Embedded Systems Programming - PA8001

http://bit.ly/15mmqf7

Lecture 7

Mohammad Mousavi

m.r.mousavi@hh.se

HALMSTAD UNIVERSITY

For the Development of Organisations, Products and Quality of Life

Center for Research on Embedded Systems

School of Information Science, Computer and Electrical Engineering

Real Time?

In what ways can a program be related to time in the environment (the *real time*)?

Salvador Dali, The Persistence of Memory.

Real Time

An external process to . . .

- Sample: reading a clock,
- React: a handler for an interrupt clock, and
- Constraint: a deadlock to respect.

Real Time

An external process to . . .

- Sample: reading a clock,
- React: a handler for an interrupt clock, and
- Constraint: a deadlock to respect.

Real Time

An external process to . . .

- Sample: reading a clock,
- React: a handler for an interrupt clock, and
- Constraint: a deadlock to respect.

Sampling the time

Requires a hardware clock (read as an external device)

Multitude of alternatives

- Units? Seconds? Milliseconds? CPU cycles?
- Since when? Program start? System boot? Jan 1, 1970?
- Real time? Time stops when: other threads are running? when CPU sleeps? Time that cannot be set and always increases?

Sampling the time

Requires a hardware clock (read as an external device)

Multitude of alternatives

- Units? Seconds? Milliseconds? CPU cycles?
- Since when? Program start? System boot? Jan 1, 1970?
- Real time? Time stops when: other threads are running? when CPU sleeps? Time that cannot be set and always increases?

Sampling the time

Requires a hardware clock (read as an external device)

Multitude of alternatives

- Units? Seconds? Milliseconds? CPU cycles?
- Since when? Program start? System boot? Jan 1, 1970?
- Real time? Time stops when: other threads are running? when CPU sleeps? Time that cannot be set and always increases?

Sampling the time

Requires a hardware clock (read as an external device)

Multitude of alternatives

- Units? Seconds? Milliseconds? CPU cycles?
- Since when? Program start? System boot? Jan 1, 1970?
- Real time? Time stops when: other threads are running? when CPU sleeps? Time that cannot be set and always increases?

Sampling the time

Requires a hardware clock (read as an external device)

Multitude of alternatives

- Units? Seconds? Milliseconds? CPU cycles?
- Since when? Program start? System boot? Jan 1, 1970?
- Real time? Time stops when: other threads are running? when CPU sleeps? Time that cannot be set and always increases?

Timer/Counter1 on the AVR

What about the 16-bit counter (accessible through register TCNT1)?

Units
CPU clock (8Mhz) divided by a programmable prescaling value (1, 8, 64, 256, 1024).

Since when
System reset, timer reset or timer overflow (whichever was last).

Real time
Shows real time although can be stopped.

Aligning TCNT1 with calendar time: calculations and extra storage (for counting overflows).

Timer/Counter1 on the AVR

What about the 16-bit counter (accessible through register TCNT1)?

Units
CPU clock (8Mhz) divided by a programmable prescaling value (1, 8, 64, 256, 1024).

Since when
System reset, timer reset or timer overflow (whichever was last).

Real time
Shows real time although can be stopped.

Aligning TCNT1 with calendar time: calculations and extra storage (for counting overflows).

Timer/Counter1 on the AVR

What about the 16-bit counter (accessible through register TCNT1)?

Units
CPU clock (8Mhz) divided by a programmable prescaling value (1, 8, 64, 256, 1024).

Since when
System reset, timer reset or timer overflow (whichever was last).

Real time
Shows real time although can be stopped.

Aligning TCNT1 with calendar time: calculations and extra storage (for counting overflows).

Timer/Counter1 on the AVR

What about the 16-bit counter (accessible through register TCNT1)?

Units
CPU clock (8Mhz) divided by a programmable prescaling value (1, 8, 64, 256, 1024).

Since when
System reset, timer reset or timer overflow (whichever was last).

Real time
Shows real time although can be stopped.

Aligning TCNT1 with calendar time: calculations and extra storage (for counting overflows).

Timer/Counter1 on the AVR

What about the 16-bit counter (accessible through register TCNT1)?

Units
CPU clock (8Mhz) divided by a programmable prescaling value (1, 8, 64, 256, 1024).

Since when
System reset, timer reset or timer overflow (whichever was last).

Real time
Shows real time although can be stopped.

Aligning TCNT1 with calendar time: calculations and extra storage (for counting overflows).

Timestamps

Relative timing: prevalent in reactive systems, reactions are relative to events

Example

Teacher left 15 min. after the start of the lecture.

In embedded programming, time-stamping an event: reading performed around the event detection.

Timestamps

Relative timing: prevalent in reactive systems, reactions are relative to events

Example

Teacher left 15 min. after the start of the lecture.

In embedded programming, time-stamping an event: reading performed around the event detection.

Timestamps

Relative timing: prevalent in reactive systems, reactions are relative to events

Example

Teacher left 15 min. after the start of the lecture.

In embedded programming, time-stamping an event: reading performed around the event detection.

Time spans

The difference between two time-stamps: a time span independent of the nominal clock values (modulo clock resolution).

The meaning of time-stamp

- The time of some arbitrary program instruction?
- The beginning or end of a function call?
- The time of sending or receiving an asynchronous message?

Too much program dependent!

Time spans

The difference between two time-stamps: a time span independent of the nominal clock values (modulo clock resolution).

The meaning of time-stamp

- The time of some arbitrary program instruction?
- The beginning or end of a function call?
- The time of sending or receiving an asynchronous message?

Too much program dependent!

Time spans

The difference between two time-stamps: a time span independent of the nominal clock values (modulo clock resolution).

The meaning of time-stamp

- The time of some arbitrary program instruction?
- The beginning or end of a function call?
- The time of sending or receiving an asynchronous message?

Too much program dependent!

Time spans

The difference between two time-stamps: a time span independent of the nominal clock values (modulo clock resolution).

The meaning of time-stamp

- The time of some arbitrary program instruction?
- The beginning or end of a function call?
- The time of sending or receiving an asynchronous message?

Too much program dependent!

Time spans

The difference between two time-stamps: a time span independent of the nominal clock values (modulo clock resolution).

The meaning of time-stamp

- The time of some arbitrary program instruction?
- The beginning or end of a function call?
- The time of sending or receiving an asynchronous message?

Too much program dependent!

Time spans

The difference between two time-stamps: a time span independent of the nominal clock values (modulo clock resolution).

The meaning of time-stamp

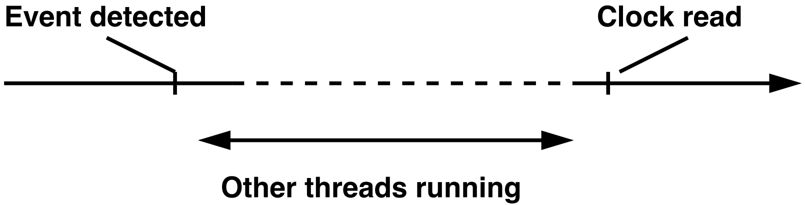
- The time of some arbitrary program instruction?
- The beginning or end of a function call?
- The time of sending or receiving an asynchronous message?

Too much program dependent!

In a scheduled system

What looks like . . .

might very well be . . .

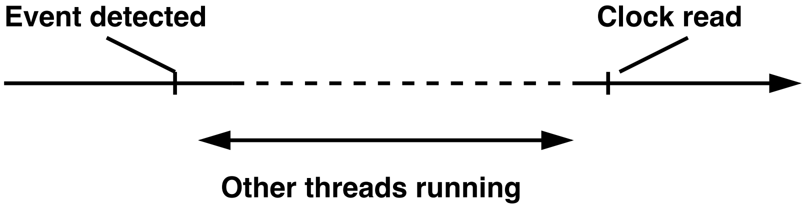


Close proximity is not the same as subsequent statements!

In a scheduled system

What looks like . . .

might very well be . . .



Close proximity is not the same as subsequent statements!

Time-stamping events

Our goal: to time-stamp events that *drive* a system

Idea!

Read the clock in the interrupt handler detecting the event

- Disable other interrupts, hence no threads might interfere
- Tight predictable upper bound on the time-stamp error

Time-stamping events

Our goal: to time-stamp events that *drive* a system

Idea!

Read the clock in the interrupt handler detecting the event

- Disable other interrupts, hence no threads might interfere
- Tight predictable upper bound on the time-stamp error

Time-stamping events

Our goal: to time-stamp events that *drive* a system

Idea!

Read the clock in the interrupt handler detecting the event

- Disable other interrupts, hence no threads might interfere
- Tight predictable upper bound on the time-stamp error

Time-stamping events

Our goal: to time-stamp events that *drive* a system

Idea!

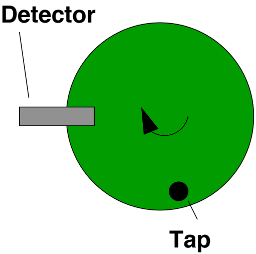
Read the clock in the interrupt handler detecting the event

- Disable other interrupts, hence no threads might interfere
- Tight predictable upper bound on the time-stamp error

Example

Calculate the speed

For a rotating wheel, measuring the time between two subsequent detections of a passing tap.

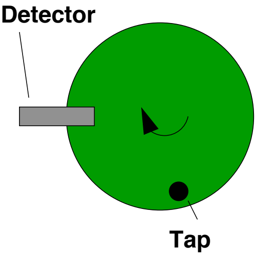



```
typedef struct{
 Object super;
 int previous;
 Other *client;
} Speedo;
...
Speedo speedo;
int main(){
   INSTALL(&speedo, detect, SIG_XX);
   return TINYTIMBER(...)
}
```

Example

Calculate the speed

For a rotating wheel, measuring the time between two subsequent detections of a passing tap.




```
typedef struct{
 Object super;
 int previous;
 Other *client;
} Speedo;
...
Speedo speedo;
int main(){
   INSTALL(&speedo, detect, SIG_XX);
   return TINYTIMBER(...)
}
```

Example

Calculate the speed

For a rotating wheel, measuring the time between two subsequent detections of a passing tap.

```
int detect(Speedo *self, int sig){
   int timestamp = TCNT1;
   ASYNC(self -> client,
        newSpeed,
        PERIMETER/DIFF(timestamp,self->previous));
   self->previous=timestamp;
}
```

DIFF will have ot take care of timer overflows!

Example

Calculate the speed

For a rotating wheel, measuring the time between two subsequent detections of a passing tap.

```
int detect(Speedo *self, int sig){
   int timestamp = TCNT1;
   ASYNC(self -> client,
        newSpeed,
        PERIMETER/DIFF(timestamp,self->previous));
   self->previous=timestamp;
}
```

DIFF will have ot take care of timer overflows!

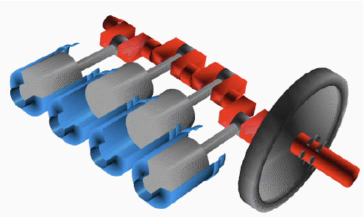
Real-time events to react to

So far: how to sample the real-time clock to know about time

Now: how to take action after a certain amount of time

Example

The wheel is an engine crankshaft and we have to emit ignition signals to each cylinder



How to postpone program execution until certain time

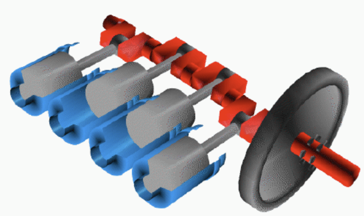
Real-time events to react to

So far: how to sample the real-time clock to know about time

Now: how to take action after a certain amount of time

Example

The wheel is an engine crankshaft and we have to emit ignition signals to each cylinder



How to postpone program execution until certain time

Reacting to real time events

Very poor man's solution

Consume a fixed amount of CPU cycles in a (silly) loop

```
int i;
for(i=0;i<N;i++);  // wait
do_future_action();
```

Problems

1. Determine N by testing!
2. N will be highly platform dependent!
3. A lot of CPU cycles will simply be wasted!

Reacting to real time events

Very poor man's solution

Consume a fixed amount of CPU cycles in a (silly) loop

```
int i;
for(i=0;i<N;i++);  // wait
do_future_action();
```

Problems

1. Determine N by testing!
2. N will be highly platform dependent!
3. A lot of CPU cycles will simply be wasted!

Reacting to real time events

Very poor man's solution

Consume a fixed amount of CPU cycles in a (silly) loop

```
int i;
for(i=0;i<N;i++);  // wait
do_future_action();
```

Problems

1. Determine `N` by testing!
2. `N` will be highly platform dependent!
3. A lot of CPU cycles will simply be wasted!

Reacting to real time events

Very poor man's solution

Consume a fixed amount of CPU cycles in a (silly) loop

```
int i;
for(i=0;i<N;i++);  // wait
do_future_action();
```

Problems

1. Determine `N` by testing!
2. `N` will be highly platform dependent!
3. A lot of CPU cycles will simply be wasted!

Reacting to real time events

Very poor man's solution

Consume a fixed amount of CPU cycles in a (silly) loop

```
int i;
for(i=0;i<N;i++);   // wait
do_future_action();
```

Problems

1. Determine `N` by testing!
2. `N` will be highly platform dependent!
3. A lot of CPU cycles will simply be wasted!

Reacting to real time events

The nearly as poor man's solution

Configure a timer/counter with a known clock speed, and busy-wait for a suitable time increment

```
unsigned int i = TCNT1+N;
while(TCNT1<i); // wait
do_future_action();
```

Problems

1. Determine N by calculation
2. Still a lot of wasted CPU!

Reacting to real time events

The nearly as poor man's solution

Configure a timer/counter with a known clock speed, and busy-wait for a suitable time increment

```
unsigned int i = TCNT1+N;
while(TCNT1<i); // wait
do_future_action();
```

Problems

1. Determine `N` by calculation
2. Still a lot of wasted CPU!

Reacting to real time events

The nearly as poor man's solution

Configure a timer/counter with a known clock speed, and busy-wait for a suitable time increment

```
unsigned int i = TCNT1+N;
while(TCNT1<i); // wait
do_future_action();
```

Problems

1. Determine `N` by calculation
2. Still a lot of wasted CPU!

Reacting to real time events

The nearly as poor man's solution

Configure a timer/counter with a known clock speed, and busy-wait for a suitable time increment

```
unsigned int i = TCNT1+N;
while(TCNT1<i); // wait
do_future_action();
```

Problems

1. Determine N by calculation
2. Still a lot of wasted CPU!

Reacting to real time events

The standard solution

Use the OS to *fake* busy-waiting

```
delay(N);   // wait (blocking OS call)
do_future_action();
```

- No platform dependency!
- No wasted CPU cycles (at the expense of a complex OS)

Still a problem . . .

. . . common to all solutions . . .

Reacting to real time events

The standard solution

Use the OS to *fake* busy-waiting

```
delay(N);    // wait (blocking OS call)
do_future_action();
```

- No platform dependency!
- No wasted CPU cycles (at the expense of a complex OS)

Still a problem . . .

. . . common to all solutions . . .

Reacting to real time events

The standard solution

Use the OS to *fake* busy-waiting

```
delay(N);   // wait (blocking OS call)
do_future_action();
```

- No platform dependency!
- No wasted CPU cycles (at the expense of a complex OS)

Still a problem ...

... common to all solutions ...

Reacting to real time events

The standard solution

Use the OS to *fake* busy-waiting

```
delay(N);   // wait (blocking OS call)
do_future_action();
```

- No platform dependency!
- No wasted CPU cycles (at the expense of a complex OS)

Still a problem ...

...common to all solutions ...

Reacting to real time events

The standard solution

Use the OS to *fake* busy-waiting

```
delay(N);   // wait (blocking OS call)
do_future_action();
```

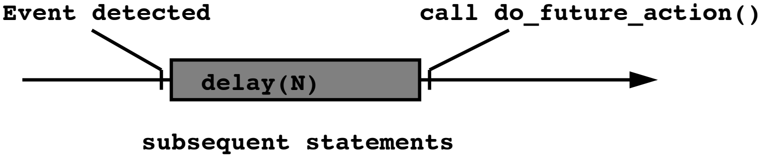
- No platform dependency!
- No wasted CPU cycles (at the expense of a complex OS)

Still a problem . . .

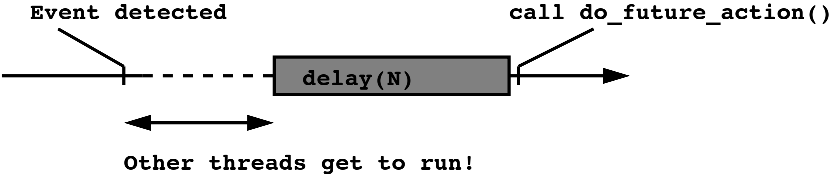
. . . common to all solutions . . .

In a scheduled system

What looks like . . .



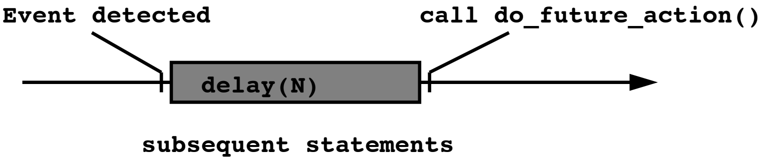
might very well be . . .



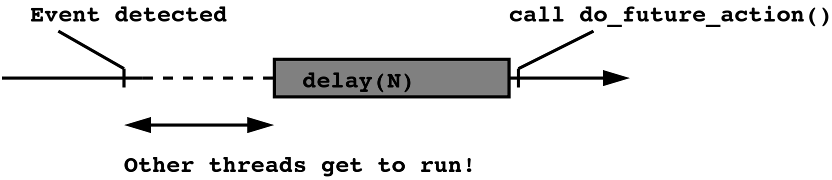
Had we known the scheduler's choice, a smaller N had been used!

In a scheduled system

What looks like . . .



might very well be . . .



Had we known the scheduler's choice, a smaller N had been used!

Relative delays

The problem: relative time without fixed references:

- The constructed real-time event will occur at after `N` units from *now*.
- What is *now*?!

Other common OS services share this problem: `sleep`, `usleep` and `nanosleep`.

We are not going to use OS services in the course.

Relative delays

The problem: relative time without fixed references:

- The constructed real-time event will occur at after `N` units from *now*.
- What is *now*?!

Other common OS services share this problem: `sleep`, `usleep` and `nanosleep`.

We are not going to use OS services in the course.

Relative delays

The problem: relative time without fixed references:

- The constructed real-time event will occur at after `N` units from *now*.
- What is *now*?!

Other common OS services share this problem: `sleep`, `usleep` and `nanosleep`.

We are not going to use OS services in the course.

Relative delays

The problem: relative time without fixed references:

- The constructed real-time event will occur at after `N` units from *now*.
- What is *now*?!

Other common OS services share this problem: `sleep`, `usleep` and `nanosleep`.

We are not going to use OS services in the course.

Relative delays

The problem: relative time without fixed references:

- The constructed real-time event will occur at after `N` units from *now*.
- What is *now*?!

Other common OS services share this problem: `sleep`, `usleep` and `nanosleep`.

We are not going to use OS services in the course.

Yet another problem

Threads and interleaving make it worse

Example

Consider a task running a CPU-heavy function `do_work()` every 100 millisecods. The naive implementation sing `delay()`:

```
while(1){
  do_work();
  delay(100);
}
```

Yet another problem

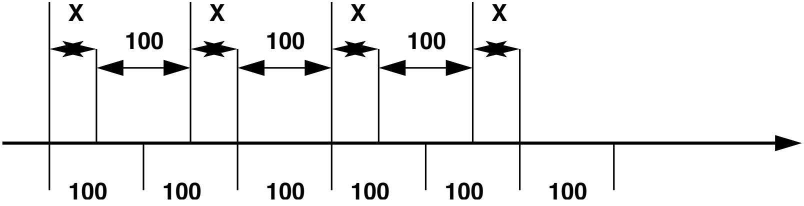
Threads and interleaving make it worse

Example

Consider a task running a CPU-heavy function `do_work()` every 100 millisecods. The naive implementation sing `delay()`:

```
while(1){
  do_work();
  delay(100);
}
```

Accumulating drift

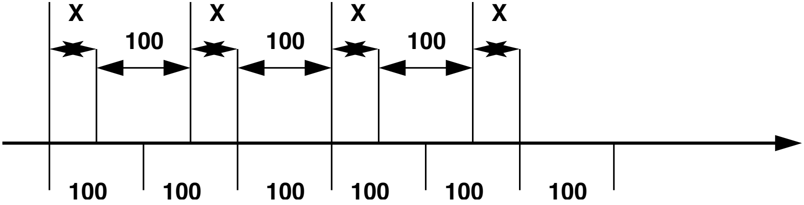


`X` is the time take to `do_work`

Each turn takes at least 100+`X` milliseconds.

A drift of `X` milliseconds will accumulate every turn!

Accumulating drift

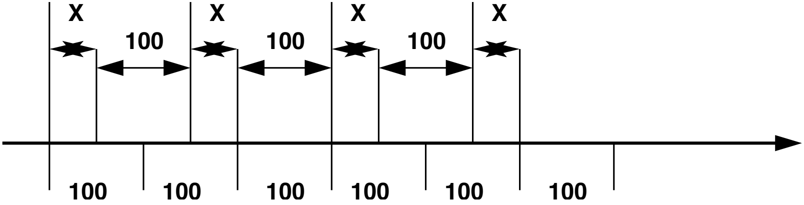


X is the time take to `do_work`

Each turn takes at least 100+X milliseconds.

A drift of X milliseconds will accumulate every turn!

Accumulating drift

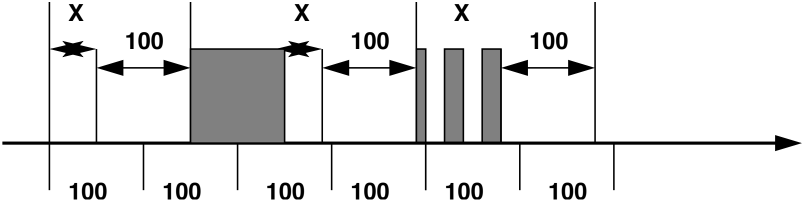


X is the time take to `do_work`

Each turn takes at least 100+X milliseconds.

A drift of X milliseconds will accumulate every turn!

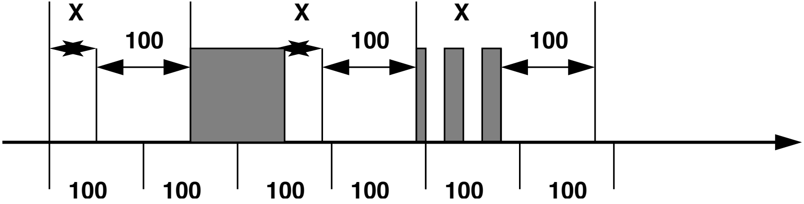
Accumulating drift



With threads and interleaving, the bad scenario gets worse!

Even with a known X, delay time is not predictable.

Accumulating drift



With threads and interleaving, the bad scenario gets worse!

Even with a known `X`, delay time is not predictable.

A stable reference

What we need is a stable time reference to use as a basis whenever we specify a relative time (instead of now).

Baselines

We introduce the baseline of a message to mean the earliest time a message is allowed to start.

Time stamps of interrupts!

The baseline of an event is its time-stamp:

A stable reference

What we need is a stable time reference to use as a basis whenever we specify a relative time (instead of now).

Baselines

We introduce the baseline of a message to mean the earliest time a message is allowed to start.

Time stamps of interrupts!

The baseline of an event is its time-stamp:

A stable reference

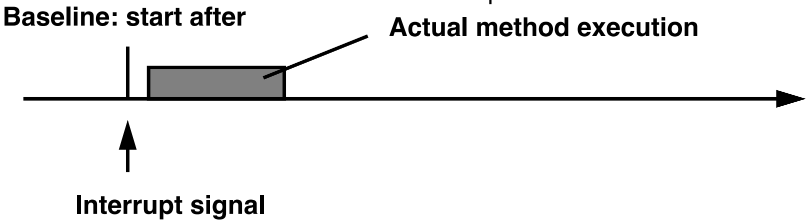
What we need is a stable time reference to use as a basis whenever we specify a relative time (instead of now).

Baselines

We introduce the baseline of a message to mean the earliest time a message is allowed to start.

Time stamps of interrupts!

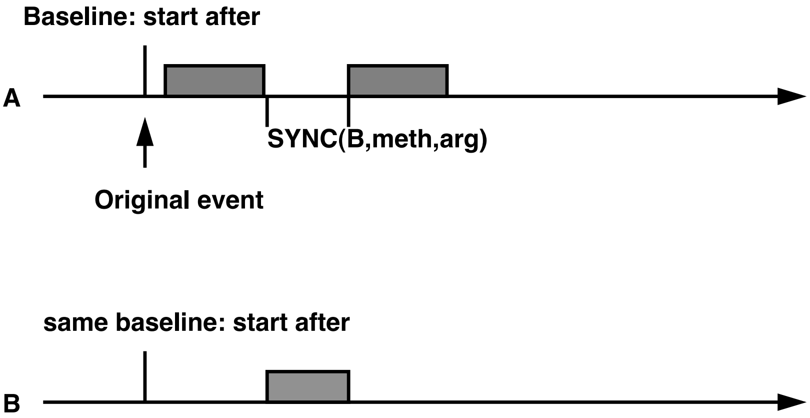
The baseline of an event is its time-stamp:



A stable reference

SYNC

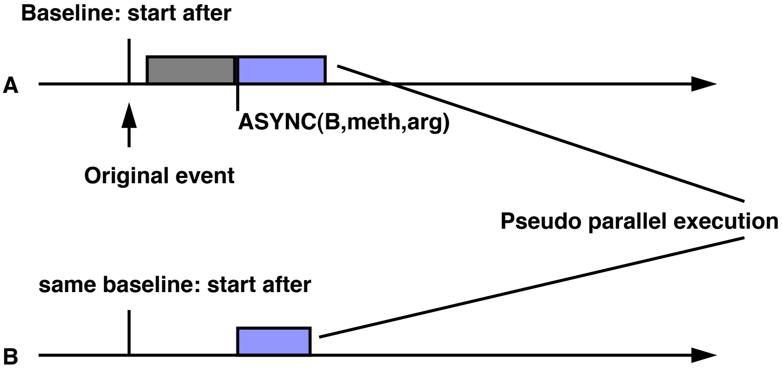
Calling methods with `SYNC` doesn't change the baseline (the call inherits the baseline)



A stable reference

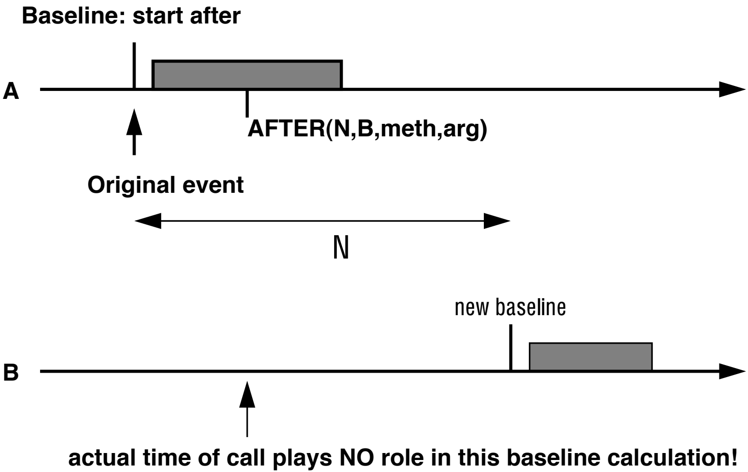
ASYNC

By default `ASYNC` method calls will inherit the baseline



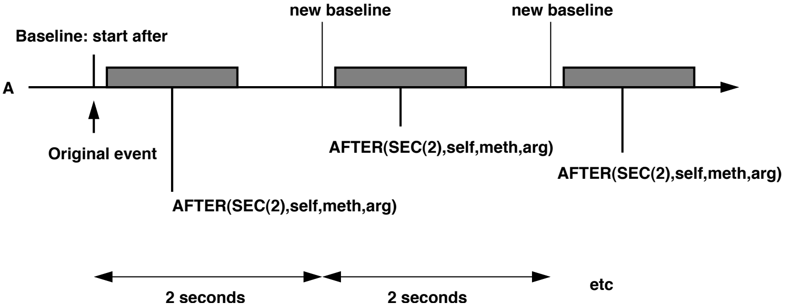
A stable reference

For `ASYNC` we may also consider adding a baseline offset `N`!



Periodic tasks

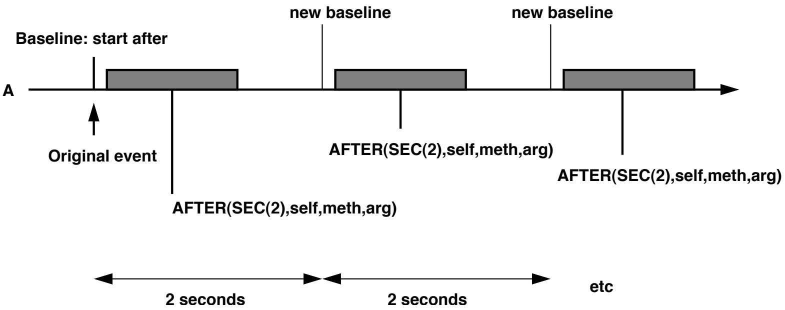
To create a cyclic reaction, simply call **self** with the same method and a new baseline:



SEC is a convenient macro that makes the call independent of current timer resolution.

Periodic tasks

To create a cyclic reaction, simply call **self** with the same method and a new baseline:



`SEC` is a convenient macro that makes the call independent of current timer resolution.

Implementing AFTER

1. Let the baseline be stored in every message (as part of the `Msg` structure)
2. `AFTER` is the same as `ASYNC`, but
   - New baseline is `MAX(now, offset+current->baseline)`
   - If `baseline > now` , put message in a timerQ instead of readyQ
   - Set up a timer to generate an interrupt after earliest baseline
   - At each timer interrupt, move first timerQ message to readyQ and configure a new timer interrupt

In fact `ASYNC` can now be defined as

```
#define ASYNC(to,meth,arg) AFTER(0,to,meth,arg)
```

Implementing AFTER

1. Let the baseline be stored in every message (as part of the `Msg` structure)
2. `AFTER` is the same as `ASYNC`, but
   - New baseline is
     `MAX(now, offset+current->baseline)`
   - If `baseline > now` , put message in a timerQ instead of readyQ
   - Set up a timer to generate an interrupt after earliest baseline
   - At each timer interrupt, move first timerQ message to readyQ and configure a new timer interrupt

In fact `ASYNC` can now be defined as

```
#define ASYNC(to,meth,arg) AFTER(0,to,meth,arg)
```

Implementing AFTER

1. Let the baseline be stored in every message (as part of the `Msg` structure)
2. `AFTER` is the same as `ASYNC`, but
   - New baseline is `MAX(now, offset+current->baseline)`
   - If `baseline > now` , put message in a timerQ instead of readyQ
   - Set up a timer to generate an interrupt after earliest baseline
   - At each timer interrupt, move first timerQ message to readyQ and configure a new timer interrupt

In fact `ASYNC` can now be defined as

```
#define ASYNC(to,meth,arg) AFTER(0,to,meth,arg)
```

Implementing AFTER

1. Let the baseline be stored in every message (as part of the Msg structure)
2. AFTER is the same as ASYNC, but
   - New baseline is

     `MAX(now, offset+current->baseline)`
   - If `baseline > now` , put message in a timerQ instead of readyQ
   - Set up a timer to generate an interrupt after earliest baseline
   - At each timer interrupt, move first timerQ message to readyQ and configure a new timer interrupt

In fact ASYNC can now be defined as

```
#define ASYNC(to,meth,arg) AFTER(0,to,meth,arg)
```

Implementing `AFTER`

1. Let the baseline be stored in every message (as part of the `Msg` structure)
2. `AFTER` is the same as `ASYNC`, but
   - New baseline is
     `MAX(now, offset+current->baseline)`
   - If `baseline > now` , put message in a timerQ instead of readyQ
   - Set up a timer to generate an interrupt after earliest baseline
   - At each timer interrupt, move first timerQ message to readyQ and configure a new timer interrupt

In fact `ASYNC` can now be defined as

```
#define ASYNC(to,meth,arg) AFTER(0,to,meth,arg)
```

Implementing `AFTER`

1. Let the baseline be stored in every message (as part of the `Msg` structure)
2. `AFTER` is the same as `ASYNC`, but
   - New baseline is
     `MAX(now, offset+current->baseline)`
   - If `baseline > now` , put message in a timerQ instead of readyQ
   - Set up a timer to generate an interrupt after earliest baseline
   - At each timer interrupt, move first timerQ message to readyQ and configure a new timer interrupt

In fact `ASYNC` can now be defined as

```
#define ASYNC(to,meth,arg) AFTER(0,to,meth,arg)
```

Implementing AFTER

1. Let the baseline be stored in every message (as part of the Msg structure)
2. AFTER is the same as ASYNC, but
   - New baseline is

     `MAX(now, offset+current->baseline)`
   - If `baseline > now` , put message in a timerQ instead of readyQ
   - Set up a timer to generate an interrupt after earliest baseline
   - At each timer interrupt, move first timerQ message to readyQ and configure a new timer interrupt

In fact ASYNC can now be defined as

```
#define ASYNC(to,meth,arg) AFTER(0,to,meth,arg)
```

Implementing AFTER

1. Let the baseline be stored in every message (as part of the Msg structure)
2. AFTER is the same as ASYNC, but
   - New baseline is
     `MAX(now, offset+current->baseline)`
   - If `baseline > now` , put message in a timerQ instead of readyQ
   - Set up a timer to generate an interrupt after earliest baseline
   - At each timer interrupt, move first timerQ message to readyQ and configure a new timer interrupt

In fact ASYNC can now be defined as

```
#define ASYNC(to,meth,arg) AFTER(0,to,meth,arg)
```

Bonus Questions

What are the issues with time in a distributed system? Find out what Lamport Clocks are and explain them (in your own words) in a few lines.
(Please send your answers by email before 13:00 today.)