Overview

- Sampling the time
  - reading the clock
- Relative delays
  - time drift
- Time reference

Aspects of Real Time

- An external process to sample
  - program can read a real-time clock
    - just as it samples any external process value (e.g., the temperature)
- An external process to react to
  - A program can let certain points in time denote events
    - by means of interrupts by a clock
- An external process to be constrained by
  - A program might be required to "hurry" enough
    - some externally visible action can be performed before a certain point in time (lecture 9)

Sampling the time

- Requires a hardware clock that can be read like a regular external device
- Multitude of alternatives:
  - Units?
    - (seconds, milliseconds, cpu cycles, system "ticks", ...)
  - Since when?
    - (program start, system boot, Jan 1 1970, ...)
  - Real time?
    - (alternatives: time that stops when other threads are running, time that stops when the CPU sleeps, time that cannot be set and always increases, ...)

Timestamps

- In isolation, reading the clock only gives information about the time when the clock was read
  - E.g.: "the clock showed 20:32 when I read it"
- In general, clock readings only meaningful when associated with other events:
  - E.g.: "the clock showed 20:32 when the sun set"
- In a program
  - associating a clock reading with other event
    - reading in close proximity to the event detection
      - otherwise time would be false (too late)

Timer/Counter1 on the AVR

- 16-bit counter
  - accessible through register TCNT1
- Units:
  - CPU clock (8 MHz)
    - divided by programmable prescaling value (1, 8, 64, 256, 1024)
    - or external source
- Since when:
  - system reset, timer reset or timer overflow (whichever was last)
- Shows real time
  - (although it can be manually stopped)
- Note: aligning TCNT1 with standard calendar time will require some calculations when interpreting timer values and some extra storage for counting timer overflows
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
- But again we must ask what each timestamp is supposed to mean...
  - the time of some arbitrary program instruction?
  - the beginning or end of a certain function call?
  - the time of sending or receiving an asynchronous message?
- All these points in time are purely internal to a program under execution, and have no meaning to an external observer
- Moreover...

In a scheduled system...

- Even if clocks read just after event
  - long time might have passed
- Other threads executed
- Close proximity not equal to subsequent statements

Time-stamping events

- We want
  - associate timestamps with the externally observable events that “drive” a system!
- Idea: read the clock inside the interrupt handler that detects the associated event
  - Other interrupts are disabled while the CPU runs an interrupt handler, hence
  - no scheduling of other threads might interfere!
  - I.e.: there is a (tight) upper bound on the timestamp error, which can be calculated from CPU data & speed
- For example, one could use the “arg” of an interrupt-driven method for passing a timestamp

Example

- Calculate the speed of a rotating wheel by measuring the time between two subsequent detections of a passing tap

Real-time events

- now we know
  - how to sample the real-time clock
  - obtain externally meaningful information about the passage of time
- want to take some action
  - ...when a certain amount of time has passed
- example
  - suppose the wheel of the previous slide is an engine crankshaft
    - our duty to emit ignition signals to each cylinder...
  - We would need a way to postpone program execution until certain points in the future
    - we would need a way to react to real-time events!

The very poor man’s solution

- Consume a fixed amount of CPU cycles in a silly loop
  ```c
  int i;
  for ( i=0; i<N; i++)
    ; // wait...
  do_future_action();
  ```
- Determine a suitable N by testing
- Problem 1: N will be highly platform dependent
- Problem 2: Just like with busy-waiting, a lot of CPU cycles will simply be wasted
The nearly as poor man’s solution

- Configure your timer/counter with a known clock speed, and busy-wait for a suitable time increment
  
  ```c
  unsigned int i = TCNT1 + N;
  while (TCNT1 < i)
    ; // wait...
  do_future_action();
  ```

- Determine N by calculation
- Problem 1 disappears; no platform dependency
- Problem 2 remains, though; a lot of wasted CPU cycles

The standard solution

- Use an Operating System call
  - “fakes” the timer increment busy-wait loop
    - making better use of the CPU resources
      - (recall the hybrid approach to event handling in general)
  - No platform dependencies
  - No wasted CPU cycles
    - at the expense of complex OS internals
  - However, all these solutions have another problem in common...

In a scheduled system...

- Had we only known about the scheduler’s choice, a smaller N could have been used...

Relative delays

- problem
  - delay services are always specified using relative time:
  - constructed real-time event
    - occur at a time obtained by adding the delay parameter N to “now”
  - “now” not 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 (argument in seconds), usleep (microseconds), nanosleep (nanoseconds). The delay call usually refers to milliseconds.
    - [And we’re not going to rely on OS services anyway...]

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
- Consider the task of running a CPU-heavy function `do_work()` every 100 milliseconds
  - Using delay(), the naive implementation would be:
    ```c
    while (!)
      do_work();
      delay(100);
    ```
  - See any problems?

Accumulating drift

- With relative delays, each turn in the loop will take at lest 100+X milliseconds, where X is the time taken to perform `do_work()`.
  - Thus, a drift of X milliseconds will accumulate every turn!
Accumulating drift

- With other threads in the system
  - the already bad scenario just 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 really need is a stable time reference to use as our basis whenever we specify relative time
- Fortunately, the idea of time-stamping every interrupt suits us well in this respect
- Let us introduce the baseline of a message – the earliest time a message is allowed to start
- Initially, the baseline of an event is its timestamp:

A stable reference

- SYNC method calls will always inherit the baseline of the caller
- ASYNC calls will inherit the baseline of the caller by default
- However, for ASYNC calls we may also consider adding a baseline offset \( N! \)
- But independence from current time requires that the new baseline is somewhere “in the future”. Otherwise:

A stable reference

- \( \text{SYNC}(B, \text{meth}, \text{arg}) \)
- \( \text{ASYNC}(B, \text{meth}, \text{arg}) \)
- \( \text{AFTER}(N, B, \text{meth}, \text{arg}) \)

I.e., calculated baseline is adjusted upwards if necessary, can’t be < “now”
A stable reference

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

```
AFTER(SEC(2), B, meth, arg)
```

```
2s
```

Convenient macro that makes the call independent of current timer resolution

Implementing AFTER

• First let the baseline be stored with every message (as part of the Msg struct)
• AFTER is same as ASYNC, but
  - New baseline is MAX(now, offset + current->baseline)
  - If baseline > now, put message in a timerQ instead of the readyQ
  - Set up timer to generate interrupt after earliest baseline
  - At each timer interrupt, move first timerQ message to readyQ and configure a new timer interrupt
• To inherit the current baseline, use offset -1. In fact:
  - #define ASYNC(to,meth,arg) AFTER(-1,to,meth,arg)

Accessing the baseline

• Because an ASYNC call from an interrupt automatically sets the baseline to "now", we can use this value for a timestamp.

```
typedef struct {
  Object super;
  int previous;
} Speedo;

Speedo speedo;
... INT(SIG_XX, ASYNC(&speedo, detect, 0) );
```

```
interrupt(SIG_XX, ASYNC(&speedo, detect, 0) );
```

```
INTERRUPT(SIG_XX, ASYNC(&speedo, detect, 0) );
```

Next lecture

• Deadlines and priorities!