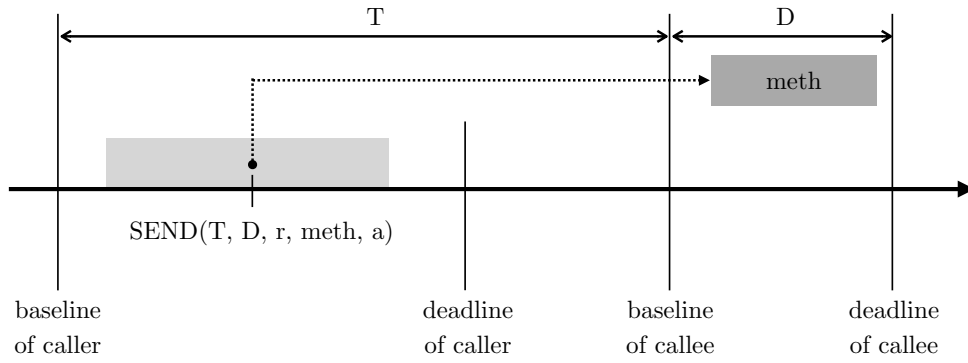

Figure 8: Combined baseline and deadline adjustment using `SEND`

execution. With this fact in mind, all variants of the asynchronous method call can actually be defined as variations of SEND:

```
BEFORE( D, ...  )  =  SEND( 0, D, ...  )
AFTER( T, ...  )   =  SEND( T, 0, ...  )
ASYNC( ...  )      =  SEND( 0, 0, ...  )
```

The baseline of a method call may preferably be used as a time-stamp for the event that initiated the call, as it is insensitive to any scheduling decisions made by the kernel. Moreover, TinyTimber offers a built-in class Timer, with methods T_RESET and T_SAMPLE, that allows the difference between two such time-stamps to be obtained. The following example shows the core of a sonar measurement application, where time-stamps and the facilities of class Timer are put to good use.

```
typedef struct {
    Object super;
     Timer timer;
} Sonar;

#define initSonar()  { initObject(), initTimer() }

#define GENERATOR_PORT = (*(char *) 0x1234)

Sonar sonar = initSonar();

int stop( Sonar *self, int arg ) {
   GENERATOR_PORT = SONAR_OFF;
}

int echo( Sonar *self, int arg ) {
   Time diff = T_SAMPLE(&self->timer);
   if (diff < MSEC(LIMIT)) ...;
}

int tick( Sonar *self, int arg ) {
   GENERATOR_PORT = SONAR_ON;
   T_RESET(&self->timer);
   AFTER( MSEC(10), self, stop, 0 );
   AFTER( MSEC(500), self, tick, 0 );
}
```

16

```
int main() {
    INSTALL( &sonar, echo, IRQ_ECHO_DETECT );
    return TINYTIMBER( &sonar, tick, 0 );
}
```

TinyTimber uses both deadlines and baselines as input to its scheduling algorithm, although it is actually only the deadlines that pose any real challenge to the scheduler. However, missed deadlines are not trapped at run-time; if such behavior is desired it must be programmed by means of a separate watchdog task.

Judging whether a TinyTimber program will meet all its deadlines at run-time is an interesting problem, that can only be solved using a separate schedulability analysis and known worst-case execution times for all methods. That topic, however, is beyond the scope of the present text.

7 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 (i.e., can be cast to) this class.

- `#define initObject() {...}`

  Initialization macro for class `Object`.

- `int SYNC( T *obj, int (*meth)(T*,A), A arg );`

  Synchronously invokes method `meth` on object `obj` with argument `arg`. 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`.

- `typedef struct ...  *Msg;`

  Abstract type of asynchronous message tags.

- `#define NULL 0`

  Unit value of type `Msg` and other pointer types.

- `Msg ASYNC( T *obj, int (*meth)(T*,A), A arg );`

  Asynchronously invokes method `meth` on object `obj` with argument `arg`. Identical to `SEND(0, 0, obj, meth, arg)`.

- `typedef signed long Time;`

  Type of time values (with platform-dependent resolution).

- `Time SEC( int seconds );`

  `Time MSEC( int milliseconds );`

  `Time USEC( int microseconds );`

  Constructs a `Time` value from an argument given in seconds / milliseconds / microseconds.

- `int SEC_OF( Time );`

  `int MSEC_OF( Time );`

  `long USEC_OF( Time );`

  Extracts whole seconds and millisecond or microsecond fractions of a `Time` value.

- `Msg AFTER( Time bl, T *obj, int (*meth)(T*,A), A arg );`

  Asynchronously invokes method `meth` on object `obj` with argument `arg` and baseline offset `bl`. Identical to `SEND(b, 0, obj, meth, arg)`.

- `Msg BEFORE( Time dl, T *obj, int (*meth)(T*,A), A arg );`

  Asynchronously invokes method `meth` on object `obj` with argument `arg` and relative deadline `dl`. Identical to `SEND(0, dl, obj, meth, arg)`.

- `Msg SEND( Time bl, Time dl, T *obj, int (*meth)(T*,A), A arg );`

  Asynchronously invokes method `meth` on object `obj` with argument `arg`, baseline offset `bl`, and relative deadline `dl`. Type `T` must be a struct type that inherits from `Object`, while `A` can be any `int`-sized type. Returns a tag that can be used to identify the message in a future call to `ABORT`.

  Offsets `bl` and `dl` allow a new execution window for the asynchronous call to be defined on basis of the current one:

$$\begin{array}{rcll} \textit{new baseline} & = & \textit{current baseline} + \texttt{bl} & \\ \textit{new deadline} & = & \textit{infinity}, & \text{if } \texttt{dl} = 0 \\ & = & \textit{new baseline} + \texttt{dl}, & \text{otherwise} \end{array}$$

- `void ABORT( Msg m );`

  Prematurely aborts pending asynchronous message `m`. Does nothing if `m` has already begun executing.

- `enum Vector = {...};`

  Platform-dependent list of interrupt source identifiers.

- `INSTALL( T *obj, int (*meth)(T*, enum Vector), enum Vector v );`

  Install method `meth` on object `obj` as an interrupt-handler for interrupt source `i`. Type `T` must be a struct type that inherits from `Object`. When an interrupt on `i` occurs, `meth` will be invoked on `obj` with `i` as its argument.

- `TINYTIMBER( T *obj, int (*meth)(T*,A), A arg );`

  Start up the TinyTimber system by invoking method `meth` on `obj` with argument `arg`; then handle all subsequent interrupts and timed events as they occur. Type `T` must be a struct type that inherits from `Object`, while `A` can be any `int`-sized type. This function never returns.

- `typedef struct {...} Timer;`

  Abstract type of timer objects.

- `#define initTimer() {...}`

  Initialization macro for class `Timer`.

- `void T_RESET( Timer *t );`

  Reset timer `t` to the value of of current baseline.

- `Time T_SAMPLE( Timer *t );`

  Return difference between current baseline and timer `t`.

- `Time CURRENT_OFFSET(void);`

  Return current time measured from current baseline.