Embedded Systems Programming

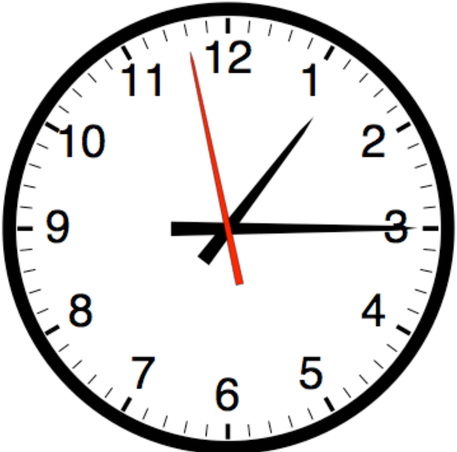
Lecture 7

Verónica Gaspes
www2.hh.se/staff/vero

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*)?



Real Time could be . . .

An external process to sample

Reading a real-time clock is like sampling any other external process value!

An external process to react to

A program can let certain points in time denote events (e.g. by means of interrupts of a clock) to which it has to react in some way or another.

An external process to be constrained by

A program might be required to hurry enough so that some externally visible action can be performed before a certain point in time (comming later on in the course)

Real Time could be . . .

An external process to sample

Reading a real-time clock is like sampling any other external process value!

An external process to react to

A program can let certain points in time denote events (e.g. by means of interrupts of a clock) to which it has to react in some way or another.

An external process to be constrained by

A program might be required to hurry enough so that some externally visible action can be performed before a certain point in time (comming later on in the course)

Real Time could be . . .

An external process to sample

Reading a real-time clock is like sampling any other external process value!

An external process to react to

A program can let certain points in time denote events (e.g. by means of interrupts of a clock) to which it has to react in some way or another.

An external process to be constrained by

A program might be required to hurry enough so that some externally visible action can be performed before a certain point in time (comming later on in the course)

Real Time could be . . .

An external process to sample

Reading a real-time clock is like sampling any other external process value!

An external process to react to

A program can let certain points in time denote events (e.g. by means of interrupts of a clock) to which it has to react in some way or another.

An external process to be constrained by

A program might be required to hurry enough so that some externally visible action can be performed before a certain point in time (comming later on in the course)

Sampling the time

Requires a hardware clock that can be read like a regular external device.

Multitude of alternatives

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

Sampling the time

Requires a hardware clock that can be read like a regular external device.

Multitude of alternatives

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

Sampling the time

Requires a hardware clock that can be read like a regular external device.

Multitude of alternatives

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

Sampling the time

Requires a hardware clock that can be read like a regular external device.

Multitude of alternatives

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

Sampling the time

Requires a hardware clock that can be read like a regular external device.

Multitude of alternatives

- Units? Seconds? Milliseconds? Cpu cycles?
- Since when? Program start? System boot? Jan 1 1970?
- Real time? Time that stops when other threads are running? Time that stops 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 it can be stopped.

Aligning TCNT1 with calendar time will require 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 it can be stopped.

Aligning TCNT1 with calendar time will require 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 it can be stopped.

Aligning TCNT1 with calendar time will require 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 it can be stopped.

Aligning TCNT1 with calendar time will require 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 it can be stopped.

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

Timestamps

In general clock readings become meaningful only when they are associated with other events: we are interested to sample to know when an event has occurred.

Example

The clock showed 11:25 when the teacher left.

In program terms, associating a clock reading with some other event means doing the reading in close proximity to the event detection.

Timestamps

In general clock readings become meaningful only when they are associated with other events: we are interested to sample to know when an event has occurred.

Example

The clock showed 11:25 when the teacher left.

In program terms, associating a clock reading with some other event means doing the reading in close proximity to the event detection.

Timestamps

In general clock readings become meaningful only when they are associated with other events: we are interested to sample to know when an event has occurred.

Example

The clock showed 11:25 when the teacher left.

In program terms, associating a clock reading with some other event means doing the reading in close proximity to the event detection.

Time spans

The difference between two timestamps is a value that is independent of the nominal clock values — it is a time span (characterized only by the clock resolution).

What could each timestamp mean?

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

All these are internal to a program under execution and have no meaning to an external observer!

Time spans

The difference between two timestamps is a value that is independent of the nominal clock values — it is a time span (characterized only by the clock resolution).

What could each timestamp mean?

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

All these are internal to a program under execution and have no meaning to an external observer!

Time spans

The difference between two timestamps is a value that is independent of the nominal clock values — it is a time span (characterized only by the clock resolution).

What could each timestamp mean?

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

All these are internal to a program under execution and have no meaning to an external observer!

Time spans

The difference between two timestamps is a value that is independent of the nominal clock values — it is a time span (characterized only by the clock resolution).

What could each timestamp mean?

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

All these are internal to a program under execution and have no meaning to an external observer!

Time spans

The difference between two timestamps is a value that is independent of the nominal clock values — it is a time span (characterized only by the clock resolution).

What could each timestamp mean?

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

All these are internal to a program under execution and have no meaning to an external observer!

Time spans

The difference between two timestamps is a value that is independent of the nominal clock values — it is a time span (characterized only by the clock resolution).

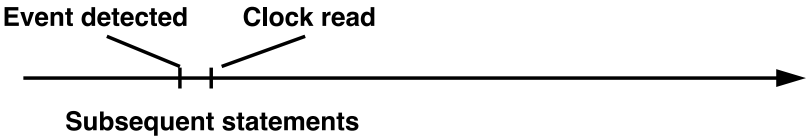
What could each timestamp mean?

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

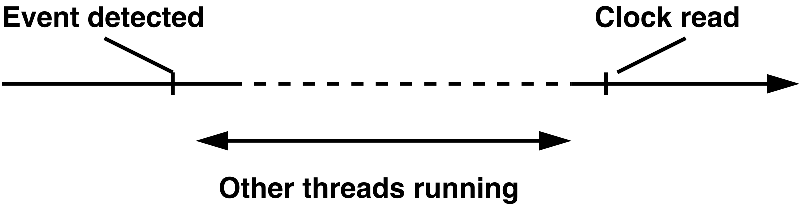
All these are internal to a program under execution and have no meaning to an external observer!

In a scheduled system

What looks like . . .



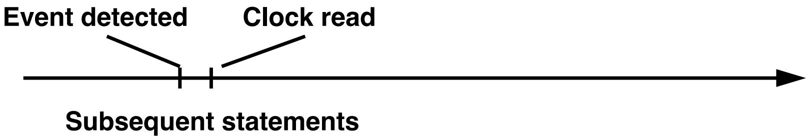
might very well be . . .



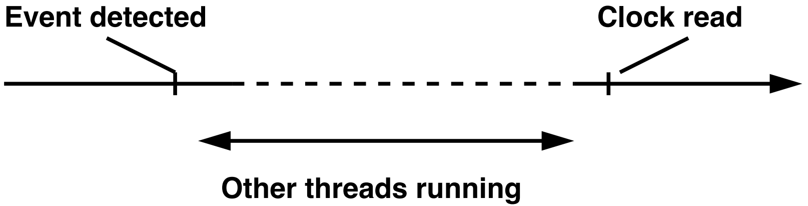
Close proximity is not the same as subsequent statements!

In a scheduled system

What looks like . . .



might very well be . . .



Close proximity is not the same as subsequent statements!

Time-stamping events

What we really want is to associate timestamps with the externally observable events that *drive* a system!

Idea!

Read the clock inside the interrupt handler that detects the asociated event

- Other interrupts are disabled while the CPU runs an interrupt handler, hence no scheduling of threads might interfere!
- There is a tight upper bound on the timestamp error, which can be calculated from CPU data and speed!

Example

One could use an arg of an interrupt driven method for passing a timestamp!

Time-stamping events

What we really want is to associate timestamps with the externally observable events that *drive* a system!

Idea!

Read the clock inside the interrupt handler that detects the asociated event

- Other interrupts are disabled while the CPU runs an interrupt handler, hence no scheduling of threads might interfere!
- There is a tight upper bound on the timestamp error, which can be calculated from CPU data and speed!

Example

One could use an arg of an interrupt driven method for passing a timestamp!

Time-stamping events

What we really want is to associate timestamps with the externally observable events that *drive* a system!

Idea!

Read the clock inside the interrupt handler that detects the asociated event

- Other interrupts are disabled while the CPU runs an interrupt handler, hence no scheduling of threads might interfere!
- There is a tight upper bound on the timestamp error, which can be calculated from CPU data and speed!

Example

One could use an arg of an interrupt driven method for passing a timestamp!

Time-stamping events

What we really want is to associate timestamps with the externally observable events that *drive* a system!

Idea!

Read the clock inside the interrupt handler that detects the asociated event

- Other interrupts are disabled while the CPU runs an interrupt handler, hence no scheduling of threads might interfere!
- There is a tight upper bound on the timestamp error, which can be calculated from CPU data and speed!

Example

One could use an arg of an interrupt driven method for passing a timestamp!

Time-stamping events

What we really want is to associate timestamps with the externally observable events that *drive* a system!

Idea!

Read the clock inside the interrupt handler that detects the asociated event

- Other interrupts are disabled while the CPU runs an interrupt handler, hence no scheduling of threads might interfere!
- There is a tight upper bound on the timestamp error, which can be calculated from CPU data and speed!

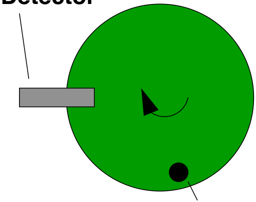
Example

One could use an arg of an interrupt driven method for passing a timestamp!

Example

Calculate the speed

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

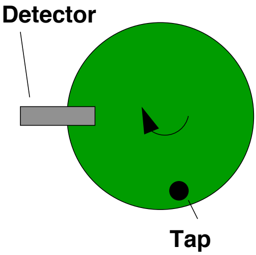



```
typedef struct{
 Object super;
 int previous;
} Speedo;
...
Speedo speedo:
Other  client;
INTERRUPT(SIG_XX,
          ASYNC(&speedo,detect,TCNT1))
```

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;
} Speedo;
...
Speedo speedo:
Other  client;
INTERRUPT(SIG_XX,
         ASYNC(&speedo,detect,TCNT1))
```

Example

Calculate the speed

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

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

DIFF will have ot take care of timer overflows!

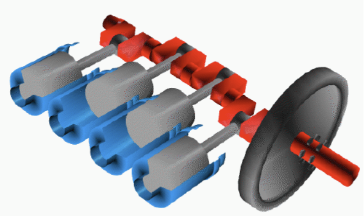
Real-time events to react to

We know how to sample the real-time clock to obtain externally meaningful information about the passage of time.

Now suppose we want to take some action when a certain amount of time has passed.

Example

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



We would need a way to postpone program execution until certain points in the future.

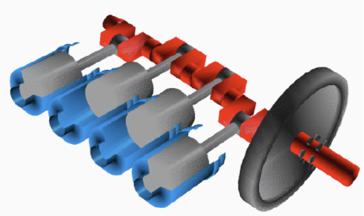
Real-time events to react to

We know how to sample the real-time clock to obtain externally meaningful information about the passage of time.

Now suppose we want to take some action when a certain amount of time has passed.

Example

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



We would need a way to postpone program execution until certain points in the future.

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 so platform dependency disapears.
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 so platform dependency disapears.
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 so platform dependency disapears.
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 so platform dependency disapears.
2. Still a lot of wasted CPU!

Reacting to real time events

The standard solution

Use an Operating system call that *fakes* the timer increment busy-wait loop while making better use of the CPU resources

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

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

Still a problem ...

...common to all solutions ...

Reacting to real time events

The standard solution

Use an Operating system call that *fakes* the timer increment busy-wait loop while making better use of the CPU resources

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

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

Still a problem . . .

. . . common to all solutions . . .

Reacting to real time events

The standard solution

Use an Operating system call that *fakes* the timer increment busy-wait loop while making better use of the CPU resources

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

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

Still a problem ...

...common to all solutions ...

Reacting to real time events

The standard solution

Use an Operating system call that *fakes* the timer increment busy-wait loop while making better use of the CPU resources

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

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

Still a problem ...

...common to all solutions ...

Reacting to real time events

The standard solution

Use an Operating system call that *fakes* the timer increment busy-wait loop while making better use of the CPU resources

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

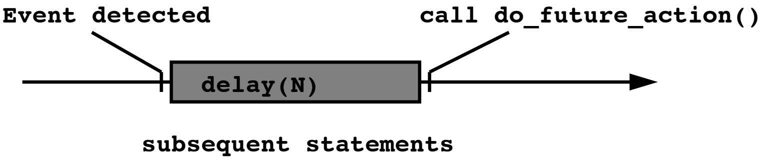
- No platform dependency!
- No wasted CPU cycles (at the expense of complex OS internals)

Still a problem ...

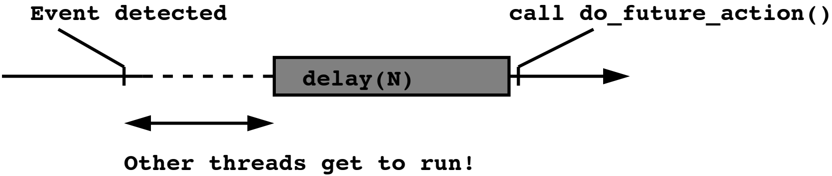
...common to all solutions ...

In a scheduled system

What looks like ...



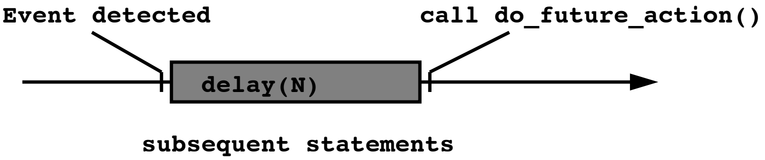
might very well be ...



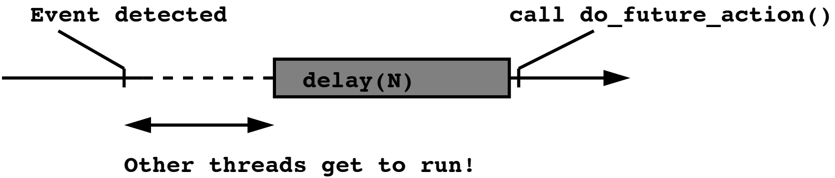
Had we known the scheduler's choice, a smaler `N` had been used!

In a scheduled system

What looks like ...



might very well be ...



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

Relative delays

The problem is that these delay services are always specified using relative time:

- The constructed real-time event will occur at a time obtained by adding the delay parameter `N` to *now*.
- But *now* is not a very meaningful time reference in a scheduled system, as it is not related to any externally observable signals.

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 is that these delay services are always specified using relative time:

- The constructed real-time event will occur at a time obtained by adding the delay parameter `N` to *now*.
- But *now* is not a very meaningful time reference in a scheduled system, as it is not related to any externally observable signals.

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 is that these delay services are always specified using relative time:

- The constructed real-time event will occur at a time obtained by adding the delay parameter `N` to *now*.
- But *now* is not a very meaningful time reference in a scheduled system, as it is not related to any externally observable signals.

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 is that these delay services are always specified using relative time:

- The constructed real-time event will occur at a time obtained by adding the delay parameter `N` to *now*.
- But *now* is not a very meaningful time reference in a scheduled system, as it is not related to any externally observable signals.

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 is that these delay services are always specified using relative time:

- The constructed real-time event will occur at a time obtained by adding the delay parameter `N` to *now*.
- But *now* is not a very meaningful time reference in a scheduled system, as it is not related to any externally observable signals.

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

Even if other threads were not to interfere, using delay services as a means of specifying future points in time has another fundamental drawback.

Example

Consider a task running a CPU-heavy function `do_work()` every 100 millisecods. Using `delay()`, the naive implementation would be:

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

Yet another problem

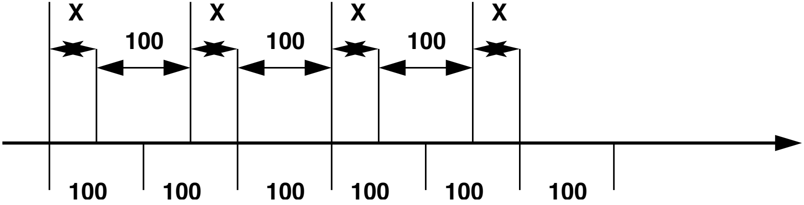
Even if other threads were not to interfere, using delay services as a means of specifying future points in time has another fundamental drawback.

Example

Consider a task running a CPU-heavy function `do_work()` every 100 millisecods. Using `delay()`, the naive implementation would be:

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

Accumulating drift

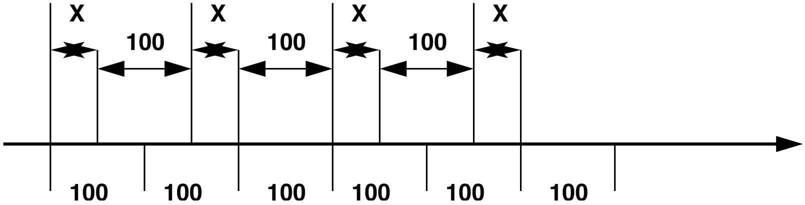


X is the time take to `do_work`

With relative delays, each turn in the loop will take at least 100+X milliseconds.

A drift of X milliseconds will accumulate every turn!

Accumulating drift

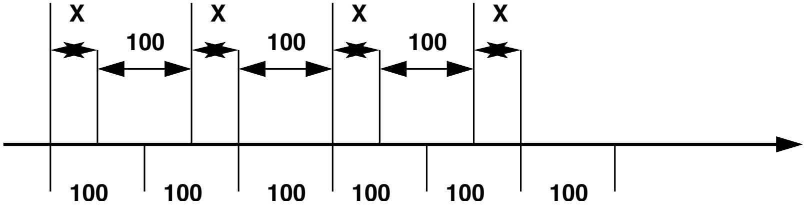


X is the time take to `do_work`

With relative delays, each turn in the loop will take at least 100+X milliseconds.

A drift of X milliseconds will accumulate every turn!

Accumulating drift

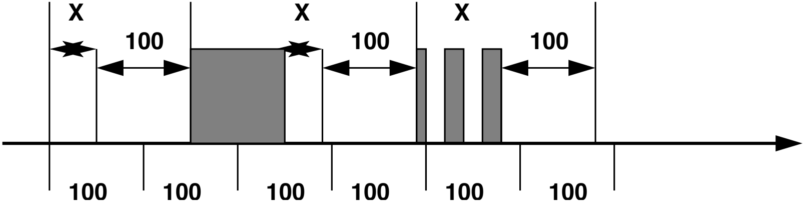


`X` is the time take to `do_work`

With relative delays, each turn in the loop will take at least 100+`X` milliseconds.

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

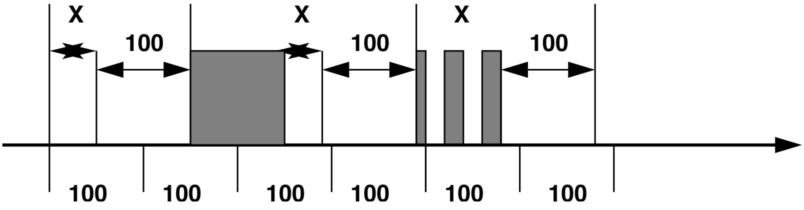
Accumulating drift



With other threads in the system, the already bad scenario gets worse!

That means that even if we knew X, we wouldn't be able to compensate the delay time in any predictable manner!

Accumulating drift



With other threads in the system, the already bad scenario gets worse!

That means that even if we knew X, we wouldn't be able to compensate the delay time in any predictable manner!

A stable reference

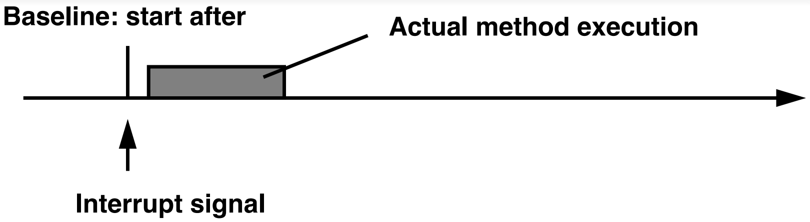
What we need is a stable time reference to use as a basis whenever we specify a relative time (instead of now).

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 timestamp:



A stable reference

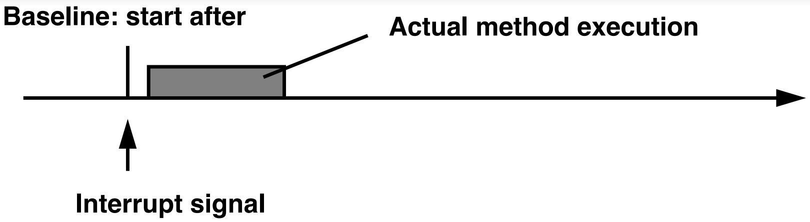
What we need is a stable time reference to use as a basis whenever we specify a relative time (instead of now).

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 timestamp:



A stable reference

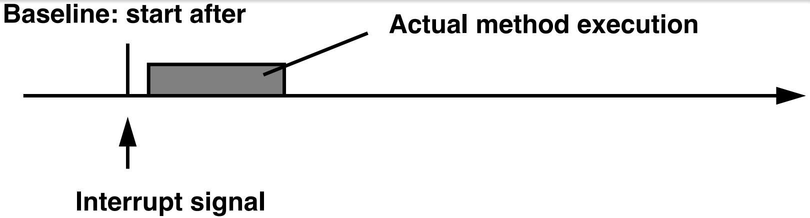
What we need is a stable time reference to use as a basis whenever we specify a relative time (instead of now).

Baselines

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

Time stamps of interrupts!

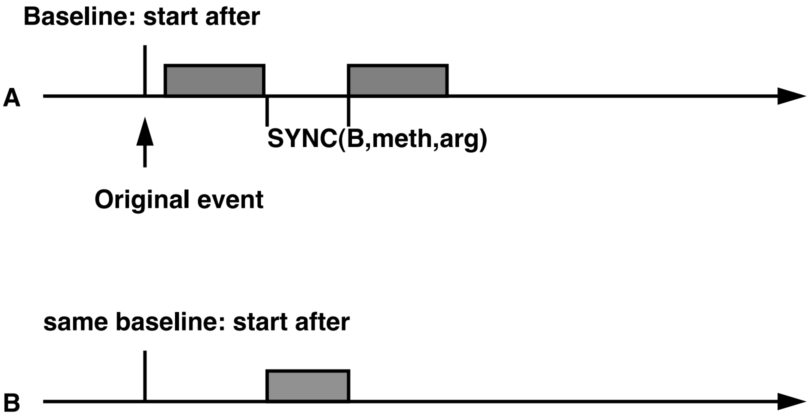
The baseline of an event is its timestamp:



A stable reference

SYNC

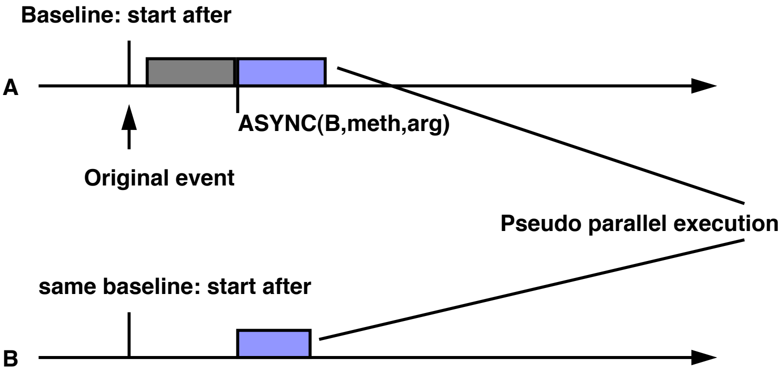
Calling methods with `SYNC` doesn't change the baseline (the call inherits the baseline)



A stable reference

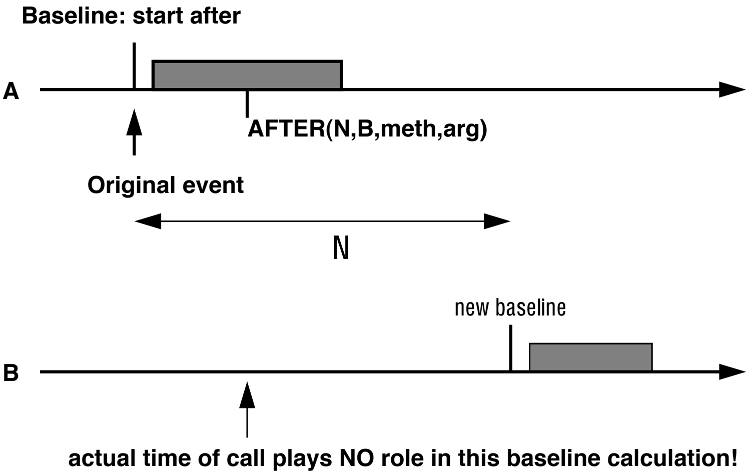
ASYNC

By default `ASYNC` method calls will inherit the baseline



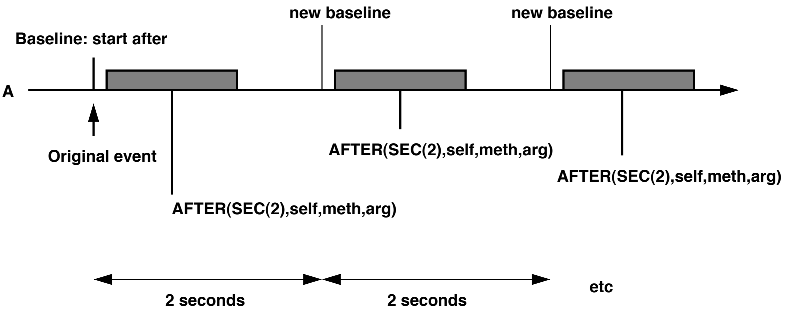
A stable reference

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



Periodic tasks

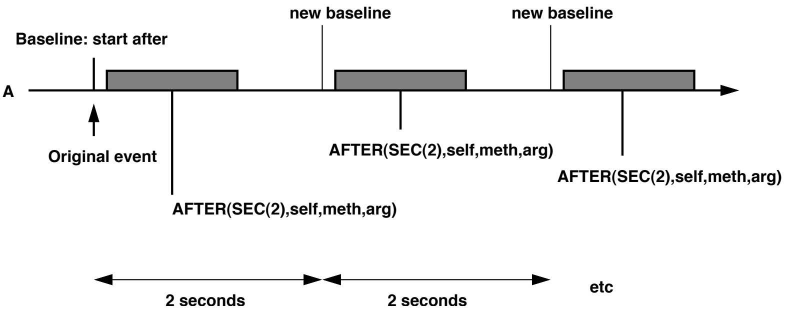
To create a cyclic reaction, simply call **self** with the same method and a new baseline:



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)
```