Event driven programming in Perl using the Event module

Table of contents

1 Event driven programming in Perl using the Event module

Jochen Stenzel (perl@jochen−stenzel.de)22 February 2000

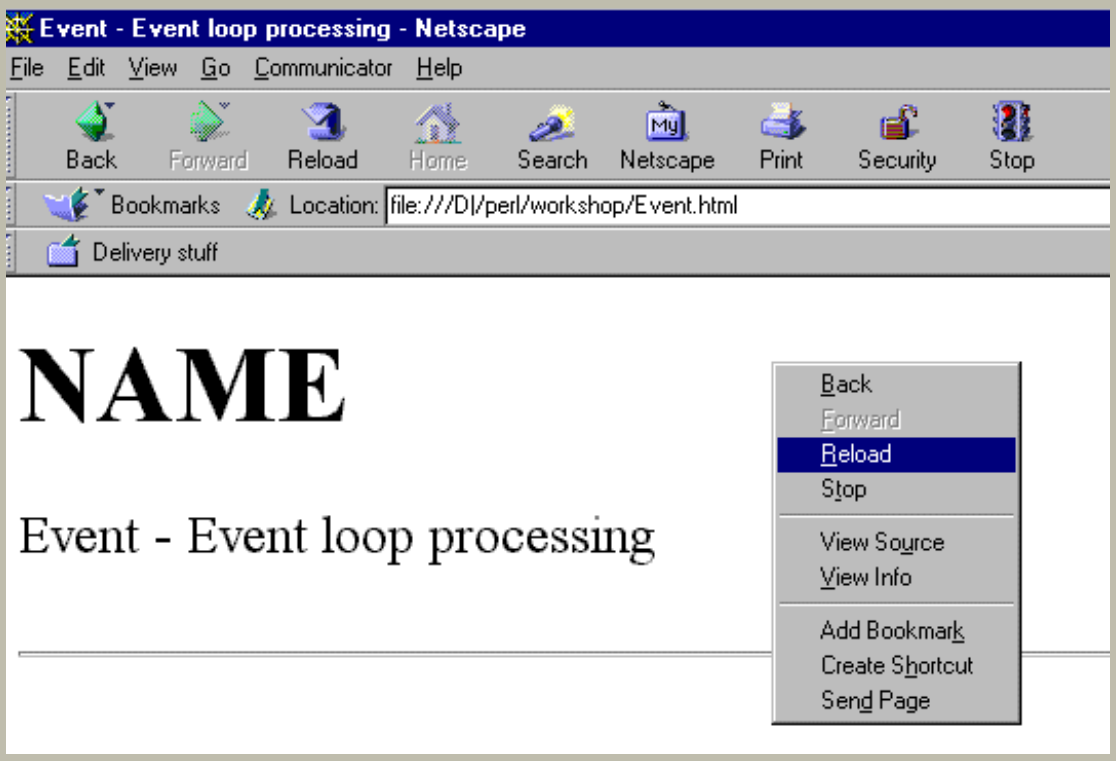
1.1 Table of Contents

---

2 1. Introduction

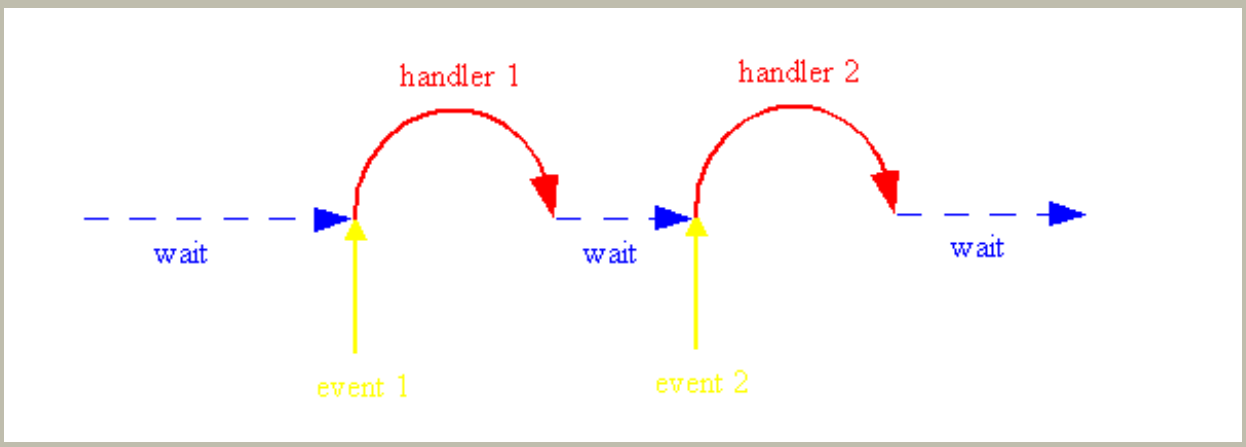
2.1 1.1. The task

Everyone of us knows programs like this

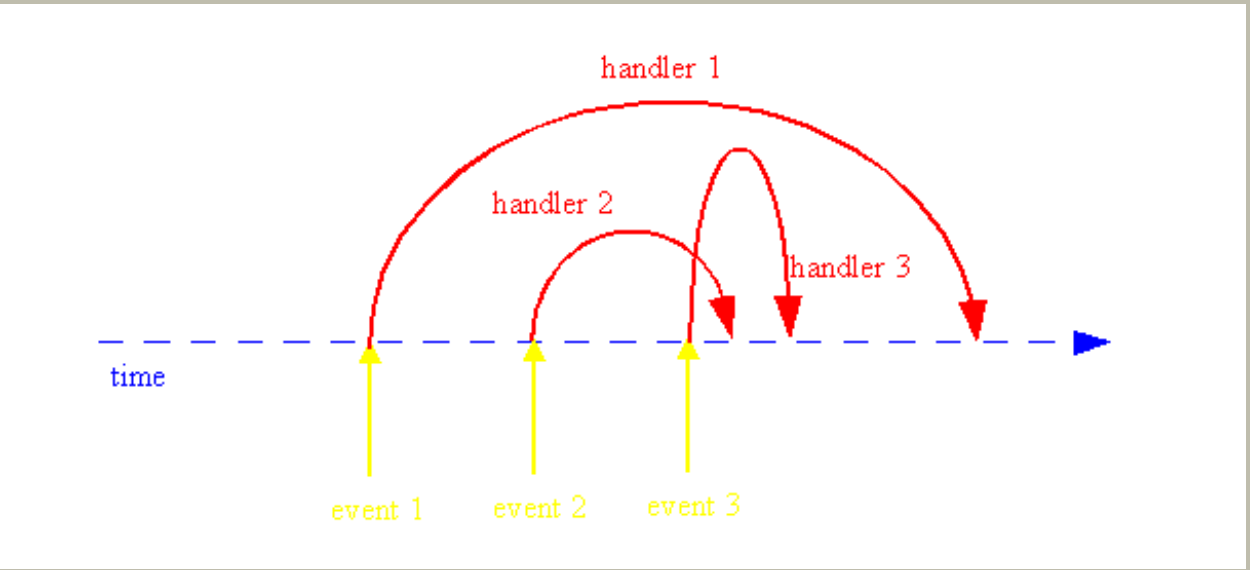


A click somewhere, and something happens. Another click somewhere else, and (usually) anything else happens. If my application is a web browser receiving a site from a server, the browser is able to take this site while it is still usable. Hopefully. So, if I prefer, I can stop the download even before it is completed by just clicking a button or choosing a menu option.

Clicks, server messages and menu choices are **events**. The principle of such a program is to work on base of events. And regardless of time and frequency of events, in every case the appropriate handler function is called to serve it. This could be illustrated like this:

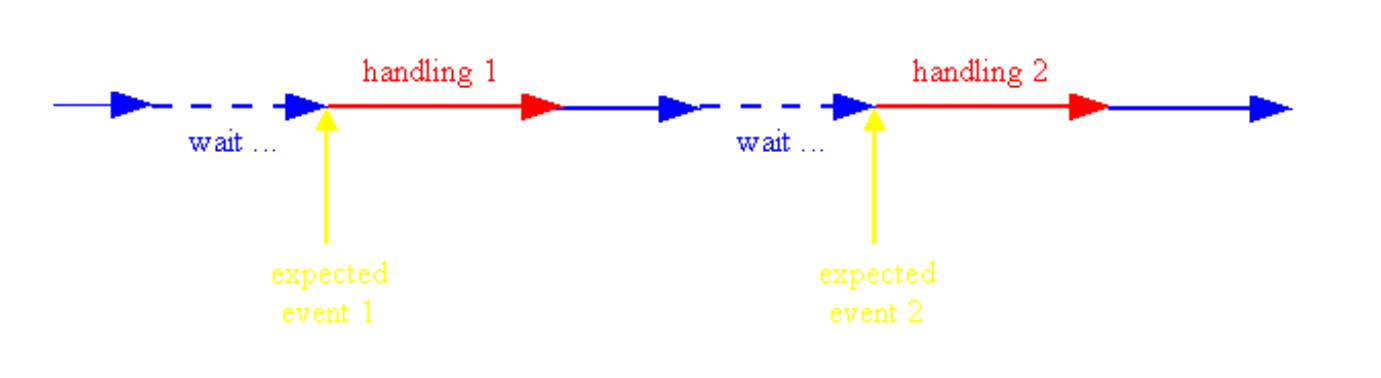


Various graphical interfaces do even more than that: they work **asynchronously** as well. This means that a second event can be detected while the first event is served, a user has not to wait until the first result arrives. It also means that the results of earlier events can be provided before the results of later events.




```
If I request a web page and use the same application
to check my mailbox while the page is loaded, mails
may arrive before the page.
```

Contrary to this, programs without a graphical interface usually work **synchronously** and can handle events only if they are expected:



The first event will be served first, and the second subsequently. (Well, signal handlers are an exception here both in Perl and C − in a limited matter.)

```
A usual shell (without job control) cannot handle new
commands while it is still processing another. But while
it offers a prompt, it cannot do anything until the user
will have been entered a new command.
```

But there is no need to restrict the event driven, asynchronous architecture to graphical interfaces. (Even if it is really hard to imagine a synchronous GUI.)

```
An asynchronous shell would provide a new prompt
while it processes a command, so the user could already
enter new commands. The shell would process all entered
commands simultaneously and would offer the first
available result first, regardless of the command
order.
```

Event driven architectures are really worth a thought everytime a program has several handling lines which do not strongly depend one from another, if these subtasks need certain start impulses and if they could be finished (completely or in parts) quick enough to avoid mutual blocking.

```
Independend tasks could be

  − background calculations;
```

```
- preparation of data to provide them quickly on request;
- displaying process progress while the process still progresses;
- accepting commands;
- receiving data from other processes;
- supervision of several IPC connections (e.g. IRC);
- signal processing;
- date reminding;
- ...
```

2.2 1.2. Implementations of asynchronous programs

There are several ways to build asynchronous programs. Multiprocess systems on base of `fork()` are very popular (think of the usual servers). *Threads* are a light weight variant of them. And, finally, *event driven programming* is another way. This term, from now on, is used in this document for systems handling occuring events on base of a *loop*. Each of these systems has certain advantages and disadvantages:

method	***expense***	***data sharing***	***parallelism***	***remarks***
fork()	relatively high (depending on the operating system)	difficult	(theoretically) real	the system limits the number of processes
Threads	relatively low	(sometimes enormous) synchronization overhead	(theoretically) real	not really usable in perl 5.005
event driven programming	low	simple	serialization	long running tasks have to be split up or delegated

With all these methods, the solution has to be split up into several parts.

Note: The mentioned methods do not mutually exclude each other in a program. Sometimes it can be useful to combine them to get the greatest possible advantage (see the callback section for examples). But here I first want to point out ways to implement the base algorithm.

Event driven programming turns out to be real alternative here!

2.3 1.3. Event handling with Perl

A simple form of event handling is provided by `%SIG`. This interface is simple indeed, easy to use − and limited. It can handle signals only, for example, and there is no loop.

But Perl provides a real base function to implement event driven programs: `select()` lets the system take control until one of certain user defined events occurs. This can be made the base of a loop. Additionally, an optional timeout can be set to reactivate the program if nothing else happens. Events are described by vectors which make the interface more sophisticated, but the **IO::Select** module simplifies both usage and readability as a wrapper.

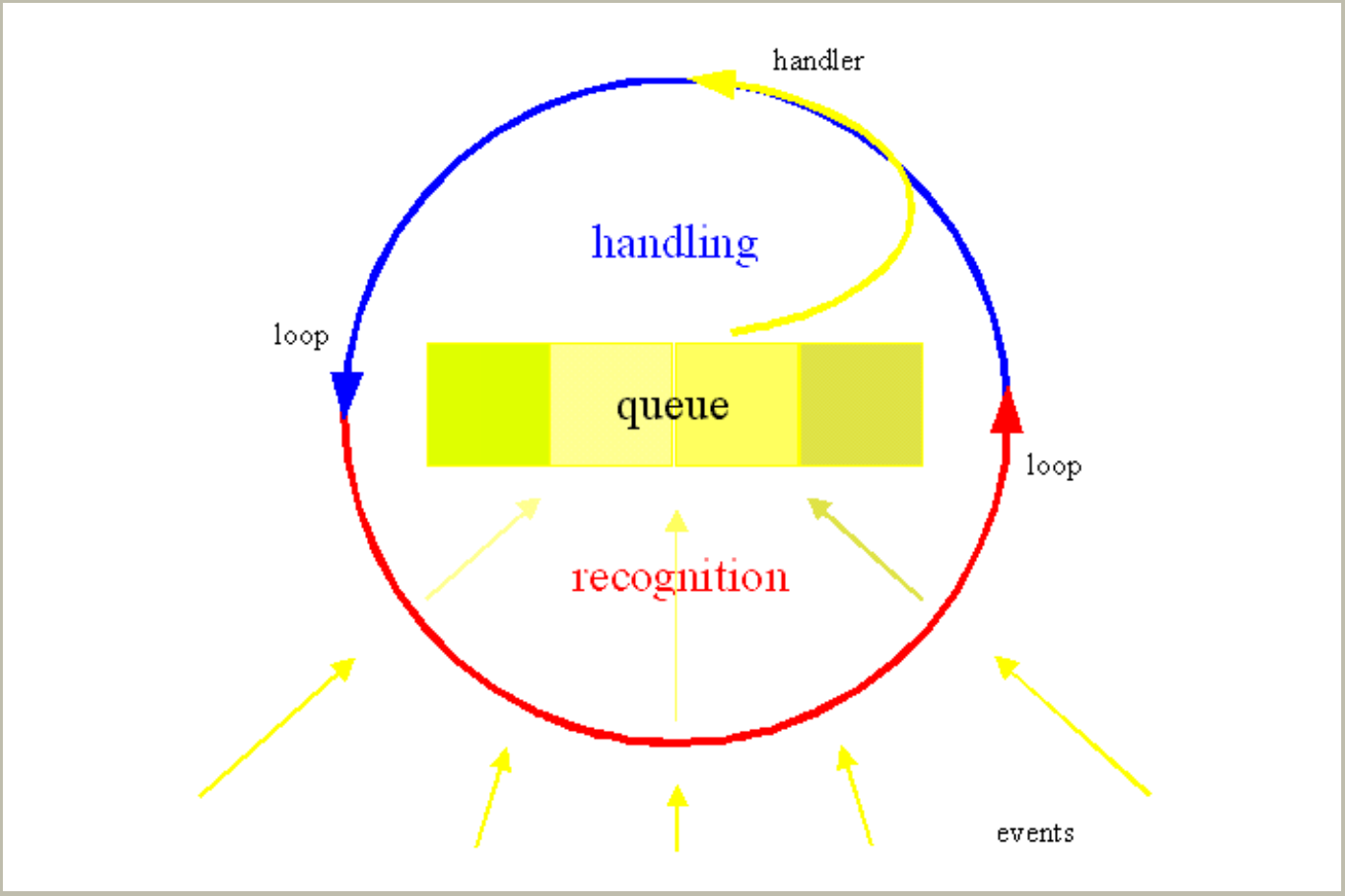
Well, on the other hand, `select()` and **IO::Select** are *restricted* to *exactly one timeout* and events happening on *handles*. Other types of events are not covered. Changing and maintaining event masks is *not* easy. While it *is* possible to build event loops basing on `select()` / **IO::Select**, building them a flexible way could become a challenge.

But there is no need for such effort: **Event by J. N. Pritikin (CPAN−ID JPRIT)** provides a powerful, flexible, scalable and fast event loop with a relatively easy interface, designed for various event types. It gives you the chance to build event driven scripts in minutes. Besides this, **Event** code is easy to read.

---

3 2. An overview of the Event module

In event driven process models, all essential things happen in exactly *one* process. To manage this, they install an *event loop* − a base function which embeds, controls and serializes anything else. While the loop is running, two tasks have to be performed again and again: events have to be *recognized* and associated functions have to be called to *handle* them.



As usual, **Event** implements the serialization of handler calls by a *queue*. To *detect* events, the module uses special objects called *watchers*.

3.1 2.1. The watcher concept

Event works an object oriented way. Its main actors are so called *watchers* which expect the happening of certain events and are prepared to initiate appropriate answers.

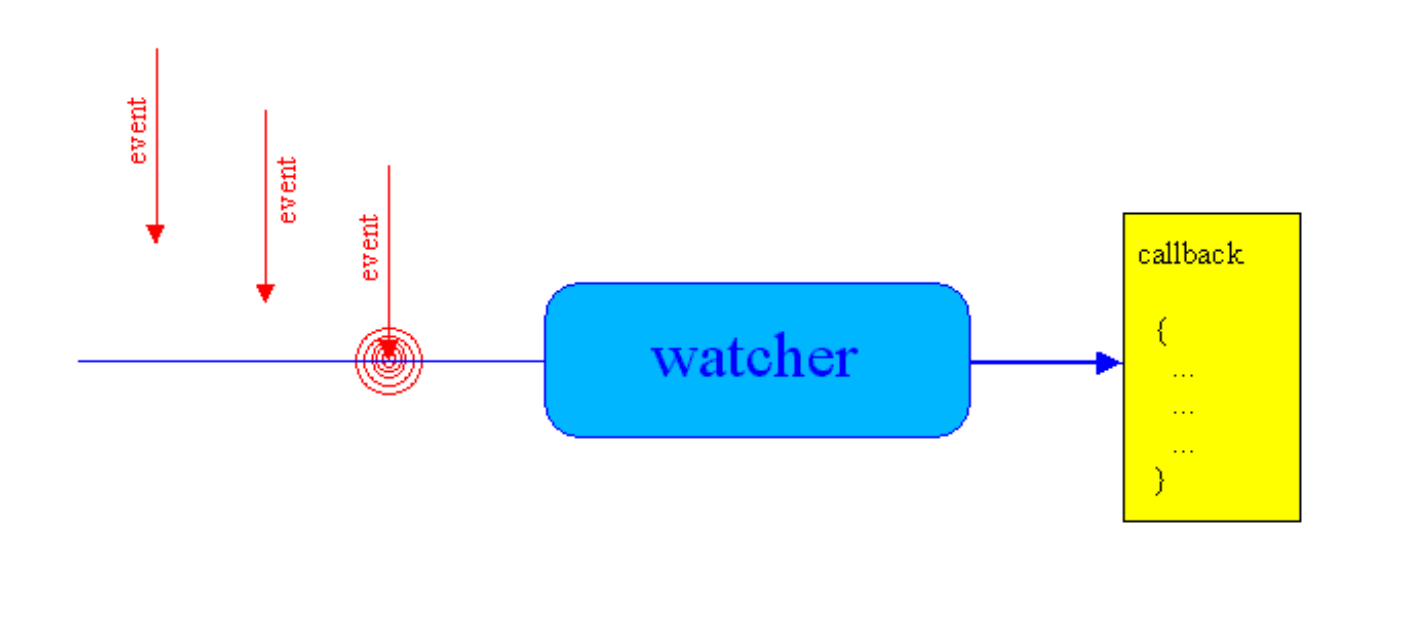
These watchers are very specialized. Each of them is responsible to detect events of a certain type. Some look for *IO* events, others for *timers*, others detect *signals* and others watch *variables*. There is even a group of watchers trained to recognize "nothing" − they get alarmed if the program is idle. And finally, a group of special agents shadows *other watchers*.

events	***watcher***
I/O	**io**
timer	**timer**
signals	**signal**
nothing else happened	**idle**

variable access	**var**
other watchers acting	**group**

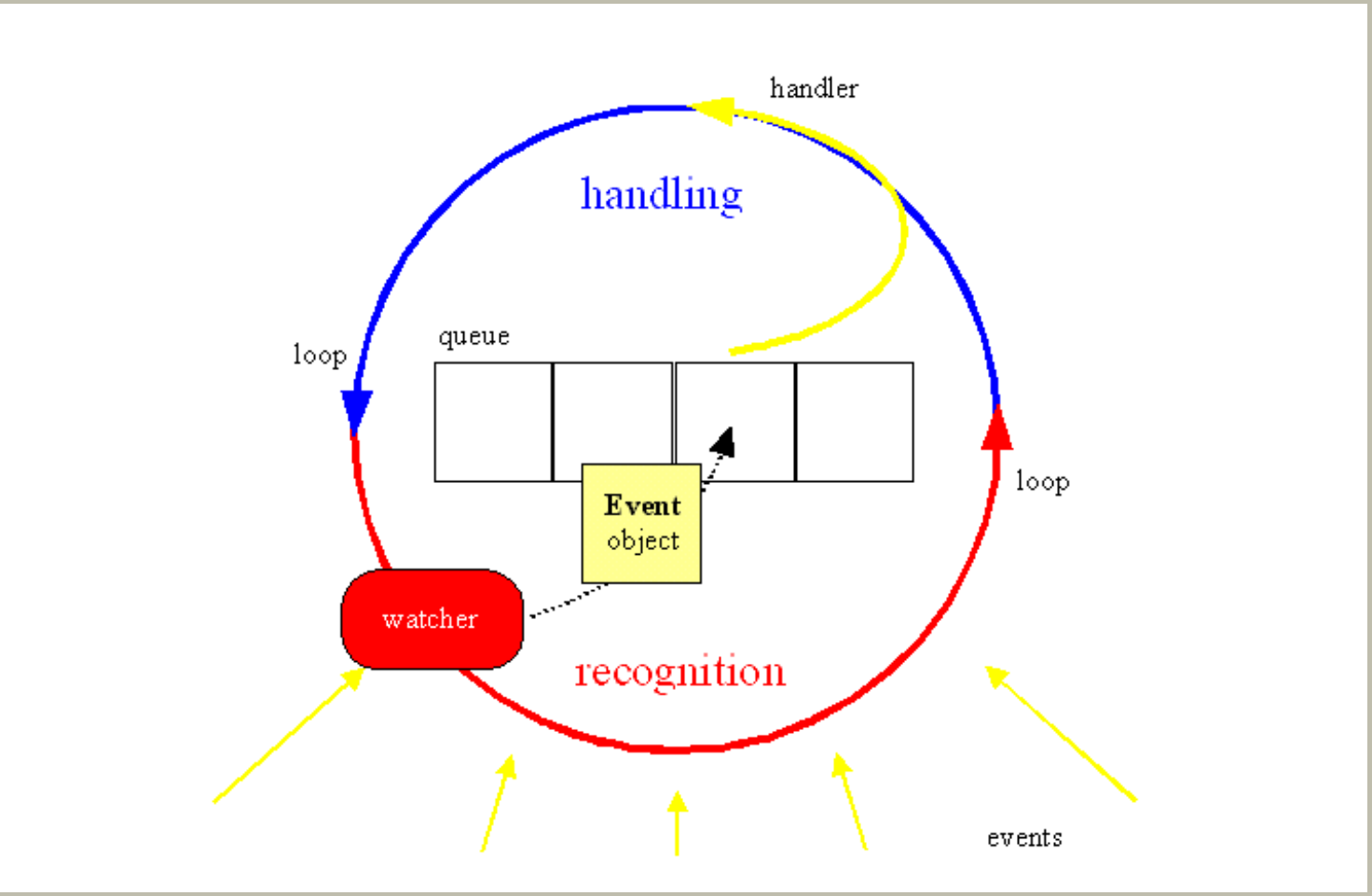
And there are more types to come, e.g. to observe *semaphores*.

Well, technically spoken, all these various watchers of course are objects of several classes derived from **Event** (**Event::io**, **Event::timer** etc.). They connect certain *events* and certain *functions*.

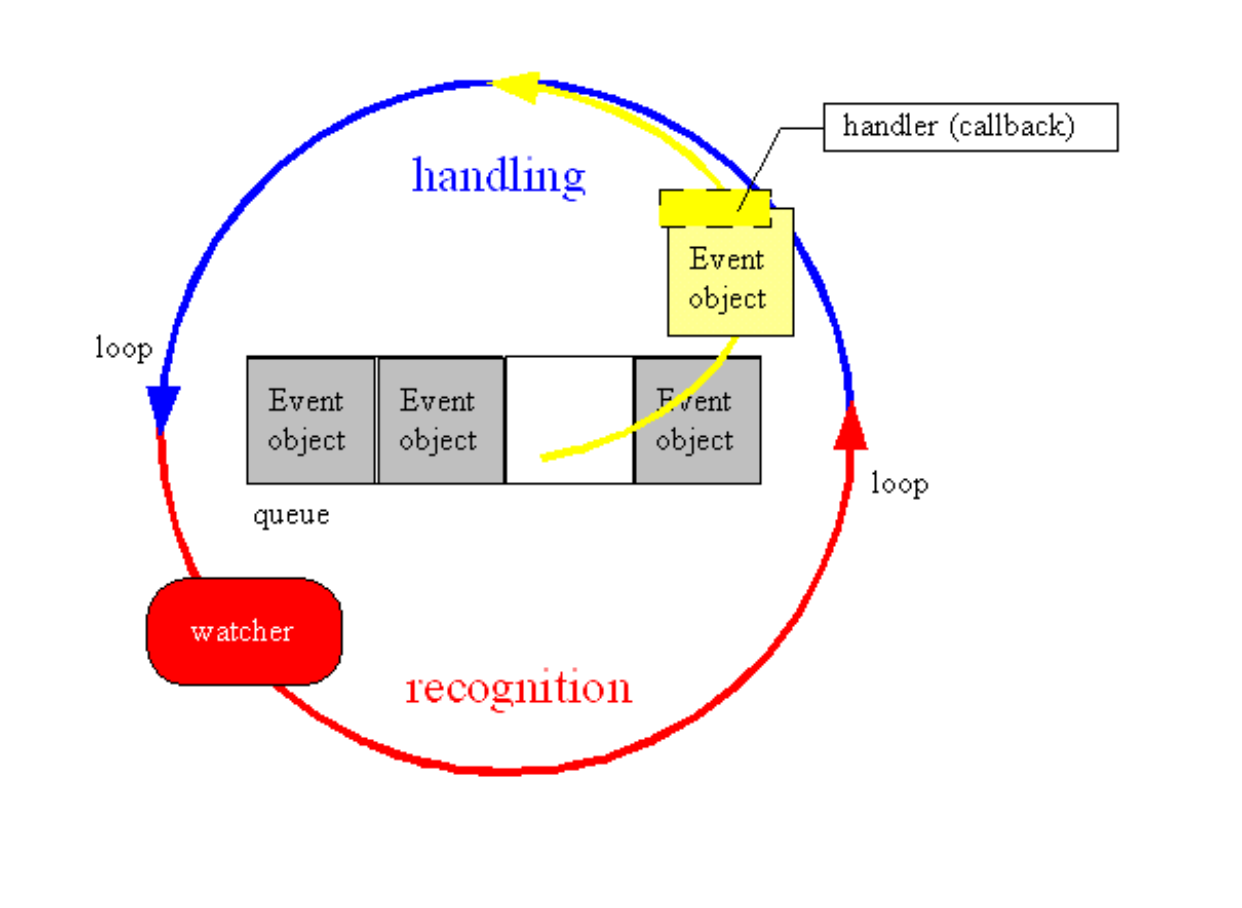


As soon as a watcher detects an occurence of its target event it initiates the call of its handler. Well, in principle.

The *real* process is more complex: in order to ensure the teamwork of all watchers, **Event** uses intermediate steps to perform the call of an event handler. If an event is recognized, a watcher generates a *new object* of a special `Event` subclass (`Event::Event` or `Event::Event::Io`, respectively) and stores it in the *queue*. (To avoid confusion, I will use `Event::Event` only from now on to name that class.) This object represents an order and contains informations about the detected event, the handler to call and the detecting watcher. By doing so, the watchers event handling is done, and it continues immediately to watch for events again.



The generated `Event::Event` object, on the other hand, remains in the queue until it is processed in the loops handling phase. The loop then calls the handler function referenced in the object.



After the order is carried out completely the loop destroys the order object.

An order object in the queue is no longer influenced by its "parent" watcher. It only contains a reference to it. That's why at a given time a number of `Event::Event` objects may be stored in the queue which are all made by the same watcher. Every watcher provides a method `pending()` which supplies its still queued orders in a list context:

```
# get the still pending orders
@pendingOrders=$watcher->pending;
# How many still unhandled tasks did the watcher produce?
print $watcher->desc, ": ", scalar(@pendingOrders), " tasks\n";
```

In a *scalar* context, `pending()` supplies a true value if such orders still exist.

To sum things up at this point, three important parts of the **Event** model became already visible: *watchers* to recognize events, *callbacks* (stored in queued `Event::Event` objects) to handle them and the *loop* which controls everything using a queue. And so, watchers, loop and callbacks are the base elements of **Event** programming.

3.2 2.2. Making watchers

An **Event** loop without active watchers would do nothing, and that's why such a loop terminates itself immediately. So, to avoid this, at least one watcher is installed usually before the loop starts:

```
# install an io watcher to check STDIN and
# initiate callback() calls if anything happens
Event->io(fd=>\*STDIN, cb=>\&callback);
```

Watcher constructors are named according to their type. In this example, an I/O watcher was built. Watchers of the other types could be installed by similar constructor calls (`Event->timer()`, `Event->var()` usf.).

The behaviour of a watcher is controlled by its *attributes*. All the parameters passed to an constructor call are simply attribute settings configuring the watcher properties. Usually, only a subset of available attributes is used explicitly in

a watchers constructor call, the remaining attributes are set by default. In the example above, the made I/O watcher should call a certain function `callback()` if something happens at handle `STDIN`.

There is no limit in the number of watchers, you can build as many as you need.

And here comes another example, installing a timer:

```
# install a pizza alarm facility
Event->timer(
             repeat   => 0,
             interval => 300,
             cb       => sub {warn "Look at the pizza!"},
            );
```

All these shown example watchers become active immediately after the constructor call. This means that they detect events. But, to give them a chance to *handle* these events, you have to establish the loop.

3.3 2.3. Starting the loop

If there is at least one active watcher, the loop can be started. This is done by the class method `loop()`:

```
Event::loop;
```

And that's all! Your script is running event driven now. Linear and synchronous program flow is left behind. If there are statements after the `loop()` call, they are delayed until the loop processing will be stopped.

As long as the loop runs, the program is controlled by the installed watchers, their callbacks and, of course, by occuring events.

A running loop can be stopped by the class method `unloop()`:

```
Event::unloop();
```

This method stops the running loop *without* effect to the installed watchers. This means that you could restart the loop later on by a new call of `loop()` and it would run as before.

Obviously, `unloop()` calls has to be implemented in watchers callbacks.

3.4 2.4. A complete example

The following example demonstrates all base elements of **Event** programming together.

```
# set pragma
use strict;

# load module
use Event qw(loop unloop);

# install initial watcher
Event->io(fd=>\*STDIN, poll=>'r', cb=>\&io);

# start loop
loop;


# FUNCTIONS ###################################################

# io handler
sub io
 {
  # read line
  my $cmd=<STDIN>;
  chomp $cmd;
```

```
  # stop processing, if necessary
  unloop, return if uc($cmd) eq 'QUIT';

  # get alarm data
  warn("[Error] Wrong format in \"$cmd\".\n"), return unless $cmd=~/^(\d+)\s+(.+)/;
  my ($period, $msg)=($1, $2);

  # install a new one shot alarm timer
  Event->timer(
               prio   => 2,                           # before IO;
               at     => time+$period,                # set alarm;
               cb     => sub                          # callback;
                         {
                              warn "[Alarm] $msg\n";        # inform
                          $_[0]->w->cancel;           # clean up
                         },
               repeat => 0,                           # one shot;
              );

  # display a message
  warn '[Info] Your timer "', $msg, '" is running.', "\n";
 }
```

This script is a reminder: you can enter dates and it will remind you. Simply, isn't it?

---

4 3. Event in detail

This chapter describes details of **Event** which the introduction could not provide.

4.1 3.1. Watcher attributes

The properties of a watcher are determined by its *object attributes*. Attributes are set explicitly in a watchers constructor or later on by attribute methods. Besides this, a lot of attributes require no explicit setting because they have reasonable default values.

A number of base attribues are owned by watchers of *all* types.

base watcher attributes

attribut	***description***
unlimited access:	
cb	callback function to be invoked if an event happens
debug	trace level setting
desc	watcher description, useful to identify and search the watcher
max_cb_tm	callback timeout, a callback is interrupted and terminated if it exceeds this limit
prio	priority
reentrant	a flag permitting nested callbacks
repeat	controls if the watcher dies after the first event or not
read−only access:	
cbtime	time of last recent callback invocation (if you check this in the related callback, it shows its current startup time)
is_running	the number of callbacks currently running for the watcher
constructors only:	
async	enforces *immediate* event handling bypassing the event queue (ignoring other watchers completely), this is overwritten by `prio`
parked	prevents that the new watcher becomes active (it is made but detects no events)
nice	priority as offset to the default value, this is overwritten by `prio` and `async`

Beginning with version 0.60, **Event** additionally provides the base attribute `data`. It is intended not for configuration but for user controlled data storage.

Besides these base attributes, each watcher type owns more specific configuration settings.

spezific watcher attributes

attribute	***description***
io:	
fd	the watched handle
poll	specifies which events are of interest: this may be read or write access to the handle, errors or timeouts (or combinations)

timeout	after this time the callback is called even without a handler event
timeout_cb	alternative callback to be called if the event times out (this is an optional attribute, by default, the watcher will invoke the `cb` callback this case as well)
hard	specifies if timeouts of repeated calls start at invokation or finish of an callback (only useful if a timeout is specified)
timer:	
at	event time (ASSUMPTION: this is overwritten by `interval`)
interval	period until event
hard	specifies if a the interval of repeated calls starts running at start or finish of an invoked callback (only useful if `interval` is used)
signal:	
signal	the watched signal (as a string)
idle:	
max	period after which the callback should be invoked, even without event
min	period to wait between two subsequent callbacks (regardless if intermediate events)
var:	
var	the watched variable (by reference)
poll	describes access types of interest: reading or/and writing
group:	
timeout	period after which the callback should be invoked, even without event
add	contains a watcher object to be shadowed (the watched watchers activity is the expected event). This is a list attribute which can be used multiply. An event is recognized if any member of the so grouped watchers acts. (Group members can be *removed* by the watcher method `del()`.)

Attributes are initialized in the *constructor* by similar named *parameters*:

```
# register a timer
Event->timer(interval=>32, hard=>1, cb=>\&callback);
```

Additionally, attributes can be queried and modified during the whole lifetime of a watcher by similar named *watcher methods*, like so:

```
# modify a watchers description
$timerWatcher->desc("Really that late?");

# report callback runtime
print "Last recent callback started at ",
      POSIX::strftime("%c\n", localtime($w->cbtime));
```

4.2 3.2. Object management

You may have wondered about the constructor calls in the examples above. Usually, it is a good Perl tradition to take and store the object a constructor supplies:

```
my $watcher=Event->io(fd=>\*STDIN, cb=>\&callback);
```

And of course, this is possible with **Event** objects as well. But more often you will see simplified **Event** code like the following:

```
Event->io(fd=>\*STDIN, cb=>\&callback);
```

What's going on? The answer is simply that **Event** manages the watcher objects internally itself. If you want to access them later, you can found them by using class methods:

method	***description***
all_watchers()	supplies all registered watchers
all_running()	provides a list of all watchers with currently running callbacks
all_idle()	offers a list of *idle* watchers ready to be served but currently delayed by higher priorized events

The internal, automatic management of a watcher performed by **Event** results in the side effect that the reference counter of an watcher object is influenced by internal module operations. That's why in

```
{
 my $watcher=Event->io(fd=>\*STDIN, parked=>1);
}
```

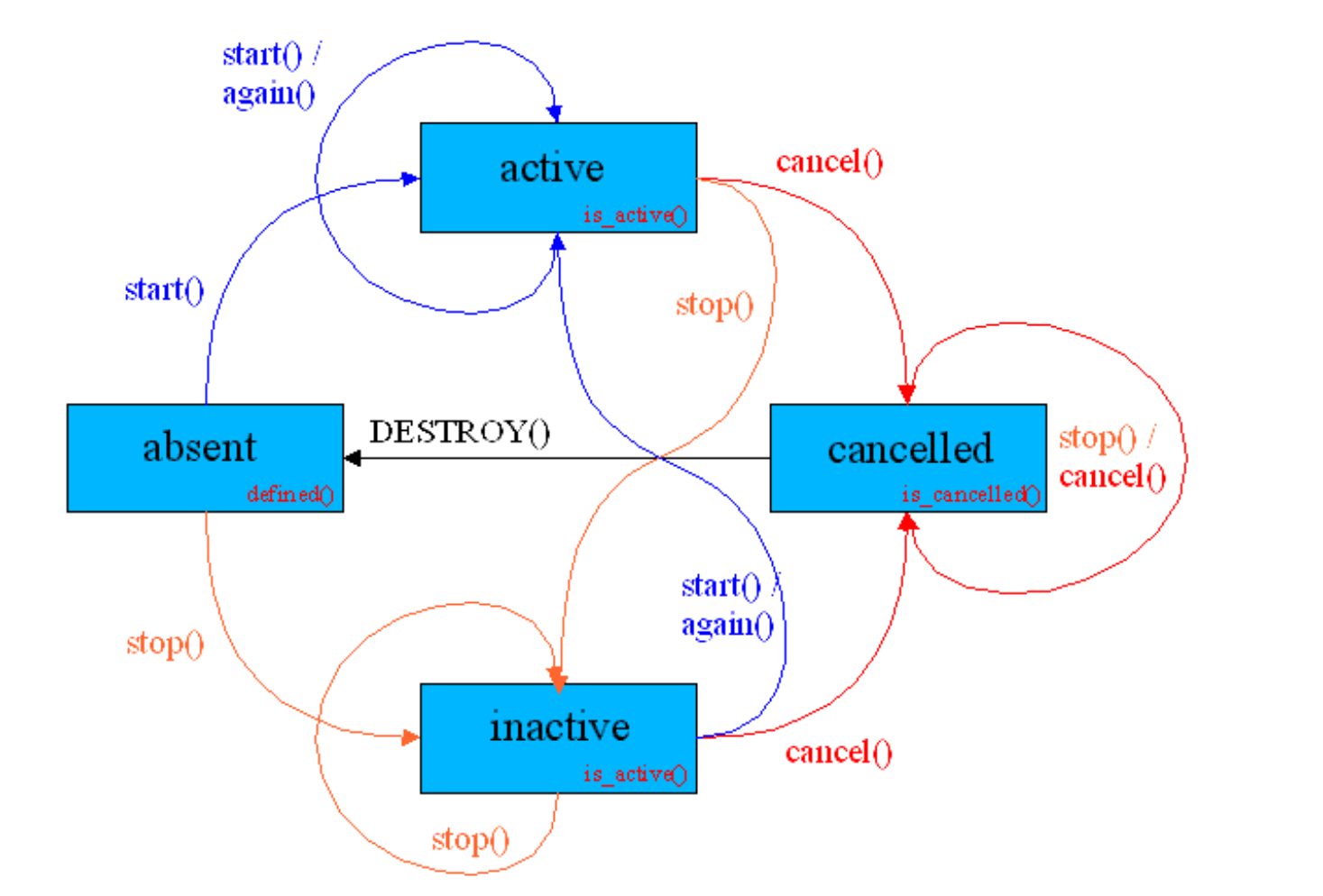
the new watcher remains alive even *after* the block is leaved.

But of course, if you want to do this, you *can* manage a watcher object yourself. This may be useful indeed if you have to access it later on because searching it via class methods can become a waste of time (if done regularly or you have to find certain watchers among a great number of such objects). Only keep in mind that you are not the only one managing a watcher object. Especially, *prevent modifying access* to a watcher after it was cancelled (or risk an exception). The current watcher state can be checked by various methods, especially by `is_cancelled()`.

The following section describes watcher states in detail.

4.3 3.3. A watchers life cycle

During its life, a watcher enters various *states* which determine its behaviour. This means that a watcher object is not only made, run and destroyed. You can, for example, deactivate it until you need it again. The several state changes can be initiated *explicitly by object methods*, or they are performed *implicitly caused by fullfilled preconditions*. The current state of a watcher is reported by special functions and methods.



The state of a watcher reflects both its *activity* and its *registration*. *Activity* describes if it waits for and detects events. *Registration* means if the loop knows the watcher so that the watcher can add handling orders to the queue.

States

ABSENT: The *initial* state of each Perl variable. The watcher is not made yet, or the watcher object was already destroyed. Such a watcher is neither registered nor active.

As usually, this special state can be detected by `defined()`.

ACTIVE: The watcher is registered and active, which means it detects events and generates handling orders in the queue.

`is_active()` replies a true value in this state.

INACTIVE: The watcher no longer takes care of events, they are ignored. It does not generate handling orders but is still registered. Regardless of all this, its entire configuration remains unchanged.

You can check for this state using the method `is_active()` as well. It supplies a "false" value this case.

CANCELLED: The watcher is no longer able to recognize events and generate orders, it is neither active nor registered. Because of the lost registration it cannot return into states where it would be registered. Its configuration remains unchanged but cannot be modified, any modifying access will raise an exception. A cancelled watcher cannot be reactivated.

There is a method `is_cancelled()` which can be used to check for this critical state before modifications are performed.

This state is intended to be very temporary. It is designed as the final watcher state before destruction and usually the watcher object passes through this state immediately inside the **Event** code. The only exception is caused by still existing external watcher references, located in an `Event::Event` object representing a still unserved order generated by the watcher, or in *your* data if you preferred to store watcher references. Take care in such cases.

State changes

Implicit changes: *In general, a watcher can occupy state **ACTIVE** only if its attributes are sufficient.*

```
A watcher without callback, for example,
cannot generate orders that make sense.
```

As soon as this precondition is violated, a watcher in state **ACTIVE** is transformed into state **INACTIVE** *automatically*. The following table describes which settings are the minimum for an active watcher.

sufficient watcher settings

watchertype	***preconditions***
all	callback set
io	timeout set or valid handle stored in `fd`
timer	timeout set, repeated calls only possible with valid interval
signal	valid signal stored in `signal`
idle	–
var	attributes `poll` and `var` have to be set, but not for read−only variables like `$1`
group	at least one watcher in the group

The following example demonstrates a forced implicit state change.

```
# deactivate an active watcher implicitly
# (demonstration only!)
my $w=Event->signal(signal=>'INT');
print "Watcher started.\n" if $w->is_active;
$w->signal('FUN');
print "Watcher deactivated.\n" unless $w->is_active;
```

Constructor calls: The parameter `parked` (if set to a true value) instructs the constructor to generate the new watcher in state **INACTIVE** by calling method `stop()` (the default state is **ACTIVE** entered by `start()`). Besides this explicit setting, **INACTIVE** is entered *implicitly* in case of insufficient attribute settings (see table above for details).

```
# new and active watcher
print Event->var(var=>\$var, cb=>\&cb)->is_active, "\n";

# similar, but explicitly deactivated
print Event->var(var=>\$var, cb=>\&cb, parked=>1)->is_active, "\n";

# insufficient attributes -> state INACTIVE
Event->io->is_active or die "[BUG] Insufficient watcher attributes!";
```

The constructor parameter `parked` was introduced to enable watcher storage. By prebuilding watchers expected to be used later on, you can accelerate your application because it is more expensive to make than to configure a watcher. On the other hand, measurements showed that the real performance advantage of such pools strongly depends on your application.

Deactivation: You can deactivate a watcher by calling its `stop()` method which enforces the watcher to enter the **INACTIVE** state. This does not influence orders of this watcher which are already stored in the queue. A deactivated watcher can be reactivated by calling its method `again()` (you may use `start()` alternatively). Of course, the **ACTIVE** state can only be entered if the watchers attributes are still sufficient.

```
# stop watcher temporarily ...
$w->stop;
```

```
...
# and reactivate it
$w->again;
```

Cancellation: To finally deactivate a watcher there is a method `cancel()`. Inside **Event**, it deregisters the watcher so that it becomes invisible to the loop and enters the state **CANCELLED**. All internal references to the watcher object are removed, and this means that unless there are further object references externally, Perls garbage collection will immediately remove the object by `DESTROY()`. But if there are still external references, the object will remain in state **CANCELLED**.

Where could external references be located? First, there may be orders made by the watcher still waiting in the queue. The order objects include a reference to their parent watcher. Second, your own code could have stored the watcher object somewhere.

```
# generate a cancelled watcher
my $cw=Event->io;
$cw->cancel;
print $cw->is_cancelled, "\n";
```

In no case the state change influences orders already stored in the queue.

4.4 3.4. Priorities

If events happen simultaneously, the callback invocation order should be determined.

```
In most cases it is useful to handle a
signal immediately, even if a timer
in the same moment wants to inform you
about coffee.
```

Priorities allow to control which event should be served before others in such a case. Lower priorized events have to wait before they can be served. For this purpose, **Event** provides eight levels of priority − including "immediately" as well as "sometimes". To simplify the interface, each watcher type has its own *default* priority.

priorities

level	***description***	***default***
−1	*asynchronous* handling: the callback is invoked without delay, ignoring the queue	
0	highest "regular" priority	
1		
2	provided as importable constant `PRIO_HIGH`	signal
3		
4	provided as importable constant `PRIO_NORMAL`	idle, io, timer, var
5		
6	lowest priority	

But of course everyone can specify its own priority hierarchy. All watcher constructors offer three attributes for this purpose: `prio` sets an explicit priority, `nice` defines the target priority as an offset to the default value, and `async` selects priority `−1`.

```
# a default signal watcher
$sigWatch=Event->signal(signal=>'PIPE');
print "Default: ", $sigWatch->prio, "\n";

# watcher with explicit priority setting
$sigWatch=Event->signal(signal=>'PIPE', prio=>1);
print "Prio 1: ", $sigWatch->prio, "\n";

# constructor using prio offset
```

```
$sigWatch=Event->signal(signal=>'PIPE', nice=>-2);
print "Default-2: ", $sigWatch->prio, "\n";

# signals should be served immediately
$sigWatch=Event->signal(signal=>'PIPE', async=>1);
print "Asynchronous: ", $sigWatch->prio, "\n";
```

If more than one priority attribute is passed to the constructor, `prio` will overwrite `async`, and both have precedence over `nice`.

And, of course, the priority setting can be modified at runtime as well, even if there is only *one* method to do this: `prio()` (`async` and `nice` are available in constructors only).

```
# signals should be served immediately
$sigWatch=Event->signal(signal=>'PIPE', async=>1);
print "Initial: ", $sigWatch->prio, "\n";

# oops, back to default priority
print "Modified priority: ", $sigWatch->prio(PRIO_HIGH), "\n";
```

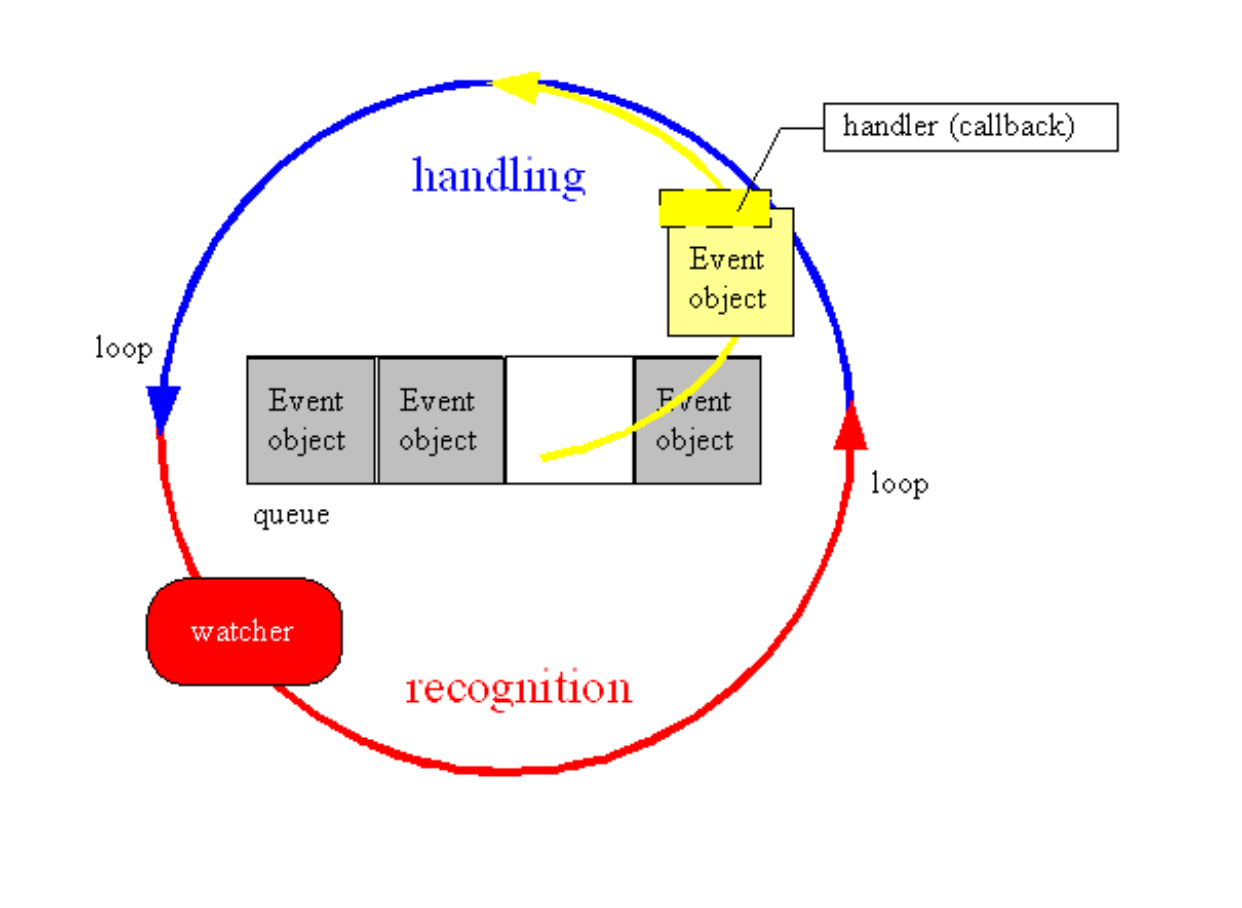
If you are building a priority hierarchy for various watchers, please keep in mind that even events of the lowest priority should finally be served. That's why an events priority has not only to reflect *importance* and *urgency* of its handling but has also to take care of its (probable) *frequency*. *If "important" events occur too often they may block all other watchers.* The optimal design sees very important events happening extremely seldom.

4.5 3.5. Watcher teams

It is explicitly allowed to have an unlimited number of watchers for the same event, regardless of the watchers type. If such a well watched event happens, *all* callbacks are invoked *subsequently*. (Nevertheless, priorities are still in effect, so there is no guarantee that the sequence of orders may not be interrupted by a callback of a watcher outside the team if you assign different priorities to the team members.)

4.6 3.6. Writing callbacks

Orders in the queue are represented by objects of the class `Event::Event`. The loop performs the order by invoking the callback function stored in this object.



The `Event::Event` object is passed to the callback as its only parameter, this is managed automatically. Because it stores more informations besides the callback itself, it connects the callback with both the initial event and the watcher which detected the event and generated the order. Once the callback is finished, this intermediate transfer object is destroyed.

```
# callback taking the Event object
Event->io(..., cb=>sub {my ($event)=@_;});
```

So well, an `Event::Event` object is very passing thing, but nevertheless it plays an important part in handling an event.

`Event::Event` objects are very similar to *watcher* objects: they own attributes which can be accessed by methods of the same name. But different to watchers `Event::Event` attributes cannot be modified, they are read−only.

Event::Event object attributes

attribute	***description***
got	only available if the corresponding watcher has a `poll` attribute: then it describes the event in `poll` format
hits	here the watcher stored the sequential number of the generated order, so you can see how often the initial event was detected by the watcher yet
prio	the parent watchers priority (at *event* time)
w	the parent watcher object (in *current* state)

Especially the offered access to the parent watcher is often used to modify the watchers configuration or state, like so:

```
sub callback
 {
  # get Event object
  my ($event)=@_;
```

```
  ...
  # cancel the initial watcher, if possible
  $event->w->cancel if $allExpectedEventsArrived;
  ...
}
```

Keep in mind that the watcher may have been modified between the events occurence and the callbacks invokation. While an `Event::Event` object "freezes" the event state in the queue, the related watcher works on and all parts of the program are free to modify it until this event will be handled by callback invokation, which *could* take a significant while. The watcher might even been cancelled which means that modifying access would raise an exception. That's why you should check a watchers state before you modify it in a callback.

```
# something seems to block us, we should act more often!
$event->w->prio($event->w->prio-1) unless $event->w->is_cancelled;
```

On the other hand, often your callbacks will not need the informations provided by the passed `Event::Event` object and you can ignore it.

Besides the parameter interface there is only *one* thing which should be taken into account: ***a callback should return quickly***. Remember that there is only one process for event recognition and handling, which ideally means the handling of all detected events in a reasonable time. As long as a callback is performed, watchers, loop and other callbacks are definitly blocked. That's why a long running callback should be shortened. There are several methods to do this, one is to *split it up into partial tasks* which are performed subsequently by a state machine. Each callback invokation could handle one state, for example. Another method is *delegation to other processes* which might run in the same process space (threads or subprocesses) or in a foreign one (on a server). The third alternative is *cooperation*: you can enforce an intermediate event recognition and handling by calling **Events** class method `sweep()`. This and other class methods of loop management are described in the next section.

4.7 3.7. Loop management

Besides watchers and callbacks the event loop itself is the third pillar of **Event**. The loop is managed by various class methods. In most cases, `loop()` and `unloop()` are sufficient.

An interesting aspect of the **Event** design is that *loops can be nested*. This means that you can call `loop()` from a running callback (which is embedded into a loop itself) to enter a new inner loop level. Nevertheless, all registered watchers are still active there.

loop control

methode	***description***
loop()	enters a new loop level. Loops are terminated automatically unless there are active watchers.
unloop()	terminates the *most inner* loop level, all registered watchers remain unchanged.
unloop_all()	terminates *all* loop levels (but still without effect to the installed watchers)
sweep()	enables a callback to let **Event** recognize and handle intermediately occured events. After doing so, `sweep()` returns immediately. (The priority of events to be served by the method can be limited by a `sweep()` parameter.)

Because a loop without active watchers terminates itself immediately, the following idiom is often used to cleanup both loops *and* watchers:

```
# stop all loops AND deregister all watchers
$_->cancel foreach Event::all_watchers;
```

5 4. Advanced features

5.1 4.1. Watching Watchers

Event provides a number of powerful features for debugging, error tracking and tuning. Already a default installation offers a class variable `$Event::DebugLevel` and a `debug` attribute in each watcher to activate traces of various levels. If compiled with `-DEVENT_MEMORY_DEBUG`, **Event** offers an additional class method `_memory_counters()` which informs about the currently installed watchers.

```
# display installed watchers
warn "[Trace] Watchers: ", join("-", Event::_memory_counters), "\n";

# This displays something like
# "1-29509-0-0-0-0-5-0-3-0-8-0-0-0-0-0-0-0-0-0", where
# each slot is a certain event or watcher counter.
```

Even more, there is an add on module **Event::Stats** which provides ways to interrogate runtime informations of every certain watcher. Finally, the add on module **NetServer::ProcessTop** can be used to install a small telnet server within an **Event** application which lists all registered watchers *live* in tradition of the UNIX utility `top`. It is fascinating to look inside the running loop, watching the whole process or a user defined group of watchers! But the highlight of all indeed is the possibility to use this server to modify and tune watcher attributes and states dynamically from a remote site.

```
Watchers at work: NetServer::ProcessTop

serviceStatvfs PID=10012 @ redbull                          | 15:57:33 [  60s]
14 events; load averages: 0.97, 0.98, 0.00; lag  0%

  EID PRI STATE   RAN  TIME   CPU TYPE
DESCRIPTION                          P1
   10   4 sleep    84  0:46 86.2%   io action registration socket
    5   3 sleep     1  0:05  9.9%   io NetServer::ProcessTop
    0   7         150  0:00  1.6%  sys idle
    7   3 wait     70  0:00  1.6% idle idle process
    3   3 sleep    51  0:00  0.4%   io interface connection to s8a8263 via port
   16   3 cpu      11  0:00  0.2%   io NetServer::ProcessTop::Client s8a8263
    2   3 sleep    13  0:00  0.0% time Event::Stats
    9   4 sleep     0  0:00  0.0%   io more restricted interface registration s
    8   4 sleep     0  0:00  0.0%   io less restricted interface registration s
   11   2 sleep     0  0:00  0.0% time controler host list update timer
   12   2 sleep     0  0:00  0.0% time action host list update timer
   13   1 sleep     0  0:00  0.0% sign signal handler for HUP
   14   1 sleep     0  0:00  0.0% sign signal handler for INT
   15   1 sleep     0  0:00  0.0%   io controler socket
    6   6 sleep     0  0:00  0.0% time system check timer: actions
    0  -1           0  0:00  0.0%  sys other processes

%
```

It is planned to extend **NetServer::ProcessTop** by remote symbol table inspection.

5.2 4.2. Watcher suspension

Every watcher can enter a special *mode* **SUSPENDED**. This mode behaves similar to a state at first sight but is very different in detail. It was implemented *for development, tuning and debugging*. This mode only effects activity.

Suspension enforces watchers in state **ACTIVE** or **INACTIVE** to behave exactly like a deactivated one *while they still own their original states*: they do not recognize events and therefore generate no orders. (It is possible to suspend a cancelled watcher as well, but without visible effect.) You may imagine that **SUSPENDED** *freezes* a watcher so that you can study it as long as you want without disturbance by timeouts or something like that. (And of course, you *can* change the watchers *real* state while it is suspended.)

SUSPENDED (similar to states) provides a special recognition method. This is `is_suspended()`.

The "freezing" of watchers is exclusively controled by the attribute `suspend` and the method `suspend()`, respectively. Event recognition and order generation are disabled as long as the attributes value is true. This takes no effect to orders already stored in the queue or to the real watcher state because *suspensions were designed as a utility for debugging, development and tuning*. That's why a watcher can be both **ACTIVE** and **SUSPENDED** at the same time. Because of this it is not recommended to use suspensions in your applications real code, `stop()` suits better there.

```
# build a new active watcher
my $w=Event->var(var=>$object, cb=>\&cb);
print "Watcher started.\n" if $w->is_active;

# suspend the watcher, check its state
$w->suspend(1);
print "Watcher is still active ...\n" if $w->is_active;
print "... but suspended.\n" if $w->is_suspended;

# cancel suspension
$w->suspend(0);
```

Additionally, the **NetServer::ProcessTop** module introduced in the previous section provides a way to suspend watchers *remotely*.

The special intention of this mode becomes visible in the following example as well. It shows that **Event** embeds a watcher into a *very special environment* if it enters **SUSPENDED**. This enables to perform operations which would normally be denied by **Event**, e.g. setting a watcher with insufficient attributes into state **ACTIVE**. Please note that **Event** rebuilds a valid watcher state when **SUSPENDED** is leaved.

```
# In this example a watcher with insufficient
# attributes is set ACTIVE. This would normally
# be prevented by Event, but is possible in
# state SUSPENDED. As soon as SUSPENDED is leaved,
# Event immediately restores a valid state.

use strict;
use Event;

# make proband
my $w=Event->io(fd=>\*STDIN, parked=>1);

# check
state($w);
switch($w, 'suspend', 1);
switch($w, 'again');
switch($w, 'stop');
switch($w, 'start');
switch($w, 'suspend', 0);
switch($w, 'suspend', 1);
switch($w, 'again');
switch($w, 'stop');
switch($w, 'start');
switch($w, 'cb', sub {});
switch($w, 'suspend', 0);
switch($w, 'cancel');
state($w);

sub switch
 {
  # get operation
  my ($w, $op, @par)=@_;

  # get current state
  my @prev=($w->is_active(), $w->is_suspended(), $w->is_cancelled());

  # perform operation
  eval {$w->$op(@par);};
  die $@ if $@;
```

```
   # check new state, prepare message
   my ($msg, $diff)=("$op(@par): ", 0);
   $msg=join('', $msg, $diff?', ':'',
             "activity: $prev[0] ==> ", $w->is_active
            ),
     $diff=1 if $prev[0] ne $w->is_active;

   $msg=join('', $msg, $diff?', ':'',
             "cancellation: $prev[2] ==> ", $w->is_cancelled
            ),
     $diff=1 if $prev[2] ne $w->is_cancelled;

   $msg=join('', $msg, $diff?', ':'',
             "suspension: $prev[1] ==> ", $w->is_suspended
            ),
     $diff=1 if $prev[1] ne $w->is_suspended;

   # report changes
   print "$msg.\n";
  }

 sub state
  {
   # get operation
   my ($w)=@_;

   # report state
   print "STATE: active=>", $w->is_active,
         ", cancelled=>", $w->is_cancelled,
         ", suspended=>", $w->is_suspended, ".\n";
  }
```

Note: **SUSPENDED** is entered *internally* during callback execution if the callbacks "parent" watcher unset its `reentrant` attribute. This way nested callbacks can be prevented by **Event** without touching the user controled watcher state.

5.3 4.3. Customization

Event offers a wide range of flexible tuning features to the experienced user which could not be described here in detail.

Raising exceptions are caught and displayed as messages, while the loop still runs uneffected. This handling is very similar to the behaviour of `eval()` but can be replaced by a user provided function.

Important parts of the internal **Event** kernel can be extended or replaced by own routines if one prefers.

Finally there is a special API which allows to write fast callbacks in `C`.

5.4 4.4. Event and other looping modules

Perl/Tk implements its own event handling. That's why it cannot be combined with **Event** today (as I know), so the usual statement that a Perl script can easily get a graphical interface by using **Perl/Tk** is not necessarily true if this script uses **Event**. Instead of this, you would have to decide which loop to use. But as I know, Nick−Ing Simmons (the author of **Perl/Tk**) is watching **Event** carefully. Maybe there is a common loop one day − but this is still only a wish.

gtk+, another popular GUI library used together with Perl, is built on yet another event model (from glibc). As I know today, there is no successfully tried way of combination with **Event**. Perhaps it could be found by using **Events** hooks?

Contrary to this, **PerlQt** which provides one more framework for graphical interfaces is reported to work *very well* with

Event.

```
J. N. Pritikin provided this example
of teamwork. It demonstrates how Event
and Qt can be combined.

use Qt 2.0;
use Event;

package MyMainWindow;

use base 'Qt::MainWindow';
use Qt::slots 'quit()';

sub quit {Event::unloop(0);}

package main;

import Qt::app;

Event->io(
          desc    => 'Qt',
          fd      => Qt::xfd(),
          timeout => .25,
          cb      => sub {
                          $app->processEvents(3000);  #read
                          $app->flushX();             #write
                         }
         );

my $w=MyMainWindow->new;

my $file=Qt::PopupMenu->new;
$file->insertItem("Quit", $w, 'quit()');
my $mb=$w->menuBar;
$mb->insertItem("File", $file);

my $at=1000;
my $label=Qt::Label->new("$at", $w);
$w->setCentralWidget($label);

Event->timer(
             interval => .25,
             cb => sub {
                        --$at;
                        $label->setText($at);
                       }
            );

$w->resize(200, 200);
$w->show;

$app->setMainWidget($w);
exit Event::loop();
```

Complicated at first sight is the teamwork of **Event** and other modules implementing some sort of event handling as well, like **Term::ReadLine::Gnu**. This module, if configured that way, catches every keystroke passed to `STDIN` to implement autocompletion of filenames and commands for example. Well, a keystroke is an event as well if someone watches `STDIN`. More than that, there are two kinds of loops now. But with the help of both module authors, it became clear that this problem can be solved. The **Event** distribution contains an example demonstrating how these modules can be combined.

---

6 5. Application examples

The following projects on base of **Event** were reported in the **Event** mailinglist:

Application
A bankers **trading system**.
The state machine library **POE** (to be found on CPAN) shall be reimplemented using **Event**.
A **system watching utility** checking logfiles, ports, network, processes and more. If an event is detected, an alarm is send to either a mailbox or a tool like Tivoli. This system is reported to be flexible, completely configurable and modular built on base of user defined agents. The author wrote: "The Event module allows me to service all agents in a controlled & timely manner ... Watching a few active log files & testing each record against 20−30 regex's, checking the process list & netstat every minute, opening a couple of application ports & passing some info from time−to−time, & keeping an eye on paging −− in total, costs about a minute of CPU a day. The benefits are many."
database frontends
web backends
client/server architectures

7 6. Outlook

Event is constantly improved. J. N. Pritikin provides excellent maintenance and listens very carefully to users. He often publishes patches and fixes within hours. Some weeks ago he announced a business caused change into a Windows environment which might possibly result in a Windows port of his module.

The Perl porters group which is the core team in Perl development already started to discuss the need of an event handling model built into Perl itself. The discussion seems to be in an early state and it is still unclear if this extension will base on **Event**. But in spite of this discussion, **Event** allows flexible event driven programming in Perl already *today*.

---

8 A. Data

Information	***Details***
name	**Event.pm** by J. N. Pritikin (JPRIT).
version	This introduction is based on version 0.67.
modules manpage	http://theoryx5.uwinnipeg.ca/CPAN/data/Event/Event.html
implementation	mostly in *C* for maximal performance. Lots of "Perl magic".
limits	**Event** loops *are* threadsafe *as long as* ***Event*** *is used in only one thread*. Better thread support is already planned.
known bugs	If a script using **Event** dies, perls final memory cleanup process can fail sometimes. If the script is started standalone this only causes some curious error messages which you might never seen before. Take care if the script call is embedded into another program. The reasons of this failure are still unclear but perl itself seems to support it by an own already reported bug.
platforms	UNIX (various derivates), a Windows port seems to be possible in the future.
support and discussion	mailinglist perl−loop@perl.org.