NAME

IPC::Run::Timer −− Timer channels for IPC::Run.

SYNOPSIS

```
   use IPC::Run qw( run  timer timeout );
   ## or IPC::Run::Timer ( timer timeout );
   ## or IPC::Run::Timer ( :all );

   ## A non-fatal timer:
   $t = timer( 5 ); # or...
   $t = IO::Run::Timer->new( 5 );
   run $t, ...;

   ## A timeout (which is a timer that dies on expiry):
   $t = timeout( 5 ); # or...
   $t = IO::Run::Timer->new( 5, exception => "harness timed out" );
```

DESCRIPTION

This class and module allows timers and timeouts to be created for use by IPC::Run. A timer simply expires when it's time is up. A timeout is a timer that throws an exception when it expires.

Timeouts are usually a bit simpler to use than timers: they throw an exception on expiration so you don't need to check them:

```
   ## Give @cmd 10 seconds to get started, then 5 seconds to respond
   my $t = timeout( 10 );
   $h = start(
      \@cmd, \$in, \$out,
      $t,
   );
   pump $h until $out =~ /prompt/;

   $in = "some stimulus";
   $out = '';
   $t->time( 5 )
   pump $h until $out =~ /expected response/;
```

You do need to check timers:

```
   ## Give @cmd 10 seconds to get started, then 5 seconds to respond
   my $t = timer( 10 );
   $h = start(
      \@cmd, \$in, \$out,
      $t,
   );
   pump $h until $t->is_expired || $out =~ /prompt/;

   $in = "some stimulus";
   $out = '';
   $t->time( 5 )
   pump $h until $out =~ /expected response/ || $t->is_expired;
```

Timers and timeouts that are reset get started by *start()* and *pump()*. Timers change state only in *pump()*. Since *run()* and *finish()* both call *pump()*, they act like *pump()* with respect to timers.

Timers and timeouts have three states: reset, running, and expired. Setting the timeout value resets the timer, as does calling the *reset()* method. The *start()* method starts (or restarts) a timer with the most recently set time value, no matter what state it's in.

Time values

All time values are in seconds. Times may be any kind of perl number, e.g. as integer or floating point seconds, optionally preceded by punctuation-separated days, hours, and minutes.

Examples:

```
   1           1 second
   1.1         1.1 seconds
   60          60 seconds
   1:0         1 minute
   1:1         1 minute, 1 second
   1:90        2 minutes, 30 seconds
   1:2:3:4.5   1 day, 2 hours, 3 minutes, 4.5 seconds
   'inf'       the infinity perl special number (the timer never finishes)
```

Absolute date/time strings are *not* accepted: year, month and day-of-month parsing is not available (patches welcome :–).

Interval fudging

When calculating an end time from a start time and an interval, IPC::Run::Timer instances add a little fudge factor. This is to ensure that no time will expire before the interval is up.

First a little background. Time is sampled in discrete increments. We'll call the exact moment that the reported time increments from one interval to the next a tick, and the interval between ticks as the time period. Here's a diagram of three ticks and the periods between them:

```
    -0-0-0-0-0-0-0-0-0-0-1-1-1-1-1-1-1-1-1-1-2-...
    ^                    ^                    ^
    |<--- period 0 ---->|<--- period 1 ---->|
    |                    |                    |
  tick 0               tick 1               tick 2
```

To see why the fudge factor is necessary, consider what would happen when a timer with an interval of 1 second is started right at the end of period 0:

```
    -0-0-0-0-0-0-0-0-0-0-1-1-1-1-1-1-1-1-1-1-2-...
    ^                  ^  ^                    ^
    |                  |  |                    |
    |                  |  |                    |
  tick 0               |tick 1               tick 2
                       |
                  start $t
```

Assuming that *check()* is called many times per period, then the timer is likely to expire just after tick 1, since the time reported will have lept from the value '0' to the value '1':

```
    -0-0-0-0-0-0-0-0-0-0-1-1-1-1-1-1-1-1-1-1-2-...
    ^                  ^  ^    ^               ^
    |                  |  |    |               |
    |                  |  |    |               |
  tick 0               |tick 1|              tick 2
                       |       |
                  start $t    |
                               |
                         check $t
```

Adding a fudge of '1' in this example means that the timer is guaranteed not to expire before tick 2.

The fudge is not added to an interval of '0'.

This means that intervals guarantee a minimum interval. Given that the process running perl may be suspended for some period of time, or that it gets busy doing something time-consuming, there are no other guarantees on how long it will take a timer to expire.

SUBCLASSING

INCOMPATIBLE CHANGE: Due to the awkwardness introduced by ripping pseudohashes out of Perl, this class *no longer* uses the fields pragma.

FUNCTIONS & METHODS

timer

A constructor function (not method) of IPC::Run::Timer instances:

```
$t = timer( 5 );
$t = timer( 5, name => 'stall timer', debug => 1 );

$t = timer;
$t->interval( 5 );

run ..., $t;
run ..., $t = timer( 5 );
```

This convenience function is a shortened spelling of

```
IPC::Run::Timer->new( ... );
```

. It returns a timer in the reset state with a given interval.

If an exception is provided, it will be thrown when the timer notices that it has expired (in *check()*). The name is for debugging usage, if you plan on having multiple timers around. If no name is provided, a name like "timer #1" will be provided.

timeout

A constructor function (not method) of IPC::Run::Timer instances:

```
$t = timeout( 5 );
$t = timeout( 5, exception => "kablooey" );
$t = timeout( 5, name => "stall", exception => "kablooey" );

$t = timeout;
$t->interval( 5 );

run ..., $t;
run ..., $t = timeout( 5 );
```

A This convenience function is a shortened spelling of

```
IPC::Run::Timer->new( exception => "IPC::Run: timeout ...", ... );
```

. It returns a timer in the reset state that will throw an exception when it expires.

Takes the same parameters as "timer", any exception passed in overrides the default exception.

new

```
IPC::Run::Timer->new()   ;
IPC::Run::Timer->new( 5 )   ;
IPC::Run::Timer->new( 5, exception => 'kablooey' )   ;
```

Constructor. See "timer" for details.

check

```
check $t;
check $t, $now;
$t->check;
```

Checks to see if a timer has expired since the last check. Has no effect on non-running timers. This will throw an exception if one is defined.

IPC::Run::pump() calls this routine for any timers in the harness.

You may pass in a version of now, which is useful in case you have it lying around or you want to check several timers with a consistent concept of the current time.

Returns the time left before end_time or 0 if end_time is no longer in the future or the timer is not running (unless, of course, *check() expire()*s the timer and this results in an exception being thrown).

Returns undef if the timer is not running on entry, 0 if *check()* expires it, and the time left if it's left running.

debug
: Sets/gets the current setting of the debugging flag for this timer. This has no effect if debugging is not enabled for the current harness.

end_time

```
    $et = $t->end_time;
    $et = end_time $t;

    $t->end_time( time + 10 );
```

Returns the time when this timer will or did expire. Even if this time is in the past, the timer may not be expired, since *check()* may not have been called yet.

Note that this end_time is not start_time($t) + interval($t), since some small extra amount of time is added to make sure that the timer does not expire before *interval()* elapses. If this were not so, then

Changing *end_time()* while a timer is running will set the expiration time. Changing it while it is expired has no affect, since *reset()*ing a timer always clears the *end_time()*.

exception

```
    $x = $t->exception;
    $t->exception( $x );
    $t->exception( undef );
```

Sets/gets the exception to throw, if any. 'undef' means that no exception will be thrown. Exception does not need to be a scalar: you may ask that references be thrown.

interval

```
    $i = interval $t;
    $i = $t->interval;
    $t->interval( $i );
```

Sets the interval. Sets the end time based on the *start_time()* and the interval (and a little fudge) if the timer is running.

expire

```
    expire $t;
    $t->expire;
```

Sets the state to expired (undef). Will throw an exception if one is defined and the timer was not already expired. You can expire a reset timer without starting it.

is_running
is_reset
is_expired
name
: Sets/gets this timer's name. The name is only used for debugging purposes so you can tell which freakin' timer is doing what.

reset

```
    reset $t;
    $t->reset;
```

Resets the timer to the non-running, non-expired state and clears the *end_time()*.

start

```
    start $t;
    $t->start;
    start $t, $interval;
    start $t, $interval, $now;
```

Starts or restarts a timer. This always sets the start_time. It sets the end_time based on the interval if the timer is running or if no end time has been set.

You may pass an optional interval or current time value.

Not passing a defined interval causes the previous interval setting to be re-used unless the timer is reset and an end_time has been set (an exception is thrown if no interval has been set).

Not passing a defined current time value causes the current time to be used.

Passing a current time value is useful if you happen to have a time value lying around or if you want to make sure that several timers are started with the same concept of start time. You might even need to lie to an IPC::Run::Timer, occasionally.

start_time
Sets/gets the start time, in seconds since the epoch. Setting this manually is a bad idea, it's better to call "start"() at the correct time.

state

```
    $s = state $t;
    $t->state( $s );
```

Get/Set the current state. Only use this if you really need to transfer the state to/from some variable. Use "expire", "start", "reset", "is_expired", "is_running", "is_reset".

Note: Setting the state to 'undef' to expire a timer will not throw an exception.

TODO

use Time::HiRes; if it's present.

Add detection and parsing of [[[HH:]MM:]SS formatted times and intervals.

AUTHOR

Barrie Slaymaker <barries@slaysys.com>