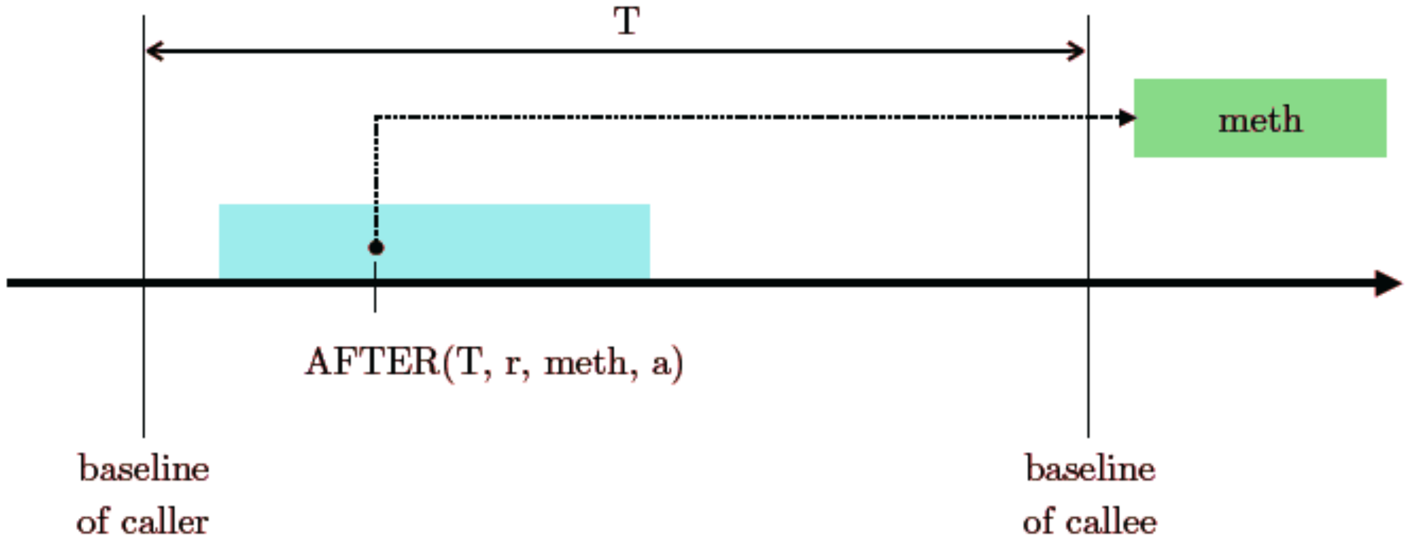
Real-Time Systems

Exercise #3

Victor Wallsten

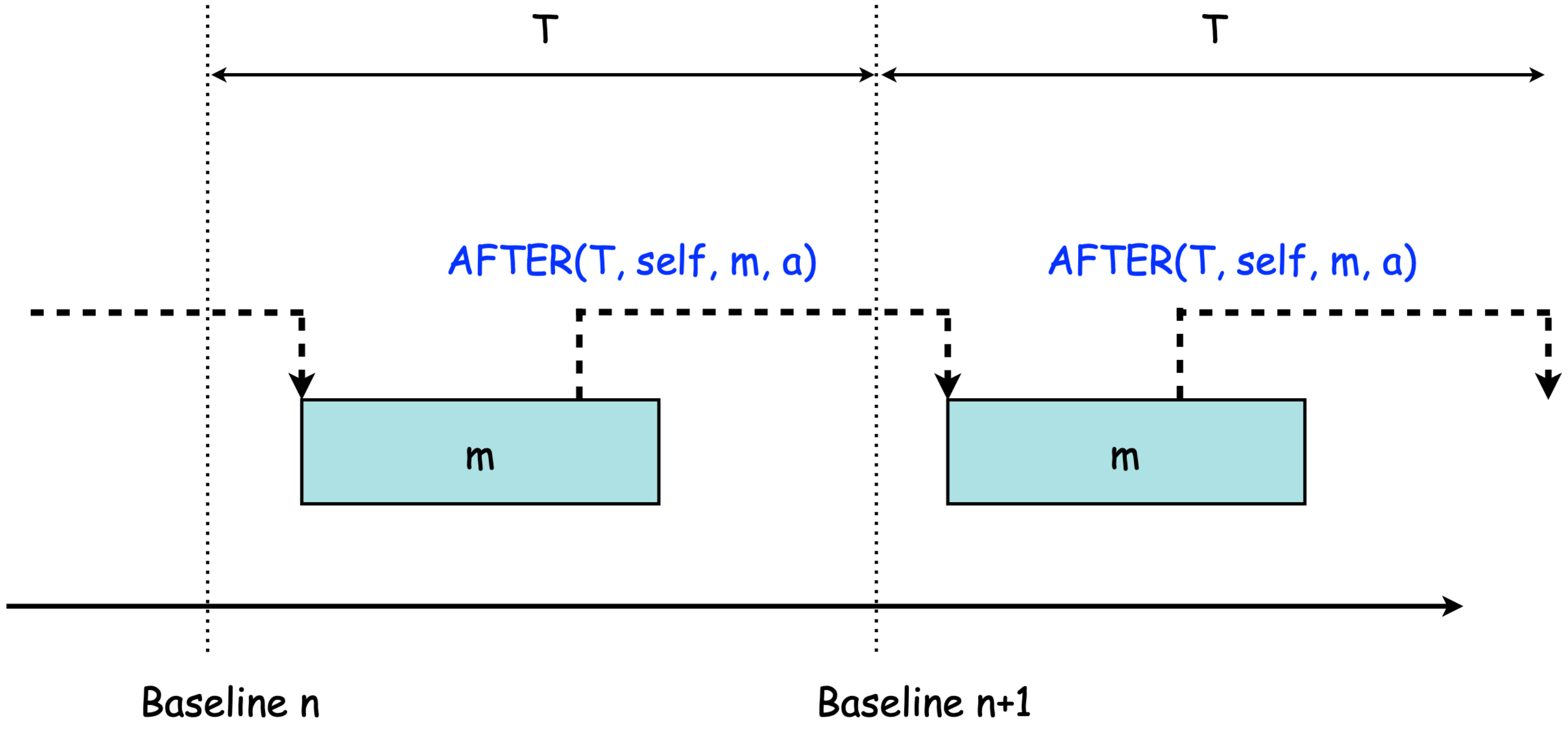
Department of Computer Science and Engineering
Chalmers University of Technology

Examples with AFTER()



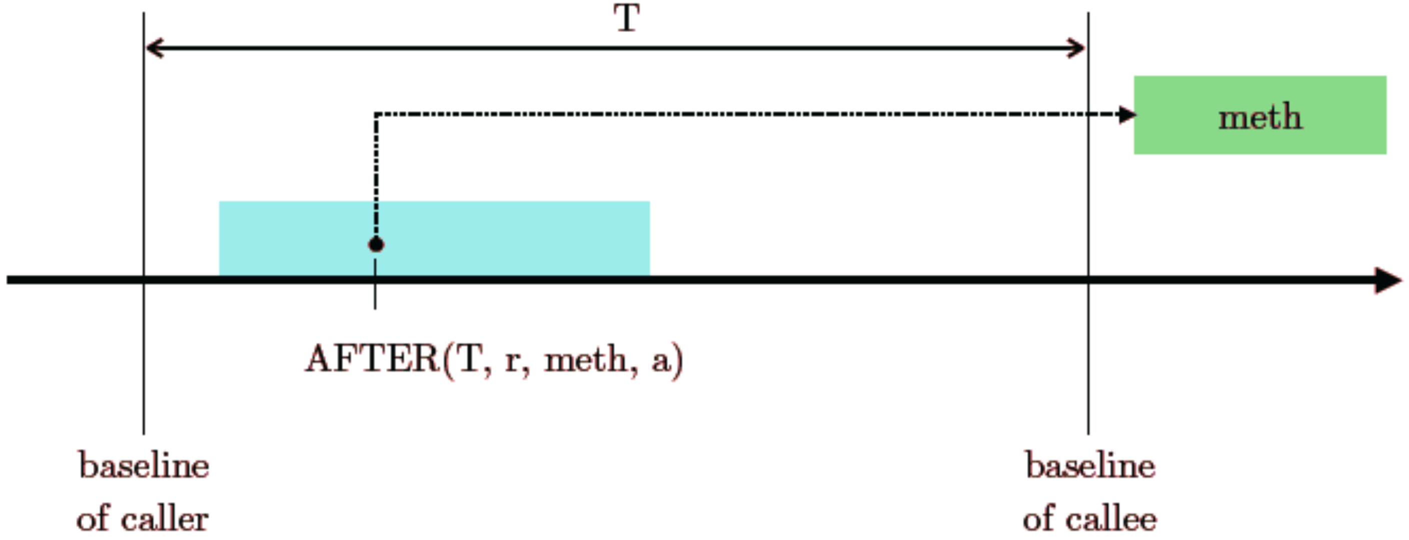
```
// Call method ‘do_some_work’ in object ‘task1’
// after 2 ms
AFTER(MSEC(2),&task1,do_some_work,0);

// Call method ‘do_more_work’ in object ‘task2’
// after 500 μs
AFTER(USEC(500),&task2,do_more_work,1);
```

Periodicity with AFTER()



```
// Call yourself after 15 ms
int do_some_work(MyObject *self, int n) {
        ... // do some work
        AFTER(MSEC(15),self,do_some_work,0);
}
```

Some more about AFTER()



An `AFTER()` call with a baseline of 0 means that the called method runs with the same baseline as the caller.

```
ASYNC(&obj,meth,n)  == AFTER(0,&obj,meth,n);
```

Some more about `AFTER()`

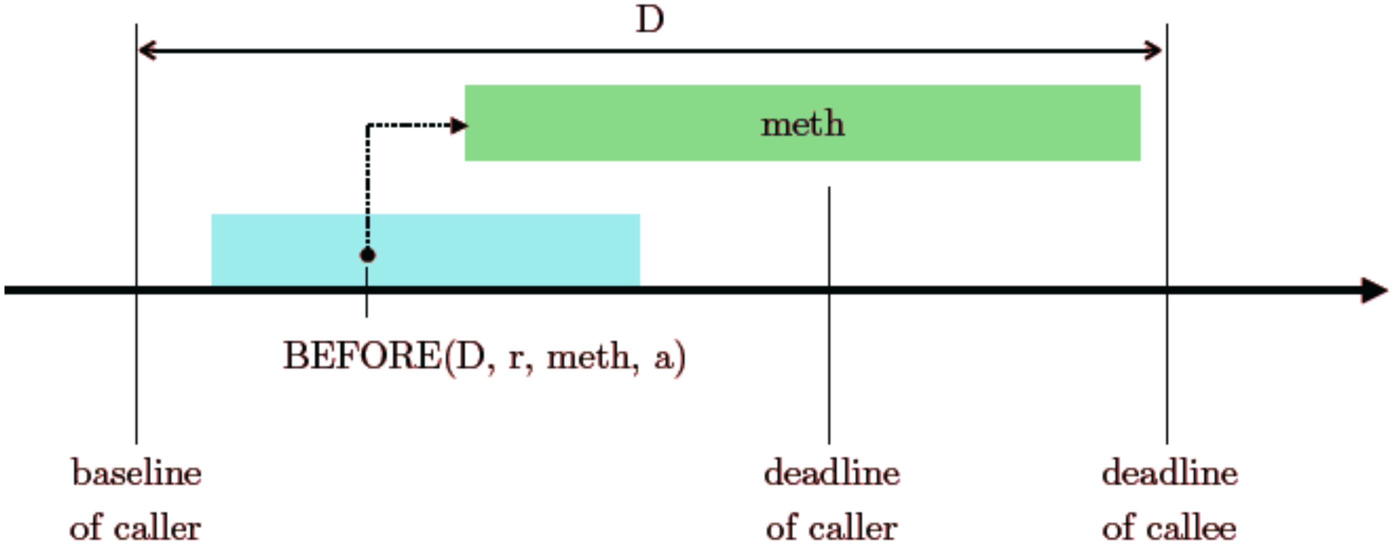
Using the baseline to derive time offsets makes the actual time the `AFTER()` call is made less critical!

```
int do_some_work(MyObject *self, int n) {
    ... // do some work
    AFTER(SEC(T),&obj,do_more_work,0);
}
```

has the same behavior as

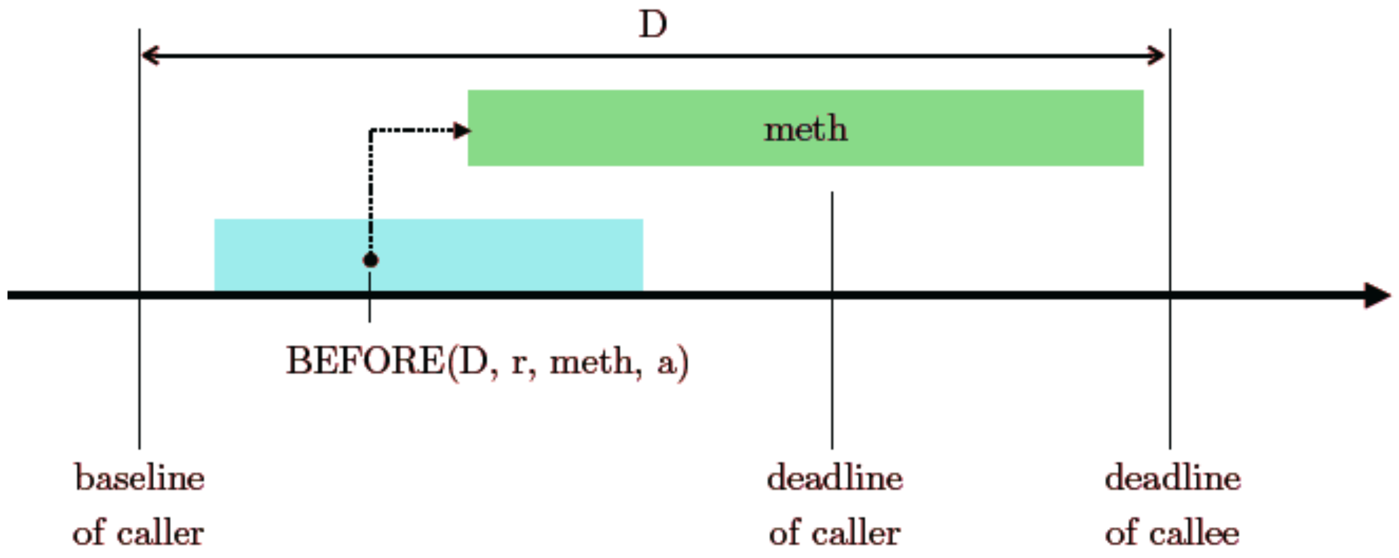
```
int do_some_work(MyObject *self, int n) {
    AFTER(SEC(T),&obj,do_more_work,0);
    ... // do some work
}
```

Examples with BEFORE()



```
// Call method ‘stop’ in object ‘engine’ with a
// deadline of 30 ms
BEFORE(MSEC(30),&engine,stop,0);

// Call method ’move’ in object ’motor’ with a
// deadline of 2 µs
BEFORE(USEC(2),&motor,move,1);
```

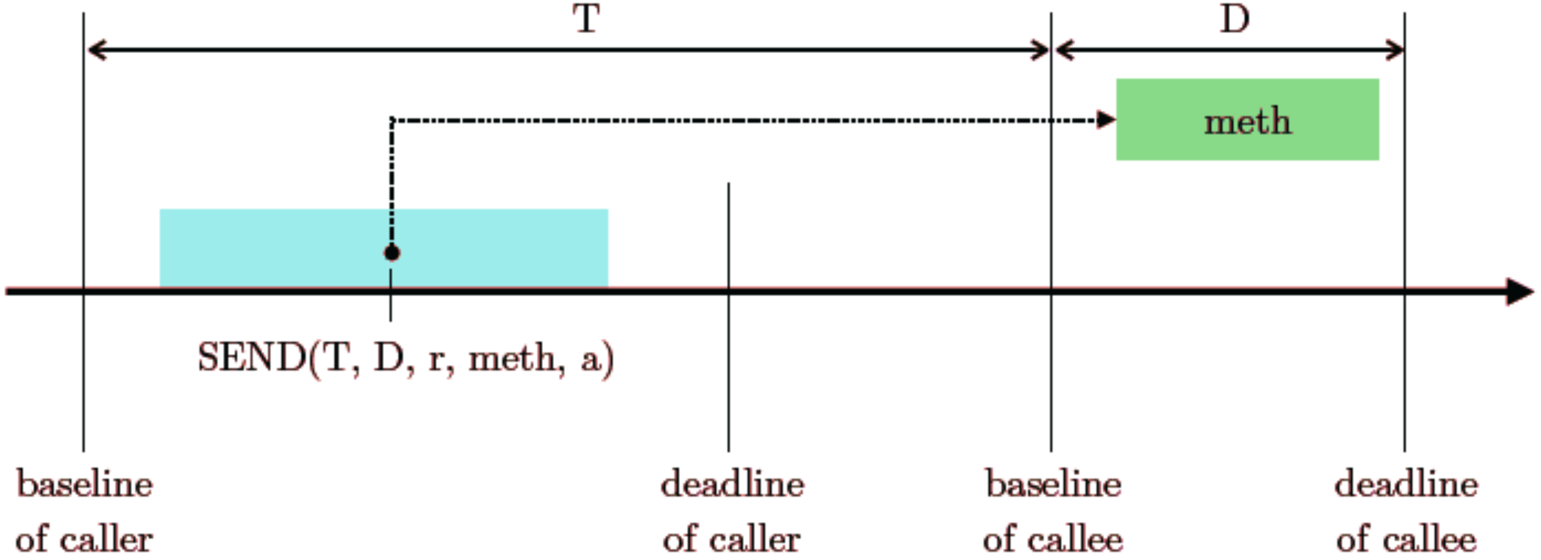
Some more about `BEFORE()`



The `BEFORE()` call has an implicit baseline of 0, i.e., the called method runs with the same baseline as the caller.

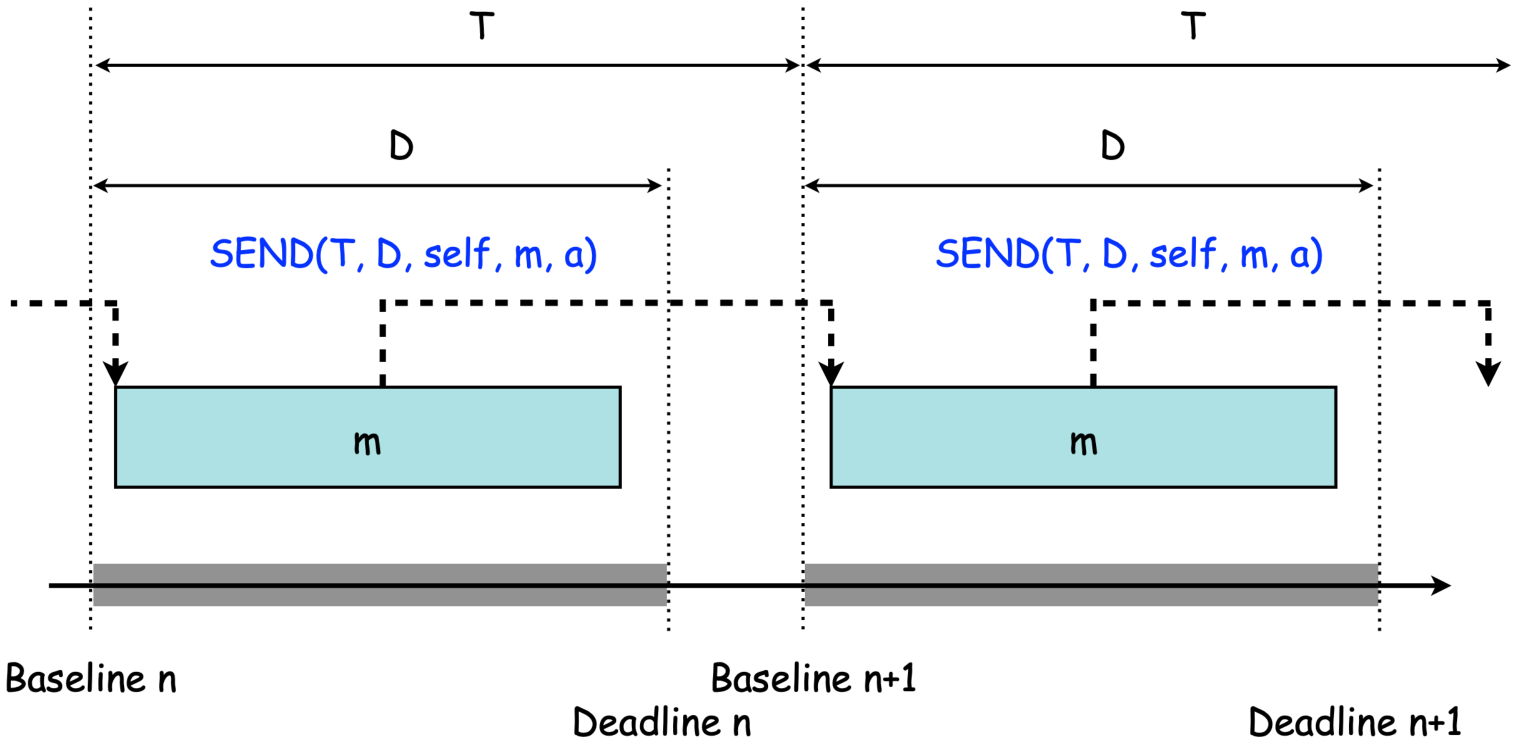
To assign a deadline to a delayed method call, you need to use the `SEND()` call.

Examples with SEND()



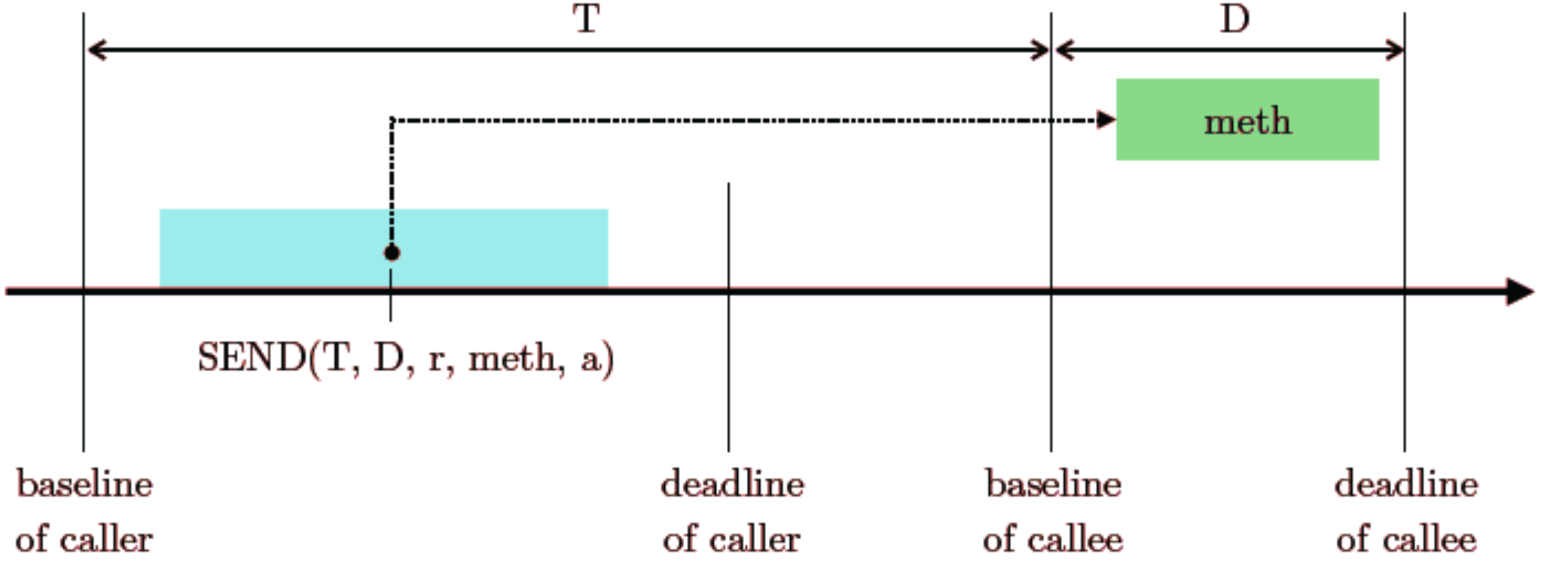
```
// Call method 'stop' in object 'engine' after 2 s,
// with a deadline of 30 ms
SEND(SEC(2),MSEC(30),&engine,stop,0);

// Call method 'move' in object 'motor' after 50 ms,
// with a deadline of 2 µs
SEND(MSEC(50),USEC(2),&motor,move,1);
```

Periodicity with SEND()



```
// Call yourself after 15 ms, with a deadline of 30 µs
int do_some_work(MyObject *self, int n) {
        ... // do some work
        SEND(MSEC(15),USEC(30),self,do_some_work,0);
}
```

Some more about SEND()



The SEND() call is the fundamental building block for the AFTER, BEFORE and ASYNC calls.

```
AFTER(T,&obj,meth,n)  == SEND(T,0,&obj,meth,n);
BEFORE(D,&obj,meth,n)  == SEND(0,D,&obj,meth,n);
ASYNC(&obj,meth,n)  == SEND(0,0,&obj,meth,n);
```

Example: periodic tasks

Problem: Implement two periodic tasks with a shared object in C using the TinyTimber kernel.

- Assume that an object `actobj` of type `Actuator` is shared by two periodic tasks `task1` and `task2` with periods 300 µs and 500 µs, respectively.
- Both tasks may concurrently call method `update` of shared object `actobj` with a value 10 and 20, respectively.
- The old value of object `actobj` should be returned by the `update` method, to be used by the tasks.

Example: periodic tasks

Problem: Implement two periodic tasks with a shared object in C using the TinyTimber kernel.

- Assume that an object `actobj` of type `Actuator` is shared by two periodic tasks `task1` and `task2` with periods 300 μs and 500 μs, respectively.
- Both tasks may concurrently call method `update` of shared object `actobj` with a value 10 and 20, respectively.
- The old value of object `actobj` should be returned by the `update` method, to be used by the tasks.
- Add deadlines of 100 μs and 150 μs to `task1` and `task2`, respectively.

Example: periodic tasks

Problem: Implement two periodic tasks with a shared object in C using the TinyTimber kernel.

- Assume that an object `actobj` of type `Actuator` is shared by two periodic tasks `task1` and `task2` with periods 300 μs and 500 μs, respectively.
- Both tasks may concurrently call method `update` of shared object `actobj` with a value 10 and 20, respectively.
- The old value of object `actobj` should be returned by the `update` method, to be used by the tasks.
- Add deadlines of 100 μs and 150 μs to `task1` and `task2`, respectively.
- Stop the execution of `task1` and `task2` after 100 ms and 200 ms, respectively.

Given code for Actuator object

```c
typedef struct {
    Object super;
    int state;
} Actuator;

// Declare the update method

int update(Actuator *, int);

// Initialization macro

#define initActuator() { initObject(), 0 }

// Create an object of type Actuator

Actuator actobj = initActuator();
```

Given code for Actuator object

```
// This method updates the hardware with a new setting, and
// returns the old setting.

int update(Actuator *self, int new_value){
    int old_value = self->state;
    self->state = new_value;
    ...           // code updating the actuator hardware
    return old_value;
}
```

Template code for periodic tasks

```
typedef struct {// Class definition
    Object super;




} TaskObject;

// Method declarations
void task1code(TaskObject *, int);
void task2code(TaskObject *, int);

// Initialization macro
#define initTaskObject(        ) { initObject()                }

// Create two objects of type TaskObject
TaskObject task1 = initTaskObject(                            );
TaskObject task2 = initTaskObject(                            );
```

Template code for periodic tasks

```
// Each task sends a new value to method actobj, and uses
// the old value returned from method actobj
void task1code(TaskObject *self, int value){

   int old_state = SYNC(&actobj, update, value);
   ...       // do something with the old value



}

void task2code(TaskObject *self, int value){

   int old_state = SYNC(&actobj, update, value);
   ...       // do something else with the old value



}
```

Template code for periodic tasks

```
// How to begin the initial invocation?

void kickoff(TaskObject *self , int unused) {
   // Give an initial value of 10 for task1, and 20 for task2



}

int main() {
   TINYTIMBER(&task1, kickoff, 0);
   return 0;
}
```