NAME

"IO::Async::Timer::Periodic" – event callback at regular intervals

SYNOPSIS

```
use IO::Async::Timer::Periodic;

use IO::Async::Loop;
my $loop = IO::Async::Loop->new;

my $timer = IO::Async::Timer::Periodic->new(
   interval => 60,

   on_tick => sub {
      print "You've had a minute\n";
   },
);

$timer->start;

$loop->add( $timer );

$loop->run;
```

DESCRIPTION

This subclass of IO::Async::Timer implements repeating events at regular clock intervals. The timing may or may not be subject to how long it takes the callback to execute. Iterations may be rescheduled runs at fixed regular intervals beginning at the time the timer was started, or by a fixed delay after the previous code has finished executing.

For a `Timer` object that only runs a callback once, after a given delay, see instead IO::Async::Timer::Countdown. A Countdown timer can also be used to create repeating events that fire at a fixed delay after the previous event has finished processing. See als the examples in `IO::Async::Timer::Countdown`.

EVENTS

The following events are invoked, either using subclass methods or CODE references in parameters:

on_tick

Invoked on each interval of the timer.

PARAMETERS

The following named parameters may be passed to `new` or `configure`:

on_tick => CODE

CODE reference for the `on_tick` event.

interval => NUM

The interval in seconds between invocations of the callback or method. Cannot be changed if the timer is running.

first_interval => NUM

Optional. If defined, the interval in seconds after calling the `start` method before the first invocation of the callback or method. Thereafter, the regular `interval` will be used. If not supplied, the first interval will be the same as the others.

Even if this value is zero, the first invocation will be made asynchronously, by the containing `Loop` object, and not synchronously by the `start` method itself.

reschedule => STRING

Optional. Must be one of `hard`, `skip` or `drift`. Defines the algorithm used to reschedule the next invocation.

`hard` schedules each iteration at the fixed interval from the previous iteration's schedule time, ensuring a regular repeating event.

`skip` schedules similarly to `hard`, but skips over times that have already passed. This matters if the duration is particularly short and there's a possibility that times may be missed, or if the entire process is stopped and resumed by `SIGSTOP` or similar.

`drift` schedules each iteration at the fixed interval from the time that the previous iteration's event handler returns. This allows it to slowly drift over time and become desynchronised with other events of the same interval or multiples/fractions of it.

Once constructed, the timer object will need to be added to the `Loop` before it will work. It will also need to be started by the `start` method.

AUTHOR

Paul Evans <leonerd@leonerd.org.uk>