Event driven programming in Perl using the Event module

Event driven programming in Perl using the Event module

.2.00Jochen Stenzel (perl@jochen−stenzel.de)7 December 2000

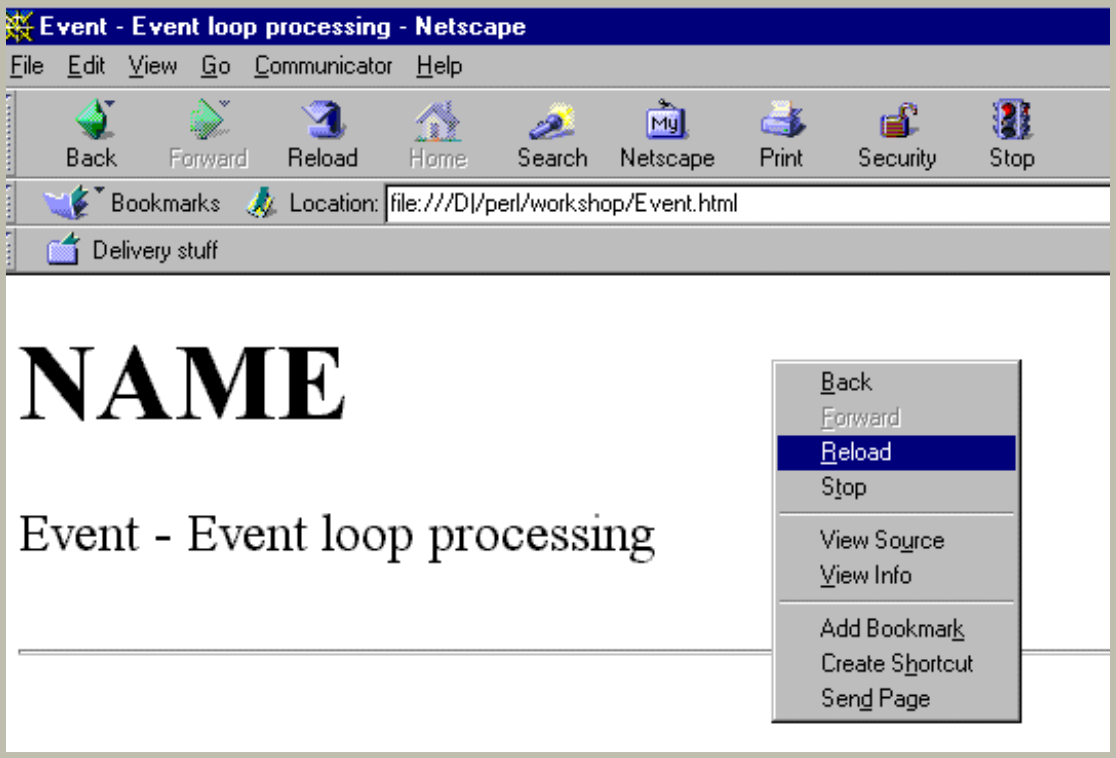
Table of Contents

---

1. Introduction

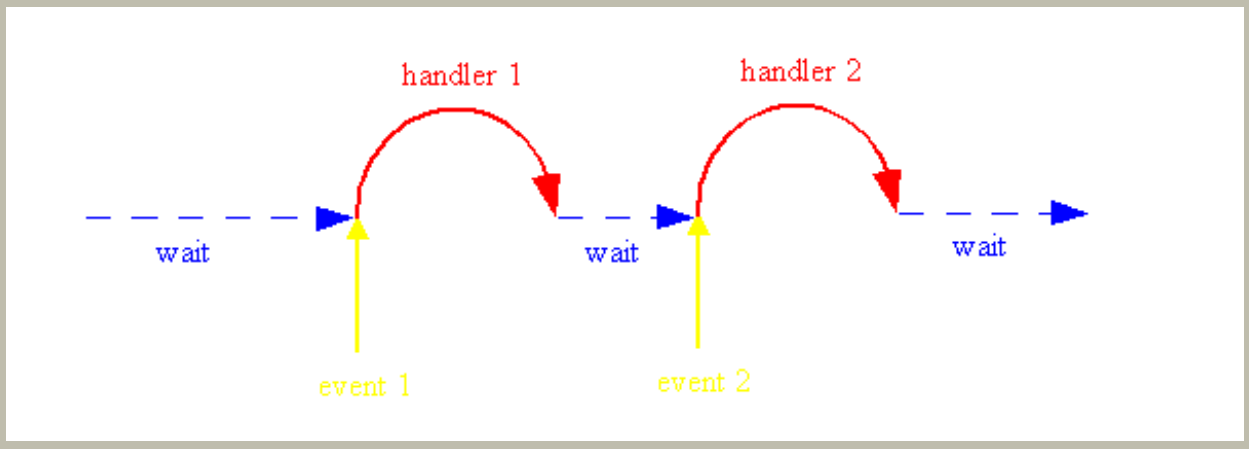
1.1. The task

Everyone of us knows programs like this

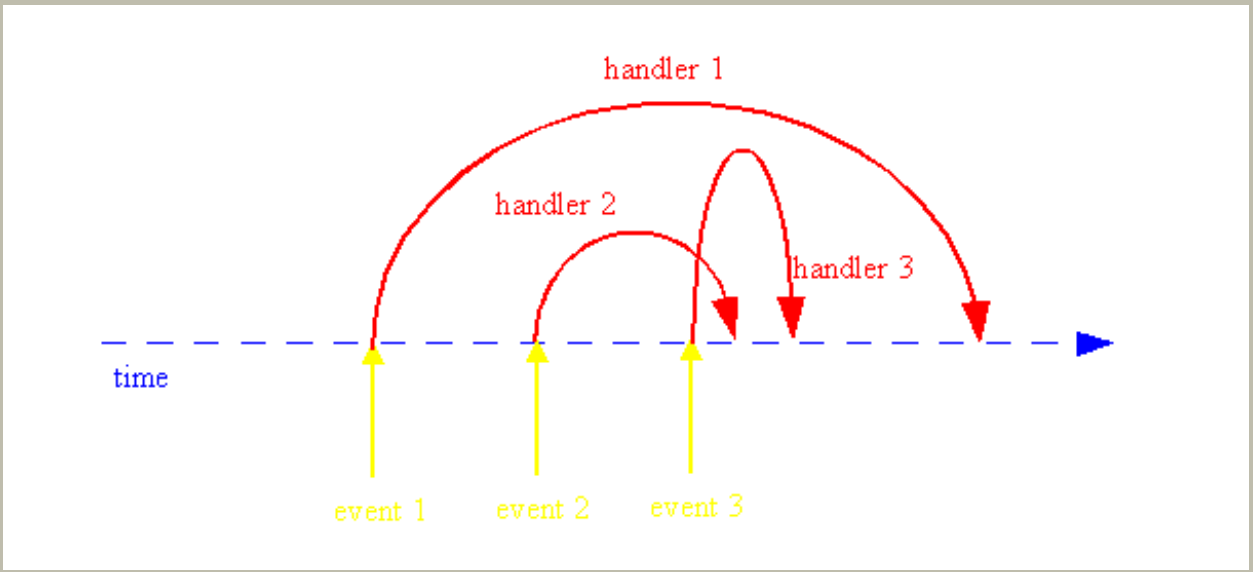


A click somewhere, and something happens. Another click somewhere else, and (usually) anything else happens. If my application is a web browser receiving a site from a server, the browser is able to take this site while it is still usable. Hopefully. So, if I prefer, I can stop the download even before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

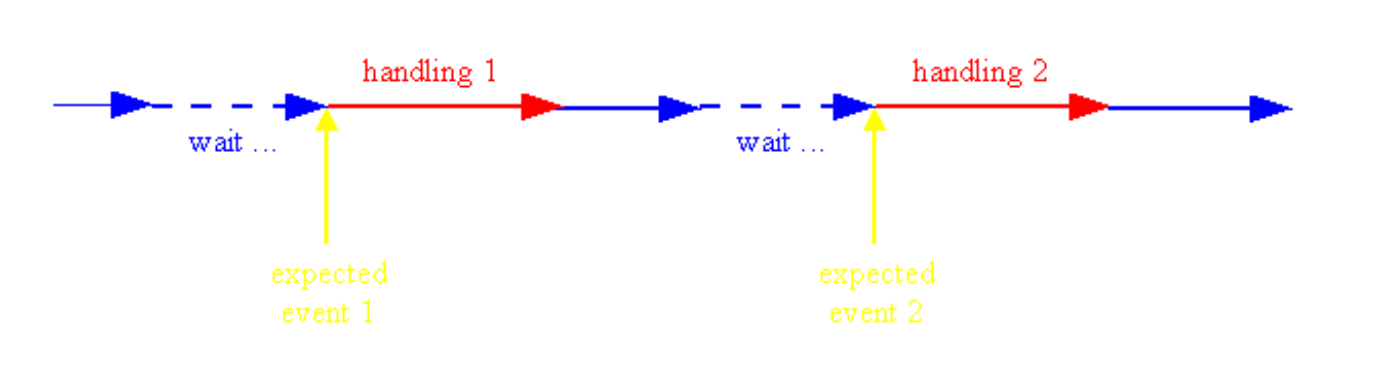


Various graphical interfaces do even more than that: they work **asynchronously** as well. This means that a second event can be detected while the first event is served, a user has not to wait until the first result arrives. It also means that the results of earlier events can be provided before the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

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!

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.

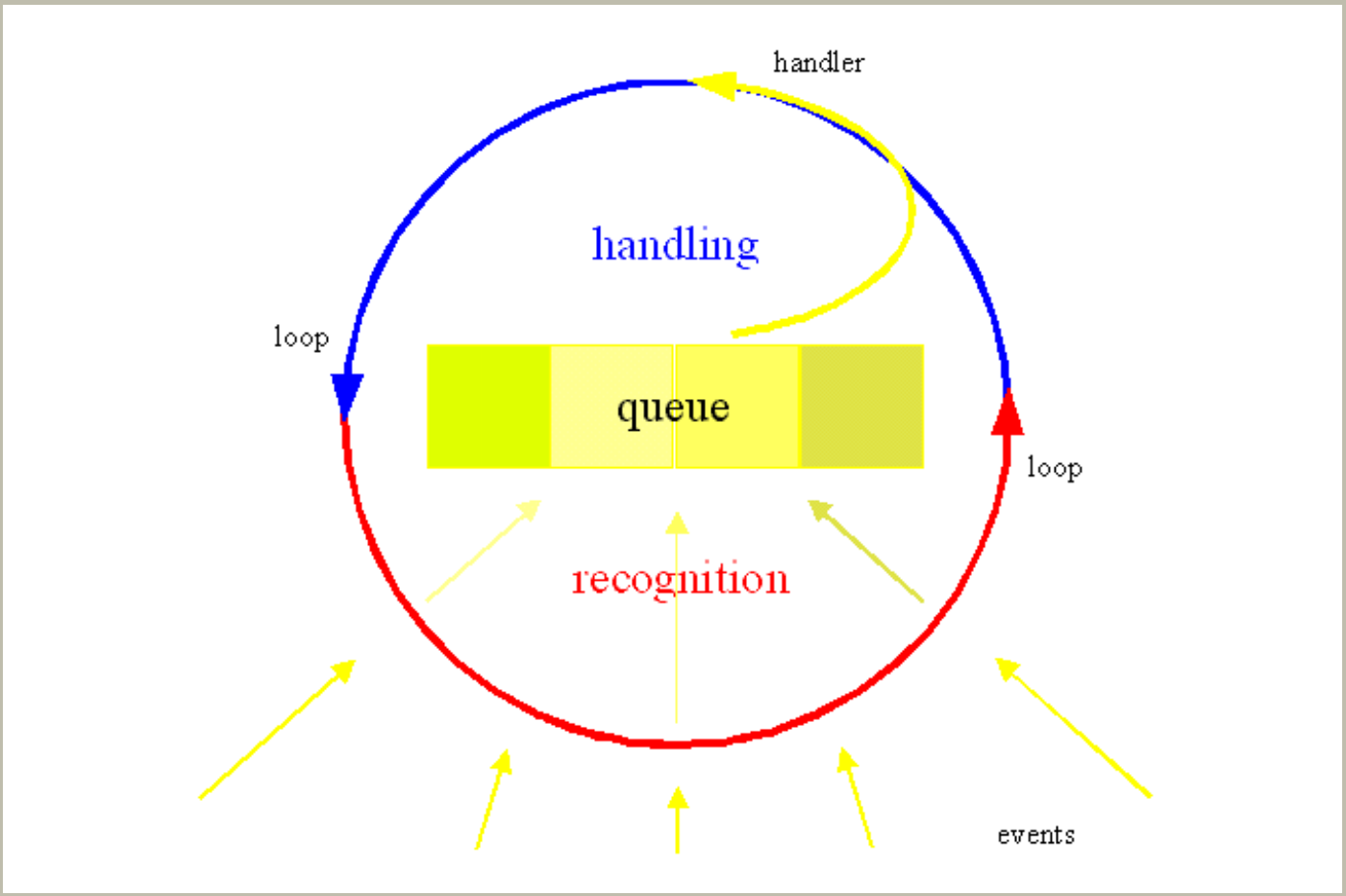
Well, on the other hand, `select()` and **IO::Select** are *restricted* to *exactly one timeout* and events happening on *handles*. Other types of events are not covered. Changing and maintaining event masks is *not* easy. While it *is* possible to build event loops basing on `select()` / **IO::Select**,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

But there is no need for such effort: **Event by Joshua 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.

---

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*.

2.1. The watcher concept

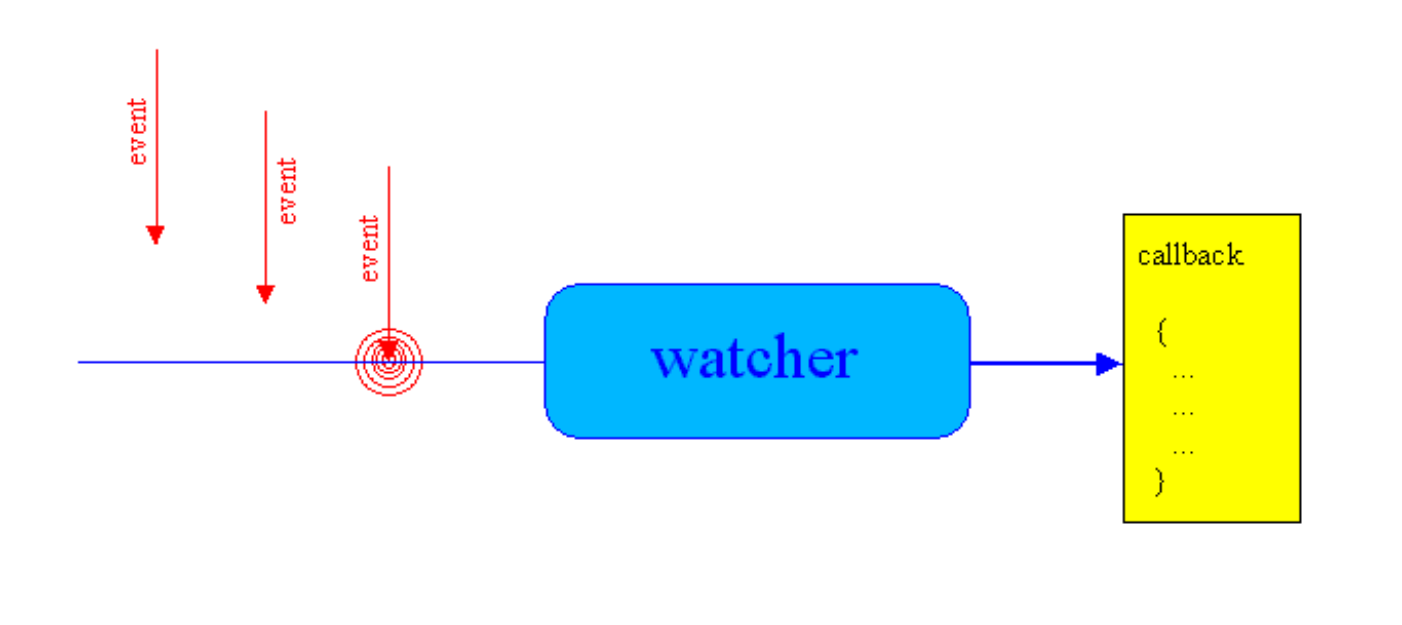
Event works an object oriented way. Its main actors are so called *watchers* which expect the happening of certain events and are prepared to initiate appropriate answers.

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**

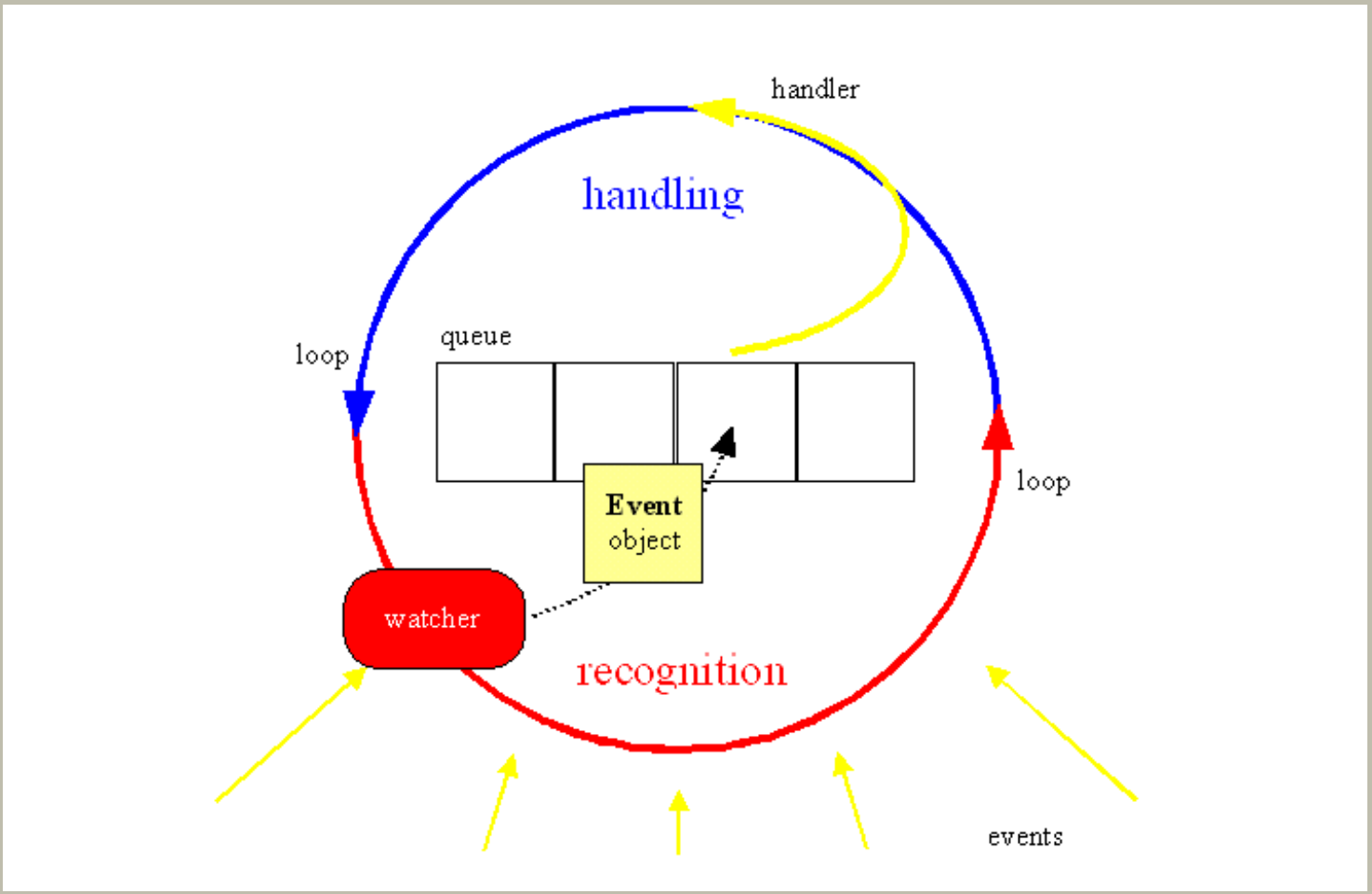
nothing else happened	**idle**
variable access	**var**
other watchers acting	**group**

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*.

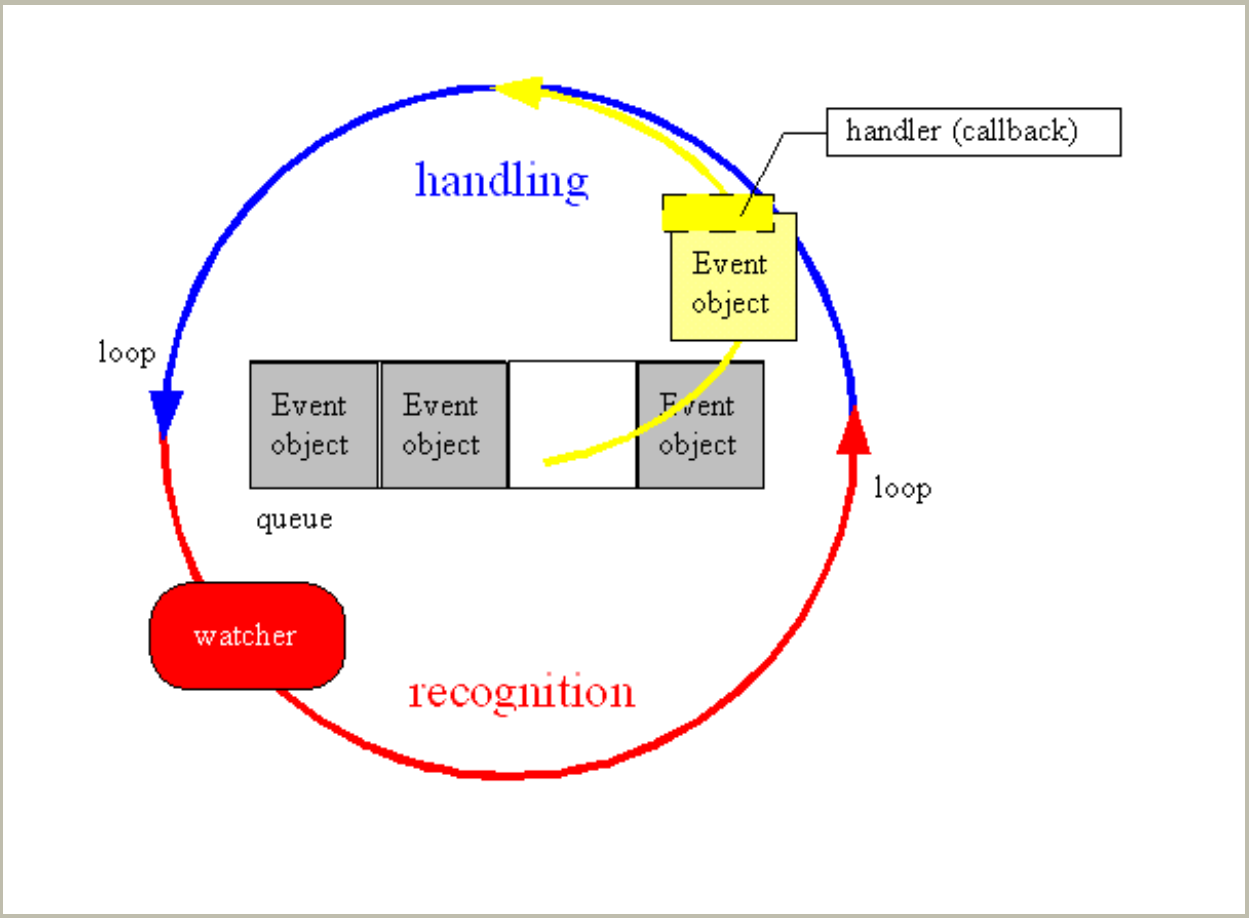


As soon as a watcher detects an occurence of its target event it initiates the call of its handler. Well, in principle.

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.



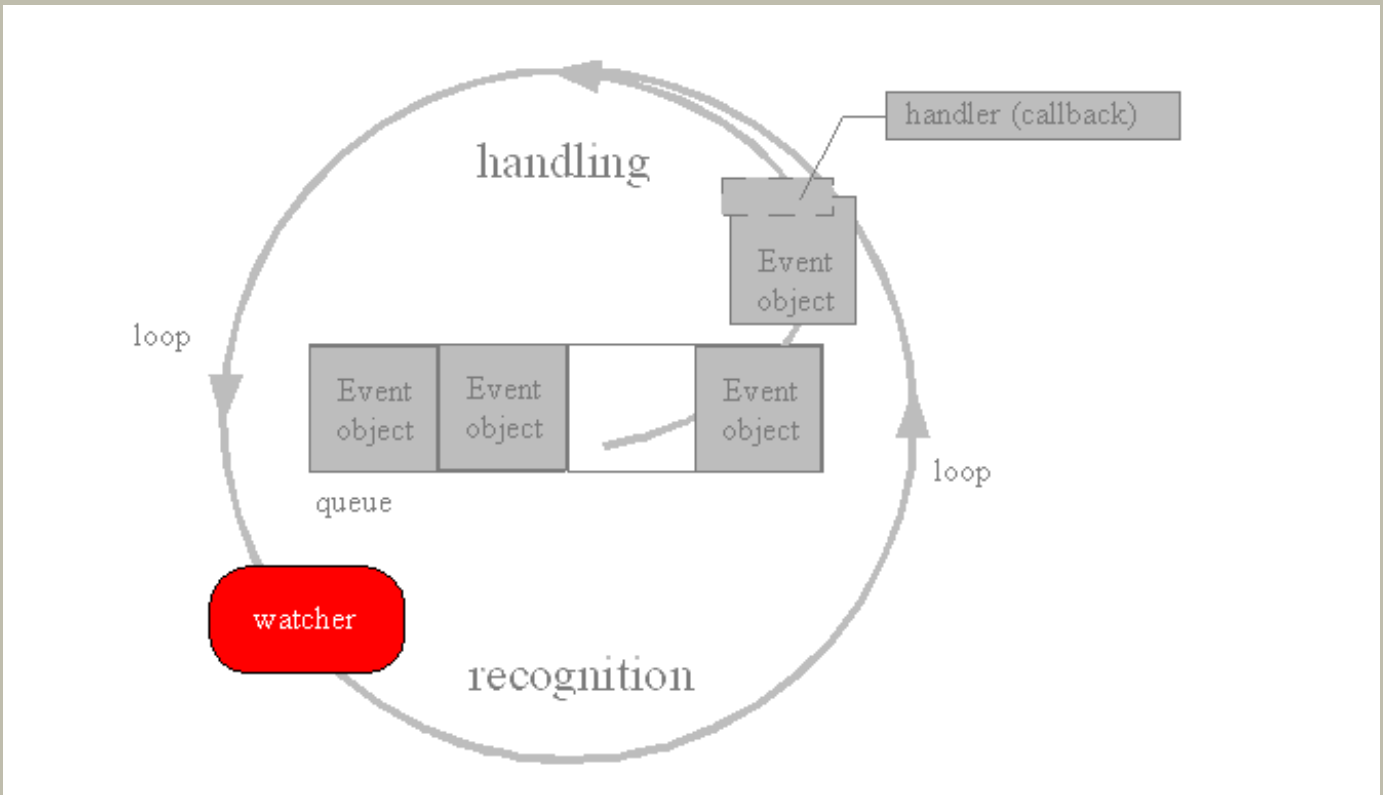
After the order is carried out completely the loop destroyes the order object.

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.



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(
             at => time+300,
             cb => sub {warn "Look at the pizza!";},
            );
```

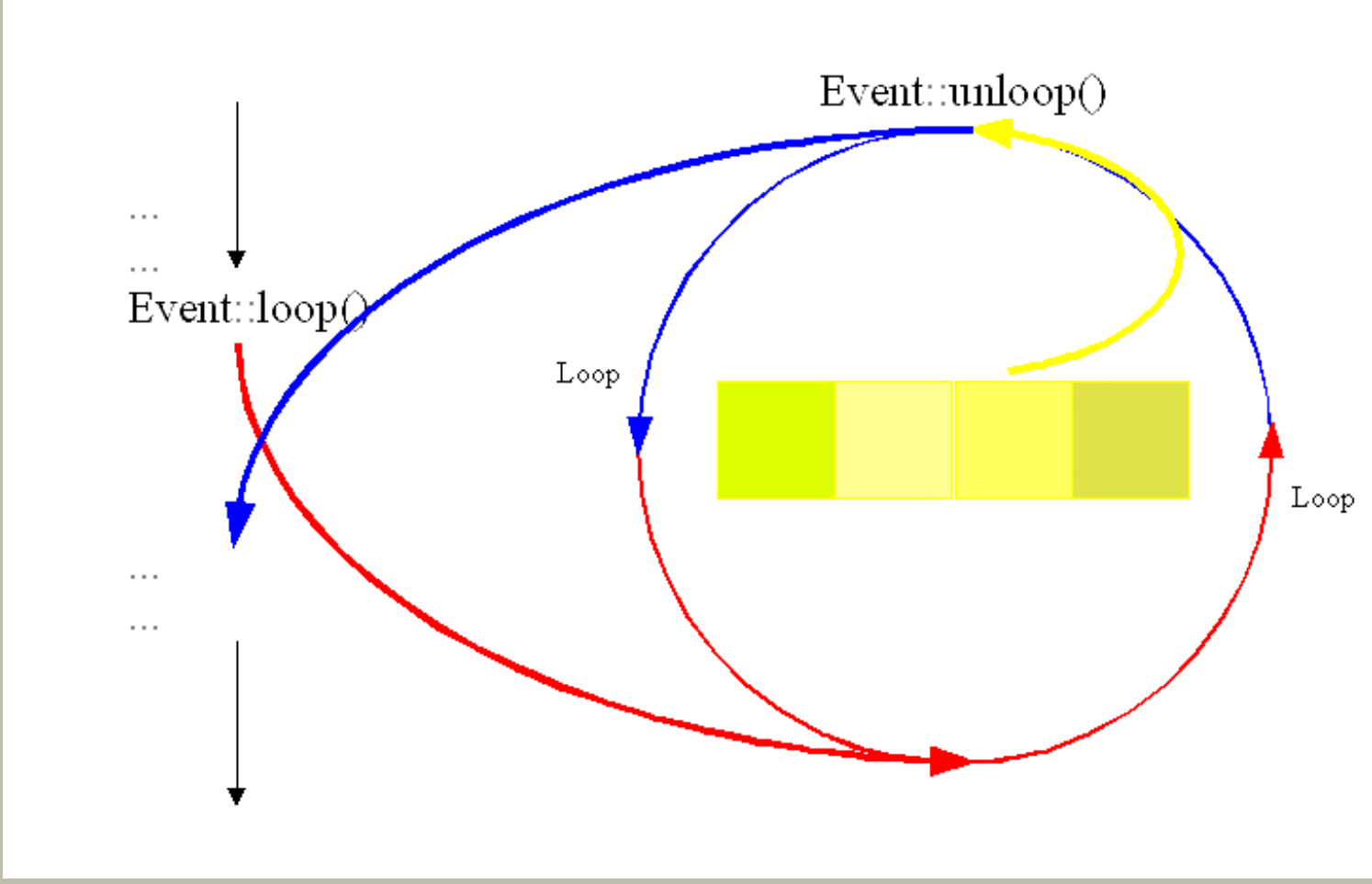
All these shown example watchers become active immediately after the constructor call. This means that they detect events. But, to give them a chance to *handle* these events, you have to establish the loop.

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.

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.

```
Working on the current project, at 17:20 you
realize that you will have to interrupt in a
while, so you quickly note:

3600 Last metro today!
[Info] Your timer "Last metro today!" is running.

A few minutes later, you start the coffee
machine but immediately continue working.
Just for safety you type in:

300 coffee
[Info] Your timer "coffee" is running.

Now this evening is saved. Five minutes later
a message arrives:

[Alarm] coffee

, and just in time 18:20 the message

[Alarm] Last metro today!

reminds you to go home.
```

Simply, isn't it?

3. Event in detail

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

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 stops after the first event or not (please note that stopping a watcher does *not* mean to cancel it − a stopped watcher is *inactivated but still registered* − see the watcher state chapter for details)
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.

For *developers*, version 0.75 introduced `private` to build watcher subclasses.

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

specific 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	the clock time when the timer will expire (ASSUMPTION: this is overwritten by `interval`)
interval	seconds between two consecutive timer expiries
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	the maximum amount of time to wait between consecutive
callbacks, regardles	s of the loops busyness
min	the minimum amount of time to wait between consecutive
callbacks, regardles	s of the amply idle time
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));
```

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 find 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);
}
```

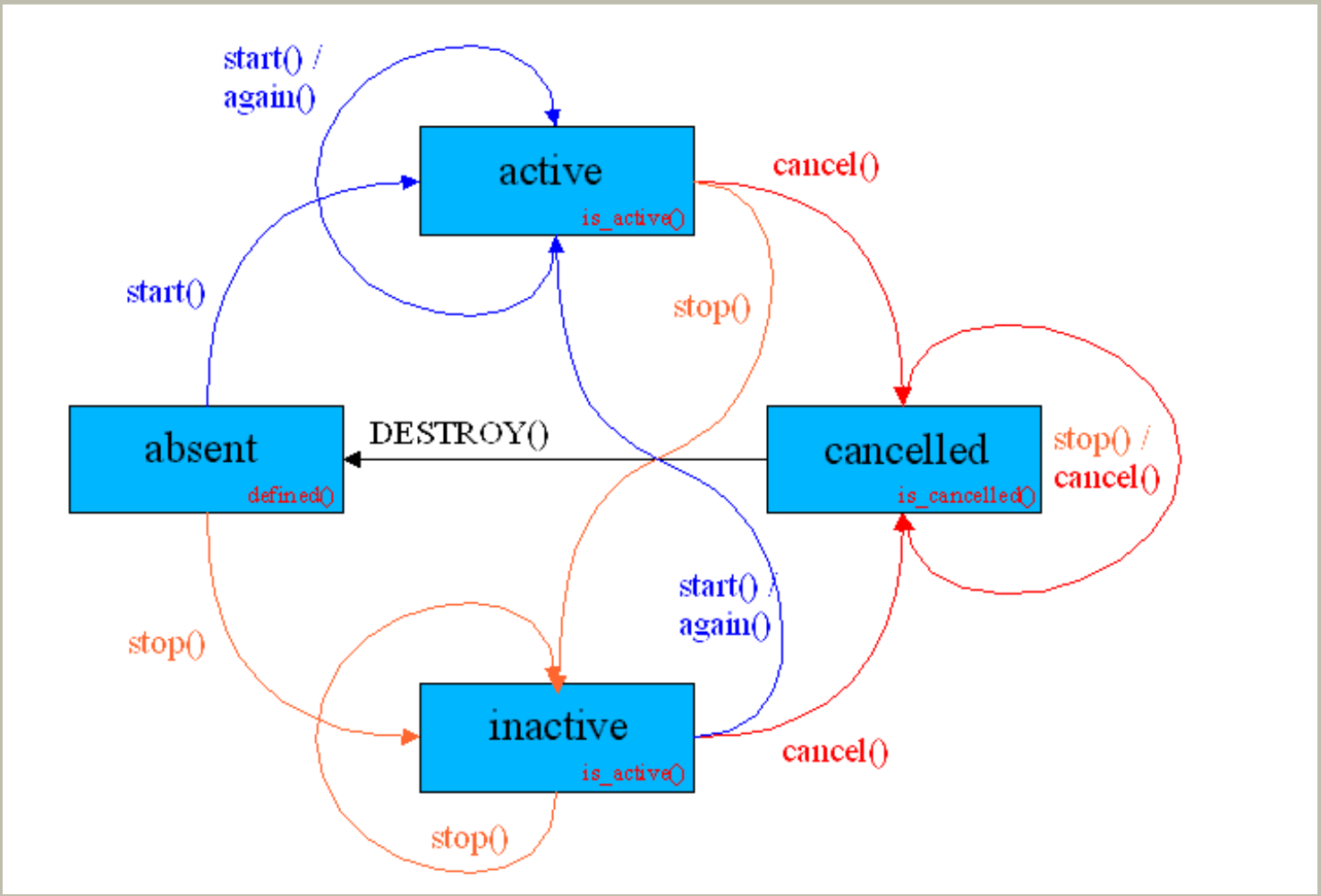
the new watcher remains alive even *after* the block is leaved.

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.

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

watcher type	***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;
```

There's another group of implicit state changes: *every non repeating watcher which is not explicitly cancelled enters **INACTIVE** after callback execution*. This is important because such watchers tend to be forgotten after callback invokation, but in state **INACTIVE** they are still alive and still consuming memory.

Note: An application which dynamically installs non repeating watchers will consume more and more memory unless these watchers are explicitly cancelled or reused after callback execution.

Here is an illustration script:

```
# load module, set pragma
use Event;
use strict;

# declare a simple counter
my $i=0;

# install the main timer for report purposes
Event->timer(
             interval => 0.1,
             cb       => sub {
                              # report registered watchers
                              my @watchers=Event::all_watchers;
                              warn "There are ", scalar(@watchers), " watchers now.\n";
                              # install a new non repeating timer
                              Event->timer(
```

```
                                        at => time,
                                        cb => sub {$i++; warn "$i. one shot callback!\n";}
                                       );
                                }
            );

# enter loop
Event::loop;
```

The next table summarizes where to take special attention to inactivated watchers.

watcher	*implicit change into INACTIVE after callback*
io	**repeat** set to 0
timer	specified by **at**, or **repeat** set to 0
signal	**repeat** set to 0
idle	by default (ASSUMPTION)
var	**repeat** set to 0
group	by default (ASSUMPTION)

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.

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";
```

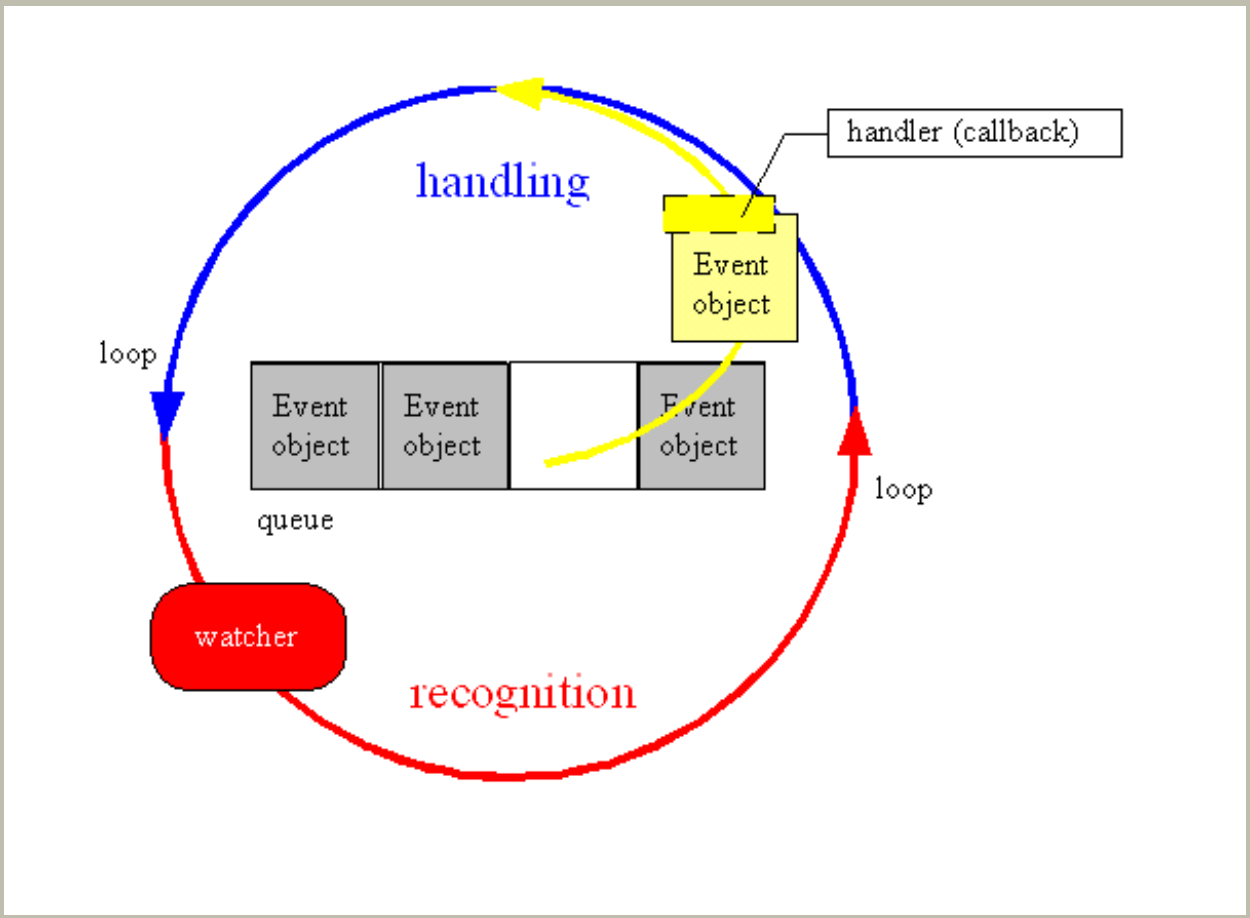
If you are building a priority hierarchy for various watchers, please keep in mind that even events of the lowest priority should finally be served. That's why an events priority has not only to reflect *importance* and *urgency* of its handling but has also to take care of its (probable) *frequency*.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

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.)

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.

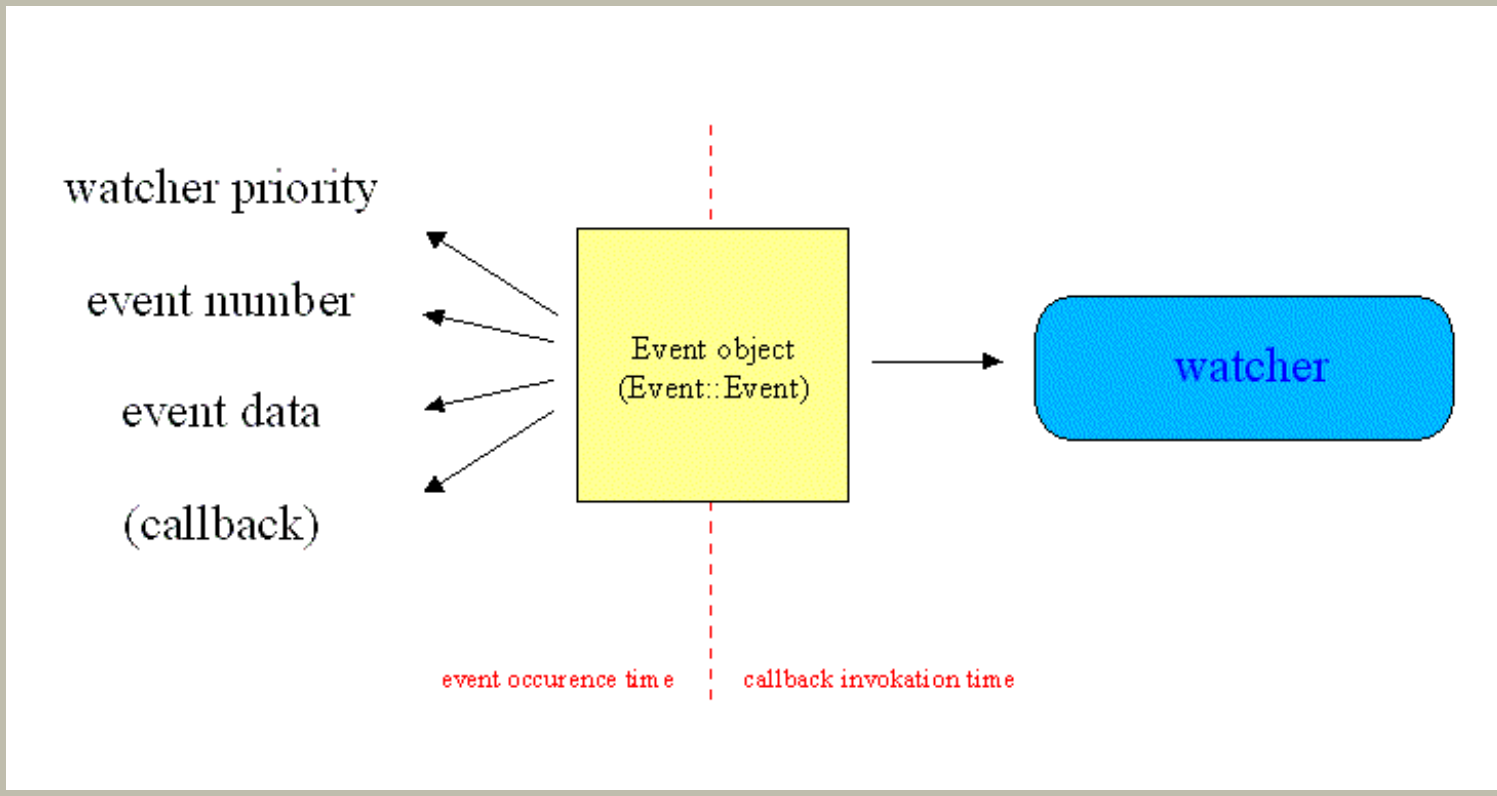


The `Event::Event` object is passed to the callback as its only parameter, this is managed automatically. Because it stores more informations besides the callback itself, it connects the callback with both the initial event and the watcher which detected the event and generated the order. Once the callback 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	informs how many triggers of the watched event should be
serviced by invocation of	the callback
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 definitely 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.

There is one exception to the rule that a callback runs exclusively: if a signal arrives while a callback performs `sleep()` or `select()` to delay itself, *these* functions will be terminated by the signal. Callback execution will then be continued after the `sleep()` or `select()`, regardless of the remaining period of delay. `If` you really need to implement such a delay in a callback, `Event::sleep()` can be used which is not interrupted by signals. But seen in the light of the previous paragraph it would be a better choice to avoid such delays at all.

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;
```

4. Watchers by example

This section is intended to introduce the several watcher types by discussing example code. This is not a complete review of all features but more a summary of experiences (and discussions at the mailing list).

As this is a "learning by doing" section, one does not have to be an experienced **Event** programmer to understand it. On the other hand, it surely does not repeat everything said in other sections. You can start right here but reading previous chapters may help.

In all examples, it is assumed that "`use strict`" is in action and **Event** is loaded, so the following is an "invisible startup part" of all code sequences shown:

```
# set pragma
use strict;

# load modules
use Event;
```

If anything more is required, it will be mentioned in the examples.

4.1. Signal watchers

They seem to be the simplest type of watcher because there is only a very limited and predefined event set. More, probably every Perl programmer is familiar with `%SIG` which makes it easy to imagine what signal handling basically means. Or, in other words, the following should be intuitive:

```
1: # install signal watcher
2: Event->signal(
3:               signal => 'INT',
4:               cb     => sub {warn "Why do you want to kill me?\n";},
5:              );
6:
7: # start loop
8: Event::loop;
```

Well, right. This script installs a watcher for the `SIGINT` signal. Everytime such a signal arrives (e.g. when a user types CTRL−C), the callback function should be invoked. In this case, the callback only displays a message.

To do this, we build and install a new watcher (lines 2 to 5). `Event->signal()` is a constructor which does this for us. Here it takes two configuring parameters: `signal` specifies the signal to be watched, and `cb` is used to set up the callback.

The *signal* is passed by a string. "INT" specifies `SIG`**`INT`**, "PIPE" points to `SIG`**`PIPE`** and so on. This is very similar to the usage of `%SIG`.

As with `%SIG`, it is impossible to catch `SIGKILL`.

The *callback* is specified by a code reference. There are several other ways to set it up which will be discussed in the subsequent section.

Knowing `%SIG` you may ask if signal watcher callbacks are special by any means, and if care should be taken to make them reentrant (another signal may arrive while the first handler is running). But this is simplified with **Event**: *once the handler is running, it runs* regardless of all events, including more signals. New signals are detected independently of a running callback and will be queued until they can be served by another callback invokation. Run this example to see it working:

```
# set pragma
use strict;

# load module
use Event;
```

```
install signal watcher
Event->signal(
              signal => 'INT',
              cb     => sub {
                              warn "Detected ", $_[0]->hits, " SIGINT events.\n";},
                              warn "Sleeping now!\n";
                              # delay
                              Event::sleep(10);
                              warn "Slept.\n"},
            );

# loop
Event::loop;
```

Once the callback is running, subsequently arriving `SIGINT` signals will not cause it to stop. Different to this, they are *collected* in the event queue until the next callback for them can be started. **Event** will then invoke *one callback for all queued* SIGINT *signals*, passing their number in `hits`.

People familiar with `%SIG` may remember two special conventions:

```
$SIG{HUP}="DEFAULT";
```

and

```
$SIG{INT}="IGNORE";
```

both have a special meaning: they do *not* install `main::DEFAULT()` or `main::IGNORE()` as event handlers (which would be the case for every other string assignment) but declare that the signals should be handled "as usual" (`"DEFAULT"`) or should be completely ignored (`"IGNORE"`). Now, are there similar features built into **Event**?

The answer is that there is no need for such special semantics because you can manage signal watchers a more consistent way to behave likewise if you wish.

For example, if you want your signal to be served by perl as usual while there is a watcher registered for it, you can change the watchers state into **INACTIVE**. So, to get a result similar to

```
{
 # temporarily reenable default handling
 local($SIG{INT})='DEFAULT';

 # do something
}
```

with **Event** you can write

```
{
 # temporarily reenable default handling
 $signalWatcher->stop;

 # do something

 # reactivate custom handling
 $signalWatcher->again;
}
```

If you want a signal to be *ignored*, just install an empty callback:

```
# install signal watcher
Event->signal(
              signal => 'INT',
              cb     => sub {} ,
             );

# start loop
Event::loop;
```

Well, but often it's the same with ignoring as with default handling: there *are* installed signal handlers doing real things but they should be disabled *temporarily*. This case, the shown solution is not satisfying at first sight because the

installed ignorance is permanent. To work around this, you can temporarily exchange the stored callbacks:

```
{
 # temporarily ignore the signal
 my $cb=$signalWatcher->cb;
 $signalWatcher->cb(sub {});

 # do something

 # reactivate original handling
 $signalWatcher->cb($cb);
}
```

This seems to be the simplest way. Another idea is to have a *set* of watchers for the same signal from which exactly one could be chosen to be active at a given time. All others would been waiting in state INACTIVE. Having set up this, it could be managed that one of these set watchers ignores the signal, while others handle it really. See the subsequent discussion of multiple watchers for examples.

Finally, to sum up this comparison of `%SIG` and **Event**, it is not recommended to mix both models. It's surely possible to do this and may work in most cases, but it's inconsistently at least. But not only that. `%SIG` is less flexible, not scalable, offers less tuning features and *runs completely besides **Event***. Why missing the power and flexibility of **Event** without need?

Signal watchers have a very high priority by default to be served immediately. Actually it's the *highest* priority set by default (accessible by the constant `PRIO_HIGH`). This makes sense because most signals represent very important and urgent system messages. Nevertheless, if you want a signal to be served not that quickly, you can set up another priority. For example, user signals may be less important than `SIGINT`.

```
# install signal watcher
Event->signal(
              signal => 'USR1',
              cb     => sub {warn "Hi!\n";},
              prio   => PRIO_HIGH+1,
             );
```

Even with high priority, signals may arrive faster than they can be handled (because they have a high priority to the system as well). That's why a signal callback may be invoked only *once for a sequence* of similar signals which arrived almost "together" − that means, with a very high frequency.

4.2. Excurse: Callback setup

Now that we made first watchers, let's have a closer (and more general) look at callback setup. Callbacks are usually set up by a code reference:

```
$watcher->cb(\&callback);
```

Symbolic references are supported as well. This is known from `%SIG` where you can write

```
$SIG{INT}='package::callback';
```

Different to this, **Event** does *not* accept

```
Event->signal(
              signal => 'INT',
              cb     => 'callback', # does not work
             );
```

Event provides a more general way by accepting an *array reference*.

```
Event->signal(
              signal => 'INT',
              cb     => ['package', 'function'], # invokes package::function()
             );
```

The first array element names the package, while the second element is the function name. This means that the `'package'` string in the example above can be replaced by any package or class name of your choice which will

obviously result in invoking `package::function()` or `Class::function()` as the callback.

More, the first element in the anonymous array can also be an *object*:

```
Event->signal(
              signal => 'INT',
              cb     => [$object, 'method'], # invokes $object->method()
             );
```

which means that *the method `method()` of the passed object* is called as callback. The various ways of callback setup by array references are demonstrated in the following example:

```
 1: # declare helper class
 2: package OO;
 3:
 4: # class variable
 5: my $counter=0;
 6:
 7: # constructor
 8: sub new
 9:  {
10:   # get parameters
11:   my ($class)=@_;
12:
13:   # init object variable
14:   my ($me)=++$counter;
15:
16:   # make and supply new object
17:   bless(\$me, $class);
18:  }
19:
20: # demonstration method/function
21: sub display
22:  {
23:   # get parameters
24:   my ($me)=@_;
25:
26:   # report invokation method
27:   warn ref($me) ? "Called as method of object $$me (@_).\n"
28:                 : "Called as function (@_)\n";
29:  }
30:
31: # reply a true value to flag successfull init
32: 1;
33:
34: # main package
35: package main;
36:
37: # install watchers
38: Event->signal(signal=>'INT',  cb=>[new OO, 'display']);
39: Event->signal(signal=>'HUP',  cb=>[new OO, 'display']);
40: Event->signal(signal=>'PIPE', cb=>['OO',   'display']);
41:
42: # enter loop
43: Event::loop;
```

If the script receives `SIGINT`, the *first*`OO` object reports that its `display()` method was invoked. It is the same for `SIGHUP` which is handled by calling the `display()` method of the *second*`OO` object. Different to this, `SIGPIPE` is handled by the `OO`*class* method `display()`. Run the script to see yourself.

4.3. Timers

Timers are widely used to detect timeouts, make a process waiting for another, implement alarms and reminders and so on. But nevertheless it's not always that easy (and sometimes a little bit tricky) to implement them without event handling. The simplest task is to make a program wait for a certain period of time, which is usually done by calling `sleep()` or the three argument `select()`.

```
sleep(20);
```

```
select(undef, undef, undef, 30);
```

Reminders can obviously be realized by checking date lists periodically and mentioning all currently open dates.

```
while (%dateList)
 {
  # check date list
  foreach my $date (grep(time>=$_, sort {$a<=>$b} keys %dateList))
   {
    # report date and remove it from the list
    warn "[", formatDate($date), "] $dateList{$date}\n";
    delete $dateList{$date};
   }

  # wait ...
  sleep(60);
 }
```

Checking a code sequence timeout is trickier. There is an idiom which uses `eval()` together with `alarm()` to implement a timeout for a certain code block:

```
# This example is derived from perls documentation.

# perform code with timeout control
eval
 {
  local $SIG{ALRM}=sub {die "alarm\n"}; # NB: \n required
  alarm $timeout;
  ...
  alarm 0;
 };

# timed out?
if ($@)
 {
  # propagate unexpected errors
  die unless $@ eq "alarm\n";
  # timed out
 }
else
 {
  # no timeout
 }
```

Hm. Looking at these examples, I think that although there is a number of ways to make timers work without event handling, they are not very flexible sometimes. They are indeed sufficient for simple cases, but there is also a tendency to end in code which is not really easy to read and maintain when things become more complex. Imagine you have to combine a number of these timers, to cascade them (e.g. having to check a timeout for a code block and timeouts for several nested subblocks seperately) or to add additional tasks like user interaction ...

That's different with event handling, especially with **Event**. It is very easy to set up a timer for whatever purpose you want. It is done the same way everytime, so if you know how to set up a timer for a reminder, you know how to set up a timer for timeouts. So, by the way, here is a reminder.

```
# This example demonstrates the usage of Event
# timer watchers.
#
# Call the script this way:
# script <seconds to alarm> <event to remember (any text)>.

# load module
use Event;

# install timer
Event->timer(
             at => time+shift,
             cb => sub {
                       warn "Time to remember \"@ARGV\".\n";
                       $_[0]->w->cancel;  # clean up
```

```
                        }
            );

# enter loop
Event::loop;
```

All **Event** watchers are installed by `Event` methods which characterize the watcher type, so the constructors name is "timer" this case. The `at` attribute determines *when* the timer event shall happen, this point of time is passed in UNIX time format (seconds since 1970). Here it is calculated as summary of the current time and the user set period until the requested alarm.

Well, this scripts task could have been performed by a simple `sleep()` call as well! Surely, but add more dates to the list and watch them all together ... and the `sleep()` program quickly becomes complex! Not so with **Event**:

```
# install requested timers
while (@ARGV)
 {
  my ($period, $description)=(shift, shift);
  Event->timer(
               at => time+$period,
               cb => sub {
                          warn "Time to remember $description.\n";
                          $_[0]->w->cancel;  # clean up
                         }
              );
 }

# enter loop
Event::loop;
```

This is almost the same code and still intuitive and clear, but now a user can pass as many alarm requests as he wishes by providing a period / description pair for each alarm in the script call. (`"script 5 'the second alarm' 2 '1st'"`) An even more flexible reminder (for an unlimited number of dates and additional user interaction) with **Event** was simple enough to be used as an introduction example in "A complete example" above.

Note that in both examples `Event::unloop()` is not called. The loops will automatically terminate after the last mentioned alarm because every installed timer watcher dies after callback execution. Well, not necessarily ... but because we used `at` to specify the event. `at` installs *one shot timers*. This is different with `interval` which is an alternative attribute: timers set up by `interval` automatically *repeat*.

```
Event->timer(
             interval => 5,
             cb       => sub {warn "It happened (again).\n";}
            );
```

`interval` sets up the event time by an immediately beginning period (in seconds). The shown timer will detect its first event 5 seconds after installation, and then it will continue to work and detect events every 5 seconds.

Is there a way to install a timer with `interval` but make it a *one* shot watcher? Yes, of course: the `repeat` attribute just flags this behaviour.

```
Event->timer(
             interval => 5,
             repeat   => 0,
             cb       => sub {
                              warn "It just happened.\n";
                              $_[0]->w->cancel; # clean up
                             }
            );
```

This timer runs only *once*, exactly like an `at=>time+5` one.

Now that we combined `repeat` with `interval`, can we do the same with `at` to install *repeating* "at" timers? No. Repetition always requires an `interval` to be specified. A repeating timer without an interval will throw an exception.

Speaking about intervals, let's have a final look at the simple repeating watcher set up by `interval`. This time, the callback reports its invokation time.

```
# store startup time
my $startup=time;

sub callback
 {
  # report callback invokation
  warn "Event after ", time-startup, " seconds.\n";
  # have a rest
  sleep(10);
 }

B:Event->timer(
               interval => 20,
               cb       => \&callback,
              );

# enter loop
Event::loop;
```

What will this script report? Please note the `sleep()` call in the callback (I do not recommend this for real callbacks, of course ;−). Without any doubt, the sequence of messages will start by something like

```
Event after 20 seconds.
```

But then? After this report, the callback falls asleep for ten seconds. What will the next report say? This depends on the watchers restart time. By default, the timer is restarted *after callback execution*, and so the next message will arrive 50 seconds after script call (20 s until the first call plus 10 s sleeping − now the timer restarts − plus a new period of 20 s to the next event). This means that by default repeated timer events strongly depend on the watchers callback runtime. This may not be sufficient − especially with callbacks of a less determined runtime than a simple `sleep()` call provides. If you need timer events with a fixer frequency, the `hard` attribute can be used to enforce a repeating watcher to restart *before* callback invokation.

```
B:Event->timer(
               interval => 20,
               hard     =>  1,
               cb       => \&callback,
              );
```

Now the example script will report events every twenty seconds.

Finally, **Event** timers run independently on each other. They can be started, stopped and disabled asynchronously. One of the best things, in my opinion, is the fact that with **Event** timers can run besides very different watchers. The time until the timer event can be used. Timers can be *one* certain part of a complex system of user interaction, IPC, signal handling, periodically performed tasks and more. They play their part, but they do not enforce the rest of the systems code to be written a special way determined by the timers (as the `eval()/alarm()` construct does.) In short, **Event** timers provide a great flexibility and may be worth *alone* to design a script on base of **Event**.

```
I wrote a (client/server) system where
timers are dynamically installed and
started when a server interaction begins.
When a server times out to answer a request,
it is automatically disconnected by the (one
shot) timer. On the other hand, if the server
answers in time, the timer is simply cancelled
and destroyed by the related I/O watcher.
An unlimited number of such connections and
timers can be active at the same time, each
started  at an individual time and with a special
timeout.
At the same time, other timers perform periodical
tasks with user defined periods.
```

4.4. I/O watchers

I/O event watching is an everyday job. Not every program has timers, and signals are actively used only by a low percentage of all Perl scripts (I suppose). But almost *every* program implements I/O, at least to present results. In fact, most programs also get any data by I/O to work with. So, if we have an event driven program, we surely need support in this area as well. And of course it is provided by **Event**.

Well, which types of I/O handling are really an object of I/O event watching? Surely not the *presentation* of results I just mentioned above. This is an active part of a program, determined by the program flow, and a program *can* call the usual functions like `print()` to present data. Not even all *incoming* data are related to events, think of the reading of a configuration file which can be done the same usual way in an event driven application and in a common script. What's basically most interesting here is the kind of data that arrives *asynchronously* and *unpredictably* − user interaction, data arriving via IPC or income in a watched file which is written by another process. To detect these kinds of events, I/O watchers were designed. Let's start with file watching.

4.4.1. Watching files

```
01: # init file, then open file for reading
02: my $file='tailtest.tmp';
03: open(IN, $file) or die "[Fatal] Could not open $file.\n";
04:
05: # install "tail −f"
06: Event−>io(
07:           fd=>\*IN,
08:           cb=>sub {
09:                    # check for real news
10:                    return if eof(IN);
11:
12:                    # read new line and report it
13:                    print "New line detected: ", scalar(<IN>), "\n";
14:                   }
15:          );
16:
17: # start loop
18: Event::loop;
```

This is a reimplementation of `tail −f` using **Event**. (To see what happens, run the script and add something to the file "tailtest.tmp". Touch this data file initially unless it already exists.) Well, the basic structure of this program is already well known from the signal and timer examples above, it's always the same which makes it easy to use **Event** in all its various flavours. The `io()` method here installs a watcher for the handle passed to the `fd` attribute. An event is detected and the declared callback is invoked whenever something happens at this handle. "Something", by default, means that there is *something to be read* at the handle. This event "subtype" is described by the `poll` attribute whichs default value is "r" (for "read"), so that the watcher constructor above could also be equivalently written as

```
Event−>io(
          fd   => \*IN,
          cb   => sub {...},
          poll => 'r',
         );
```

There are more types of possible events at a handle: it can be *ready to be written*, or an *error* can occur. The related `poll` flags are "w" and "e".

```
# this watcher detects all kinds of io events
Event−>io(
          fd   => \*HANDLE,
          cb   => \&ioHandler,
          poll => 'erw',
         );
```

As to be seen here, various io event subtypes can be watched together. Their specification characters are just combined in the string value of `poll`. The disadvantage of this approach is that the callback has to determine first what really happened before it can operate appropriately. This detection can be performed using the `Event::Event` objects `got` attribute.

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

  # act event type dependend
  return ioError(@_) if $event->got eq 'e';
  return ioRead(@_)  if $event->got eq 'r';
  return ioWrite(@_) if $event->got eq 'w';
 }
```

or, alternatively

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

  # act event type dependend
  if ($event->got eq 'e')
   {
    # handle io errors
    ...
   }
  elsif ($event->got eq 'r')
   {
    # read from the handle
    ...
   }
  elsif ($event->got eq 'w')
   {
    # write to the handle
    ...
   }
  else
   {
    die "[BUG] Unexpected event type!\n";
   }
 }
```

Now, functions like these are a matter of individual preferences. If you want to avoid them here, you can also watch each of the various io event types by a separate watcher:

```
Event->io(fd => \*HANDLE, cb => \&ioError, poll => 'e');
Event->io(fd => \*HANDLE, cb => \&ioRead,  poll => 'r');
Event->io(fd => \*HANDLE, cb => \&ioWrite, poll => 'w');
```

These are *three* watchers working on *one* and the the same `HANDLE`. This way, every event on `HANDLE` can be handled by a specialized callback function. We have an overhead of two additional watchers now, but also the advantage of more readable and faster working callbacks.

People familiar with with the three argument `select()` will, by the way, note that the detectable io event subtypes are exactly the same that `select()` can recognize. The `select()` function is indeed comparable to io watchers as well as the basic signal handling via `%SIG` is to signal watchers.

Well, back to the `tail` example. Please have a look at line 10:

```
# check for real news
return if eof(IN);
```

I had to built this in because a *file* handle is always "ready to be read". This makes the script impressive as a demo but unusable in real life − we cannot determine if there is really new data or not, the watcher will always "detect" events even without news, and the result is traditional polling instead of real event handling (inspect the processor usage). But nevertheless, it works.

Here is a second version of the "tail" demo script which uses an additional timer to produce the detected new data itself. It by the way shows various watchers cooperating.

```
# load modules
use Event;
use FileHandle;

# declare and init variables
my ($c, $file)=(0, "tailtest.tmp");

# init file, then reopen file for reading
open(IN, ">$file");
open(IN, $file);

# install writer
Event->timer(
             interval=>5,
             cb=>sub {
                      # report action
                      print "Writing new line ", ++$c, ".\n";

                      # open file for writing, add a line, close the file
                      open(OUT, ">>$file");
                      OUT->autoflush;
                      print OUT "$c\n";
                      close(OUT);
                     }
            );

# install "tail -f"
Event->io(
          fd=>\*IN,
          cb=>sub {
                   # get filehandle and check for real news
                   my $handle=$_[0]->w->fd;
                   return if eof($handle);

                   # read new line and report it
                   print "New line detected: ", scalar(<$handle>), "\n";
                  }
         );

# start loop
Event::loop;
```

So, handles connected to real files are not the best candidates to be managed by io watchers. (A timer would be a better solution for this specific problem. There is also a murmur about new operating system kernel features currently developed which shall include file modification events. But this is still a sound of future.)

4.4.2. User interaction

Different to file connected handles, *IPC* handles like sockets or pipes, or handles connected to a terminal, are excellent candidates to be watched by io watchers. If you watch a socket or pipe for reading, an event will not be detected until data *really* comes in. But let's have a look at the most common thing in this area first, which is doubtlessly user interaction via a terminal bound to `STDIN`. A first version is quickly designed:

```
# install an io watcher to wait for user input
Event->io(fd=>\*STDIN, cb=>\&user);
```

Assumed that `STDIN` is configured the usual way, which means that it is buffered and blocking, this watcher waits on `STDIN` until a user has completed a line of input. It then passes the line to the `user` function, which could handle it as follows:

```
01: # handle a user command
02: sub user
03:  {
04:   # get the event object
05:   my ($event)=@_;
06:
07:   # get the handle
08:   my $handle=$event->w->fd;
09:
```

```
10:   # read the users input
11:   my $cmd=<$handle>;
12:
13:   # and handle it
14:   chomp($cmd);
15:
16:   # do something
17:   print '[Info] Performing your command "', $cmd, "\" ...\n";
18:   sleep(5);
19:
20:   # display a new prompt
21:   print "input> ";
22:  }
```

Note that the input still has to be read (line 11), the watcher only detected its arrival.

Getting the file handle from the event object (line 8) instead of reading `STDIN` directly makes this handler more flexible and usable for various handles.

`STDOUT` should be set to unbuffered mode to display new prompts immediately (line 21).

Well, there is nothing spectacular in this handling, but nevertheless it's worth to be mentioned. At first sight, it just reimplements a usual synchronous user interface by using **Event** which first waits until a command is entered, and makes the user wait then for command proceeding. It even copies the feature that new commands can be entered while a callback is still running. But internally, thanks to event handling the script behaves different. It's working asynchronously, the users command is only one event among others, and while the system is waiting for this special input, other events can be handled as well.

For example, informations could be periodically collected from somewhere while a script waits for new commands.

What's to say about such an interface? First, the most important callbacks are not the ones invoked by the interface io watcher but all the others. Most users are familiar with a synchronous interface and will accept that it takes time to perform their commands, but almost no user accepts a *slow* interface. This means that all the callbacks besides the interface one should return very fast.

4.4.3. A multiclient server

Now, let's go to another example. In client/server programming there are two main approaches to build a server. First, a server can handle exactly one connection a time, making subsequently connecting clients waiting in a queue. This is a one process architecture, easy to implement and sufficient for very short communications. If connections need to be established for a longer time, or clients cannot wait until their requests are answered, the server is usually built in a multi process architecture. The main process accepts new connections, immediately starts a new subprocess and delegates the further communication to it, quickly returning to accept the next request. One disadvantage of this multi process approach is that operating systems often limit the number of (sub)processes and by the way the number of simultaniously connected clients. More, it's difficult to implement communications between the several processes. Well, most servers (like HTTP or FTP ones) do not really need subprocess synchronisation because every connection is performed independently of each other.

But nevertheless connection synchronization can be very useful − to synchronize commands which may be given by various clients but should be executed sequentially, to control limited resources and their usage by the clients / connections, to accept new clients depending on the state of established connections, to pass data back from a connection handler to the main process, to pass data between clients (imagine IRC), to log connection informations in a central logfile or to let the main process control connections by special commands. All this would be easy in a combination of the two main approaches: if we had only one process, data sharing would be simple, but we should be able to handle several connections simultaniously. Well, not surprising, event handling provides a way to do this.

```
01: # set pragma
02: use strict;
03:
04: # load modules
05: use Event qw(loop unloop);
06: use IO::Socket;
07:
08: # globals
```

```
09: my $channels;
10:
11: # make socket
12: my $socket=IO::Socket::INET->new(Listen=>5, LocalPort=>8001, Reuse=>1);
13: die "[Fatal] Cannot build initial socket.\n" unless $socket;
14:
15: # install an initial watcher
16: Event->io(
17:           fd => $socket,
18:           cb => sub {
19:                     # make "channel" number
20:                     my $channel=++$channels;
21:
22:                     # accept client
23:                     my $client=$socket->accept;
24:                     warn "[Error] Cannot connect to new client.\n" and return unless $client;
25:                     $client->autoflush;
26:
27:                     # install a communicator
28:                     Event->io(
29:                               fd => $client,
30:                               cb => sub {
31:                                         # report
32:                                         warn "[Server] Talking on channel $channel.\n";
33:
34:                                         # read new line and report it
35:                                         my $handle=$_[0]->w->fd;
36:                                         my $line=<$handle>;
37:
38:                                         # talk
39:                                         $handle->print("[Channel $channel] $line");
40:
41:                                         # quit if wished (checking \r for telnet clients)
42:                                         if ($line=~/^quit\r?$/i)
43:                                           {
44:                                            $_[0]->w->cancel;
45:                                            warn "[Server] Closed channel $channel.\n";
46:                                           }
47:                                        },
48:                              );
49:
50:                     # welcome
51:                     $client->print("Welcome, this is Channel $channel.\n");
52:
53:                     # report
54:                     warn "[Server] Opened new channel $channel.\n";
55:                    },
56:          );
57:
58: # start loop
59: loop;
```

This script is a multi client server but built in one process. (Please start the server and connect to port 8001 by telnet to try this example.) It starts like usual servers by making an initial listening socket for connection requests (line 12). Then it installs an I/O watcher to look after this sockets. Note that this watcher does not block the process, it's only watching for data arriving at the socket. Most servers block at this point because they call `accept()` to wait for connections, but we can delay the call of this function because **Event**s *watcher* will detect connection requests for us (a non blocking way).

If a client tries to connect, data is sent to the initial socket, the watcher detects the I/O event and the callback is invoked. All data is still in the sockets queue. Now, in the callback, `accept()` is called (line 23). It certainly blocks nothing because at this point it's clear that there *is* a request. The `accept()` call makes a new socket, redirects the new connection to it and supplies the socket. Then, instead of making a new subprocess talking on this socket, a further I/O watcher is installed to watch the new socket (line 28). If the client sends data now, it's dynamically made connection watcher invokes the communication callback. This callback, in our example, simply sends back all input to the client and can terminate the connection on the clients request.

Note that the server is still in control of the new connection socket. It sends a welcome message to the client after installing the connection watcher (line 51), and it would be very easy to store the connection socket at a global place

to make it accessible by the whole program. On the other hand, this would complicate the connection handling because we would have additional references to the socket, thus making it staying alive even when its communication watcher is cancelled, so that the connection would remain open. It depends on the application if it is usefull to install a central cleanup facility or close the socket implicitly by cancelling the connection watcher.

Once the new watcher is installed, we now have two sockets listening and a watcher for each of them. There is still the initial socket with a watcher ready to accept more clients, and the client socket whichs watcher waits for client commands. More clients may connect, the number of connections is limited only by the number of sockets the system can handle at a time. If several clients send commands, their connection callbacks are in fact invoked subsequently, but from a users point of view, it seems that the server handles all connections simultaniously. Well, if the callbacks return quickly. Make sure they do.

A certain aspect of callback return is to return `ever`. This means: *avoid blocking callbacks*, especially in servers! From this point of view, the example server is on a good way because the client socket is made autoflushing immediately (line 25), but it's surely not perfect because it uses blocking I/O to read from the client socket (line 36). Honestly spoken, the <> operator is not very suitable this place. It all works fine with a line based protocol like FTP where every command is trailed by a newline, or if a client like telnet is used which sends commands line by line, but imagine that a (self written?) client accidentally "forgets" to send this special character. This will result in blocking at line 36, and it will not only block the callback but the complete process and thus all established connections and the initial socket watcher. For this reason, I recommend to use nonblocking sockets and to build such servers on base of a *stream protocol*. These protocols do not need certain characters or strings to determine when a message is completed. The CPAN module **IPC::LDT** implements one of them. It automatically unblocks sockets while sending or receiving data and well cooperates with **Event**.

4.5. Excurse: Passing user data

Not surprisingly, watcher attributes are intended to store *event* specific data. But sometimes it is useful to store *application* specific data in a watcher object as well.

```
A server may dynamically make watcher objects
for each client connection. Connection data
such as client host and port or the client users
account (got via identd, for example) are usually
detected when the connection is established. But
they could be of interest in the callback as well,
even more if the callback should be fast and it
would be a waste of time to collect these informations
again (via the socket). Additionally, such data
lose importance after the related connection is
closed. So it seems to be a good idea to store
these data within the watcher object. Each callback
can access it there, and it will be automatically
destroyed with the watcher.
```

For this purpose, **Event** provides a special "attribute" named `data`. It is used like any other attribute, which means that it can be initially set by a constructor parameter:

```
# now that we accepted the client,
# make a new watcher to serve it
Event->io(
          desc => "client at $host/$port",
          fd   => $clientSocket,
          cb   => \&serverClient,
          data => $perl,
          );
```

The "perl" setting in this example may flag whether a client is written in Perl and understands serialized data or not. This could be an information provided in the connection handshake.

Now, in the callback, we can retrieve the stored informations by using the `data()` method without arguments:

```
# send result to client
if ($event->w->data)
 {
  sendResult($event->w->fd,
```

```
             serialize($result),
            );
 }
else
 {
  sendResult($event->w->fd,
             makeNestedAsciiList($result),
            );
 }
```

But maybe a client decides on the fly that it is *now* able to handle serialized Perl data. Well, we can modify the stored data at any time by passing the new value as an `data()` argument:

```
# client wishes to receive serialized data now
$event->w->data(1);
```

A certain watcher object has always exactly *one* `data` attribute. But this is no limit − using references you can store whatever you want:

```
# make a new watcher to serve the client
Event->io(
          desc => "client at $host/$port",
          fd   => $clientSocket,
          cb   => \&serverClient,
          data => {
                   perl => $perl,
                   host => $host,
                   port => $port,
                  },
          );

...

# send result to client
if ($event->w->data->{perl})
 {...}
```

Note: The `data` attribute is intended for *users*. If you want to construct *a subclass which inherits* from the watcher class and are looking for a way to store class specific data, `private` is the attribute to use.

4.6. Idle watchers

```
# This example demonstrates the usage of Event
# idle watchers which can perform tasks when
# no other action is required. Here the idle
# watcher solves a math problem whenever possible.
#
# To display calculation process, simply type
# something. The script reacts and illustrates
# that other watchers are running as well.

# set pragma
use strict;

# constant
use constant VALUE => 0.001;

# load modules
use Event qw(loop unloop);

# install idle watcher
my $idle=Event->idle(
                     data     => VALUE,
                     cb       => sub {
                                     # perform complex scientific calculation
```

```
                                    $_[0]->w->data($_[0]->w->data+VALUE);
                                   },
                     repeat   => 1,
                    );

# install a simple input watcher
my $io=Event->io(
                 fd => \*STDIN,
                 cb => sub {
                           # show that you are here
                           warn "Well, what's on? The intermediate result is ", $idle->data, ".\n";
                           # read
                           <STDIN>;
                          },
                );

# start loop
loop;
```

4.7. Variable watchers

```
# Simple Event example using var watchers.
#
# It demonstrates a read access watcher,
# a write access watcher and a combined one.
#
# Please note how difficult read access
# watching is. Almost noone can predict
# how often such event will happen. Use
# read access watchers very carefully.


# set pragma
use strict;

# load module
use Event;

# install variable
my $var;

# init random generator
srand (time ^ $$ ^ unpack "%L*", `ps axww | gzip`);

# install writer
Event->timer(
             interval=>2,
             cb=>sub {
                      # check what is to do
                      if (rand(10)<5)
                        {
                         # now modify the variable (multiple access!)
                         $var++;

                         # report action
                         print "----> Modified variable.\n";
                        }
                      else
                        {
                         # only READ variable (1 access)
                         my $value=$var;

                         # report action
                         print "----> Read variable.\n";
                        }
                     }
            );

# install read var watcher
Event->var(
           var  => \$var,
```

```
           poll => 'rw',
           cb   => sub {print "[1 (r )] Read!\n";},
         );

# install write var watcher
Event->var(
           var  => \$var,
           poll => 'w',
           cb   => sub {print "[2 ( w)] Written: new value is ", ${$_[0]->w->var}, ".\n";},
         );

# install combined var watcher
Event->var(
           var  => \$var,
           poll => 'rw',
           cb   => sub {
                        # read access?
                        print "[3 (rw)] Read!\n" and return if $_[0]->got eq 'r';

                        # write access: read new value and report it
                        print "[3 (rw)] Written: new value is ", ${$_[0]->w->var}, ".\n";
                       },
         );

# start loop
Event::loop;
```

4.8. Group watchers

```
# This example demonstrates the usage of Event
# group watchers. Here the watcher watches an
# io watcher and a timer, and if none of them
# acts in a certain period of time, the group
# watcher invokes its callback.
#
# To illustrate this, the timer starts with a short
# interval and increases it in every callback.
# The io watcher listens at STDIN so the user can
# prevend the group watcher event by typing.

# set pragma
use strict;

# load modules
use Event qw(loop unloop);
require Event::group;

# install a simple input watcher
my $io=Event->io(
                 fd => \*STDIN,
                 cb => sub {
                            warn "Io here.\n";
                            # read
                            <STDIN>;
                           },
                );

# install a timer
my $timer=Event->timer(
                       interval => 1,
                       cb       => sub {
                                        # increase interval
                                        $_[0]->w->interval($_[0]->w->interval+1);
                                        warn "Timer here, next call in ", $_[0]->w->interval, " seconds.\n";
                                       },
                      );

# install group watcher
my $group=Event->group(
                       add     => $io,
                       timeout => 5,
```

```
                        cb      => sub {
                                    #
                                    warn "Action detected!\n";
                                   },
                        );
$group->add($timer);


# start loop
loop;
```

5. Advanced features

5.1. 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 −D*EVENT_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::Portal** 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::Portal

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::Portal
    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::Portal::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

%
```

NetServer::Portal also supports remote symbol table inspection.

5.2. 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::Portal** 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. Customization

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

The `private` attribute is designed to help developers building their own watcher subclasses.

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.

The *C API* provides access to the internals and is introduced in a chapter of its own below.

5.4. 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**.

```
Joshua 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 (`readline.pl`).

6. Using the C API

Event comes with a special API which allows to write fast callbacks in C. This chapter provides a first overview about this feature and is by no means complete. It is assumed that a reader is familiar with the Perl usage of **Event**.

Note: Please read the **Event::MakeMaker** documentation carefully (it is part of the **Event** package). ***The C API may be subject to change.*** This chapter is based on **Event 0.80**.

C programming in conjunction with Perl is usually associated with things like **XS** or **SWIG**. Different to this, I've decided to use the new **Inline** module for the examples in this chapter. It automatically performs XS internals in the background, thus allowing to directly embed C into Perl (right into a script or module), which makes the code shorter and much more intuitive. Additionally, it enables immediate beginning in the field of mixed Perl/C programming even for an unexperienced XS programmer (like me at the moment). All this predestinates it for tutorial purposes.

Besides pure tutorial considerations **Inline** (>=0.30) and **Event** (>=0.80) directly support each other which makes using the C API even simpler!

Note: The **Inline** module is still declared *beta software*. Nevertheless, the module is stable and all examples in this chapter are tested to run. The writing is based on **Inline 0.30** (this version (or higher) is ***required*** to run the examples).

Well! Having said all this, let's start using the C API. What is it for? It shall simply accelerate event handling. This can be done in one of three ways: by invoking C callbacks from Perl watchers, by combining Perl callbacks and C watchers, and by bypassing Perl completely. C callbacks accelerate the users event handler. C watchers accelerate event handling itself. Let's see.

6.1. Preparations

Install **Event** (>=0.80). ;–) Install **Inline** (>=0.30).

6.2. Perl watcher and C callback

In this model, the watcher is installed, initialized and maintained on Perl level, while the watchers callback is a C function. This is useful for all who need to write fast callbacks, but want to keep **Events** easy to use Perl interface.

As many callbacks do not necessarily need to access the event data which is passed to a callback, the simplest form of a C callback does not have to use **Event**'s C API at all:

```
01: # pragma
02: use strict;
03:
04: # load modules
05: use Inline with=>'Event';
06:
07: # declare the C part
08: use Inline C => <<'EOC';
09:
10: void c_callback(pe_event * event)
11:  {
12:   printf("Here is the C callback!\n");
13:  }
14:
15: EOC
16:
17: # install a timer which calls the C callback
18: Event->timer(
19:              desc     => 'Perl timer with C callback.',
```

```
20:              interval => 2,
21:              cb       => \&c_callback,
22:             );
23:
24: # start the loop
25: Event::loop;
```

Not absolutely necessary in this example but generally good advice is to `use strict` (line 2) when dealing with **Inline** to be warned if perl cannot resolve a function name.

Event and **Inline** are load *together* by the `use` statement in line 5. The special syntax `with => 'module'` commands **Inline** to automatically search for and load the named module if it finds out that this module is prepared to cooperate with **Inline**. Well, **Event** is *well* prepared for this interaction so it is accepted by **Inline** and load.

Don't care that `use Inline` is called again in line 8. This case the syntax `language => string` passes code of the specified `language` in `string` to be processed by **Inline**. *This means that the C source is transparently embedded by a simple string!* For reasons of readability the C string in this example is a here document. If you do not want to patch embedded C by Perl (at compile time!) it is suggested to use single quoted strings.

All callback functions are of the prototype **`void (pe_event *)`** (line 10). We will investigate this interface soon, for now, we just use it because the example above runs a callback which works regardless of its arguments.

Do not declare the callback `static` which would make it unusable on Perl side.

The callback body (line 12) may be replaced by anything you want.

Timer installation (lines 18 to 22) and loop startup (line 25) are performed as usual on Perl side. The C callback function `c_callback()` is made available by **Inline** so we can reference it like a Perl subroutine.

Now the script can be called as usual, and voilá: your watcher invokes a C callback!

The first time you execute the script, **Inline** will arrange the compilation and integration for you. In all subsequent calls, this step is passed by, so there will be no further delay. The callback function is integrated into the current namespace, which is `main` in this case.

Ok, that's fine, but often we *need* access to event data. That's where the callback interface becomes important.

Remember the callback interface on *Perl* side: a callback is not expected to return anything, and it receives an **Event::Event** object which provides access to event and watcher data. The C expression of such a subroutine is a `void` function taking a pointer to a `pe_event` structure:

```
void c_callback(pe_event * event) {...}
```

`pe_event` is the C structure corresponding to an `Event::Event` object (except for io events, which have an own structure named `pe_ioevent`). So whenever an `Event::Event` object is passed to a callback we automatically receive a `pe_event *` on C side to deal with.

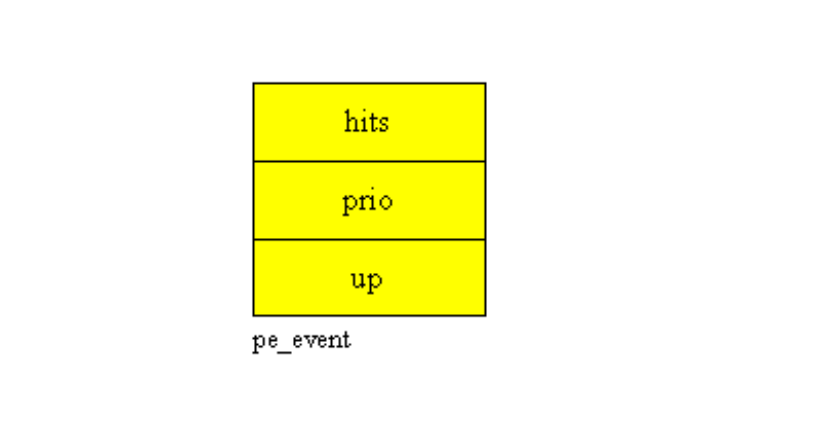
Note: People familiar with XS programming will notice that things are more complicated. To transform a Perl data structure into a C equivalent a *mapping* (or translation) has to be performed because perls internal data representations on C side are very special (see *perlapi* (http://www.perldoc.com/perl5.6/pod/perlapi.html) for details). (For example, a number is not stored in a C `int` variable but in an SV structure designed to represent all kinds of scalar values Perl knows, so one would have to transform that SV into an `int` first.) An interface programmer therefore usually has to provide and use typemapping code − but we are *completely relieved* from this task because of **Inlines** transparent integration with **Event**. By executing `use Inline with=>'Event'` **Inline** accesses typemapping informations and functions provided by **Event** and transparently integrates them into the Perl/C interface.

The result of this is very readable code − we can just access the passed data. Replace the callback function of the first example by

```
void c_callback(pe_event * event)
 {
  /* do something */
  printf(
         "Here is the C callback.\nI detected %d. events.\nThe events priority was %d.\n\n",
         event->hits,
         event->prio
        );
 }
```

As a first example, the events `hits` and `prio` data are reported. We could do it in Perl before and can do it in C now as well!

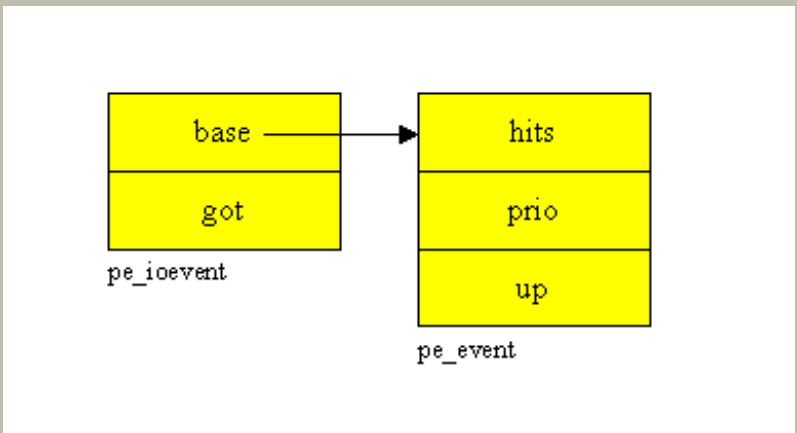
Because the `pe_event` structure represents an `Event::Event` object, each Perl side attribute has a counterpart here. Most important besides `hits` and `prio` we get access to the *watcher*.



Event::Event attribute counterparts in pe_event structures

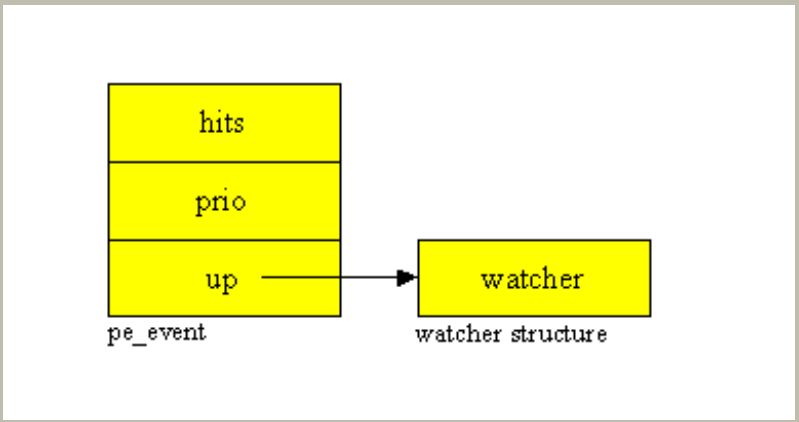
attribute	*pe event member*	*data type*
hits	hits	I16 (integer)
prio	prio	I16 (integer)
w	up	pointer to a structure which type depends on the watchers type

You may have missed the `got` data which is available if the watcher provides a `poll` attribute, like io watchers do. Well, io events are passed not by `pe_event` but by a special structure `pe_ioevent` just for this additional value:



The `got` value in `pe_ioevent` is an U16 integer.

Very similar to the Perl side, we find the watcher embedded into event data. As we are on C side now, this is a pointer to just another data structure.

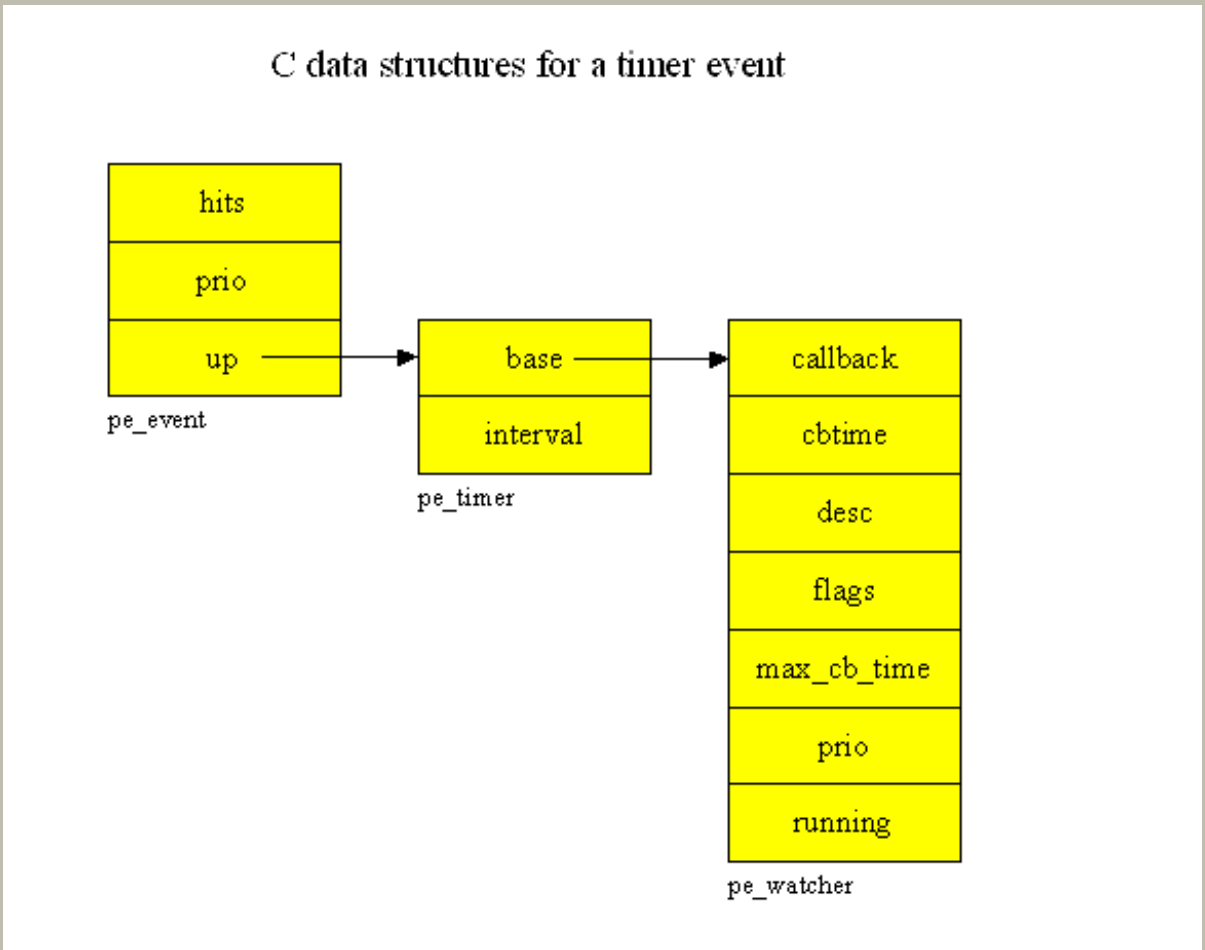


It depends on the type of the watcher which kind of structure this is, every watcher has its own. (So it is important to know which watchers shall invoke a certain callback, but this was already true on Perl side as well.)

watcher data structures

watcher type	***data structure***
io	**pe_io**
timer	**pe_timer**
signal	**pe_signal**
idle	**pe_idle**
var	**pe_var**
group	**pe_group**

Regardless of their special type, all these structures have a similar architecture. They contain an element named `base` which points to one more structure of type `pe_watcher`, holding data common to all watcher types. Additionally, there are individual structure elements for the type specific attributes.



To illustrate this, here is an example that accesses various watcher data.

```
01: # pragma
02: use strict;
03:
04: # load modules, process C code
05: use Inline with => 'Event';
06: use Inline C    => 'DATA';
07:
08: # install a timer which calls a C callback
09: Event->timer(
10:              desc     => 'Perl timer with C callback.',
11:              interval => 2,
12:              cb       => \&c_callback,
13:             );
14:
15: # start the loop
16: Event::loop;
17:
18: __END__
19:
20: __C__
22:
23: /* This is the C part. */
24:
25: void c_callback(pe_event * event)
26:  {
27:   /* get the watcher */
28:   pe_timer * watcher=(pe_timer *)event->up;
29:
30:   /* inspect base watcher attributes */
31:   printf(
32:          "Watcher (common attributes):\nmax_cb_tm: %d,\nprio: %d,\ncbtime: %f,\nis_running: %d.\n\n",
33:          watcher->base.max_cb_tm,
34:          watcher->base.prio,
35:          watcher->base.cbtime,
```

```
 36:          watcher->base.running
 37:         );
 38: }
```

In this example a new technique is used to store the C code: `use Inline C=>'DATA'` (line 6) lets **Inline** search for the source via Perls pseudo handle `DATA`. This allows to clearly separate Perl and C side which can make a script more readable. Whenever **Inline** is invoked this way it will continue reading behind the `__END__` mark, searching for a `__LANGUAGE__` mark matching the specified language. In the example above, this is `__C__` (line 20) because Inline was invoked by `use Inline C=>'DATA'`. The text after this mark is read until the next mark or the end of file and interpreted as embedded source code of the specified language.

This flexible concept allows to use the `DATA` "file" for embedded sources in various languages or even several fragments in the same language to be processed differently.

On C side now we first get access to the watcher (line 28), according to the data structure described above. The cast in this line helps to avoid compiler warnings about "incompatible pointer types".

Once we have the watcher we can access its data. This is done in lines 33 to 36 and seems to be straight forward − assumed one knows the data types. So here is a table describing these data more detailed (corresponding with the related Perl attribute table in a chapter above).

base watcher attributes on C side

Perl side attribute	**pe_watcher element**	**access in C**
unlimited access:		
cb	void * callback	
debug	flags	`WaDEBUG(watcher)` to query, `WaDEBUG_on(watcher)` and `WaDEBUG_off(watcher)` to modify
desc	SV * desc	`SvPVX(desc)` replies `char *`
max_cb_tm	I16 max_cb_time	as integer
prio	I16 prio	as integer
reentrant	flags	`WaREENTRANT(watcher)` to query, `WaREENTRANT_on(watcher)` and `WaREENTRANT_off(watcher)` to modify
repeat	flags	`WaREPEAT(watcher)` to query, `WaREPEAT_on(watcher)` and `WaREPEAT_off(watcher)` to modify
access limited to certain watchers:		
hard	flags	`WaHARD(watcher)` to query, `WaHARD_on(watcher)` and `WaHARD_off(watcher)` to modify
read−only access:		
cbtime	double cbtime	as double
is_running	IV running	as integer

Note that several boolean values are stored together in a structure element `flags` which should be accessed only by the macros mentioned in the table. (These macros are defined in an **Event** API header file which is automatically included by **Inline**.) The flags are combined on bit level (every setting has its own bit). The query macros (`WaDEBUG`, `WaREPEAT`, ...) work very perlish: for a set flag, anything different to `0` is replied, and `0` otherwise.

```
For example, to make a timer repeating, use

     WaREPEAT_on(timer);
```

```
assumed that timer is the watcher data structure.
```

One of these flags is `hard` which is known to be available for certain watchers only (see below). Now on C side, we see that nevertheless this flag is stored together with all the others.

The data marked as "read only" are of course not read only in C but in Perl, but it is a wise decision not to touch them on C side as well, because setting a wrong value may make **Event** crash.

All these base attributes are to be found in the `base` structure of *each* watcher, while the following attributes are directly embedded into the appropriate special watcher data structures.

specific watcher attributes

attribut	***pe watcher element***	***access in C***
pe_io:		
fd	int fd	
poll	U16 poll	
timeout	float timeout	
timeout_cb		
hard	this value is stored in the base structure	via base structure
pe_timer:		
at	double tm.at	as double
interval	SV * interval	`GEventAPI->sv_2interval("", interval, &period)` replies a `double` in `period`, `newSVnv(2.0)` makes a new SV holding a `double` which can be assigned to `interval`
hard	this value is stored in the base structure	via base structure
pe_signal:		
signal	IV signal	as integer
pe_idle:		
max	SV * max_interval	
min	SV * min_interval	
pe_var:		
var	SV * variable	
poll		
pe_group:		
timeout	SV * timeout	
add		

```
Note: The empty cells shall be filled soon. If you need details now, just refer to the
appropriate watcher types C implementation in subdirectory c, timer.c for example.
Search for the WKEYMETH declaration of the attribute of interest. All these
declarations consist of two types: the first sets a new attribute value, if a new
value is passed to the macro. Then it supplies the new (or just found) attribute
```

value. This is the part you should study. The second part is just for passing the attribute value to Perl because WKEYMETH functions are intended to be called from Perl side.

Walking through these tables you may have noticed that not all C data are as simple as `prio`, `cbtime` and `running` used in the last example: several of the data elements are *Perl* data types (like SV *). These data need special conversions before they can be used in a plain C context − and different to the callback parameters which were transparently transformed by **Inline** we have to do this now ourselves.

There are two kinds of Perl data here: data types of Perl itself and types declared by **Event**. Perls own data types can be converted to C data (and vice versa) by macros described in `perlapi` (see http://www.perldoc.com/perl5.6/pod/perlapi.html if you run a perl prior 5.6).

Event, fortunately, comes with all the mapping functions we need for **Event** data types. More, if used with **Inline**, both modules cooperate a way that there is automatically a structure `GEventAPI` which provides a number of pointers to the API functions. This structure is our simple gate to the API calls. The following summary example demonstrates how to use it:

```
01: # pragma
02: use strict;
03:
04: # load modules, process C code
05: use Inline with => 'Event';
06: use Inline C    => 'DATA';
07:
08: # install a timer which calls a C callback
09: Event->timer(
10:              desc     => 'Perl timer with C callback.',
11:              interval => 2,
12:              cb       => \&c_callback,
13:             );
14:
15: # start the loop
16: Event::loop;
17:
18: __END__
19:
20: __C__
21:
22: /* This is the C part. */
23:
24: void c_callback(pe_event * event)
25:  {
26:   pe_timer * watcher;
27:   double interval;
28:   static int calls=0;
29:
30:   /* get the watcher */
31:   watcher=(pe_timer *)event->up;
32:
33:   /* inspect Event data: event */
33:   printf(
34:          "Event:\nhits: %d,\nprio: %d.\n\n",
35:          event->hits,
36:          event->prio
37:         );
38:
39:   /* inspect Event data: base watcher attributes */
40:   printf(
41:          "Watcher (common attributes):\ndesc: %s,\nmax_cb_tm: %d,\nprio: %d,\ncbtime: %f,\nis_running: %d.\
41:          SvPVX(watcher->base.desc),
42:          watcher->base.max_cb_tm,
43:          watcher->base.prio,
44:          watcher->base.cbtime,
45:          watcher->base.running
46:         );
47:
48:   /* get interval */
49:   GEventAPI->sv_2interval(
50:                           "label",
```

```
51:                          watcher->interval,
52:                          &interval
53:                         );
54:
55:   /* inspect Event data: specific watcher attributes */
56:   printf(
57:          "Watcher (timer attributes):\ninterval: %f.\n\n",
58:          interval
59:         );
60:
61:   /* stop repeating after several calls */
62:   if (++calls>10)
63:    {
64:     WaREPEAT_off(timer);
65:     printf("Timer stops repeating.\n");
66:    }
67:  }
```

Lines 49 to 53 show how an API function is called to map data. Please refer to the tables above for details on calling other API functions.

Line 64 uses an API macro to modify a timer flag.

Not all mapping has to be done for **Event** data − line 41 demonstrates a transformation by a *Perl* API macro.

To do something similar for other watcher types, the appropriate structures and structure elements have to be used according to the tables above.

6.3. C watcher, C callback

In this "pure" solution the event handling is done on C side almost completely. (We do not manage the loop ourselves, this shall be done on Perl side still.) Doing so presumes more knowledge both of Event and Perl internals, and you will have to read the **Event** sources at a certain point. The advantage of this approach is a maximum of speed. Let's walk through an example.

The Perl side of the script is now very small, we just embed C, let the C side install a timer and start the loop.

```
01: # pragma
02: use strict;
03:
04: # load modules
05: use Inline with => 'Event';
06: use Inline C    => 'DATA';
07:
08: # install timer
09: start_timer();
10:
11: #start the loop
12: Event::loop;
13:
14: __END__
```

The only new thing is a call to a C function `start_timer` in line 9. This call replaces the timer setup which we did in Perl before. We will investigate this function on C side. And there we are − on C side now. First a watcher variable is declared. In this first solution, this is done globally to let it stay alive after watcher setup.

```
16: __C__
17:
18: /* declare watcher variable */
19: pe_timer * timer;
```

The variables type reflects what watcher we are going to deal with.

At this point, we are ready to write the callback function.

```
21: /* callback */
```

```
22: static void c_callback(pe_event * event)
23:  {
```

The callback is declared static because we do not want to use it on Perl side. Nevertheless, from a callback authors view there is *no* difference to a callback function to be invoked from Perl side, so *one and the same C callback can be used on both sides*. If you want to provide it all overall just omit `static`.

Note: in the *background*, calls from Perl and C side *are* different, but **Inlines** transparent parameter typemapping for Perl side calls results in the fact that we receive event data as `pe_event *` in both cases.

```
24:   /* declare variables, get the watcher */
25:   pe_timer * watcher=(pe_timer *)event->up;
26:  double interval;
27:
28:   printf("Here is the C callback!\n");
29:
30:   /* inspect Event data: event */
31:   printf(
32:          "Event:\nhits: %d,\nprio: %d.\n\n",
33:          event->hits,
34:          event->prio
35:         );
36:
37:   /* inspect Event data: base watcher attributes */
38:   printf(
39:          "Watcher (common attributes):\ndesc: %s,\nmax_cb_tm: %d,\nprio: %d,\ncbtime: %f,\nis_running: %d.\
40:          SvPVX(watcher->base.desc),
41:          watcher->base.max_cb_tm,
42:          watcher->base.prio,
43:          watcher->base.cbtime,
44:          watcher->base.running
45:         );
46:
47:   /* get interval */
48:   GEventAPI->sv_2interval("label", watcher->interval, &interval);
49:
50:   /* inspect Event data: specific watcher attributes */
51:   printf(
52:          "Watcher (timer specific attributes):\ninterval: %f.\n\n",
53:          interval
54:         );
55:  }
```

Finally, the watcher has to be set up. This is the code:

```
57: void start_timer()
58:  {
59:   if (!timer)
60:     {
61:      /* make new watcher */
62:      timer=GEventAPI->new_timer(0, 0);
63:
64:      /* set it up */
65:      timer->base.ext_data=(void*)timer;
66:      timer->base.callback=(void*)c_callback;
67:
68:      sv_setpv(timer->base.desc, "timer setup in C");
69:
70:      WaREPEAT_on(timer);
71:
72:      timer->tm.at=2.0;
73:      timer->interval=newSVnv(2.0);
74:     }
75:
76:   /* start the watcher */
77:   GEventAPI->start((pe_watcher*)timer, 0);
78:  }
```

First, the watcher data structure has to be made. Because **Event** implements several different watchers, the API provides various functions to instantiate the specific data structures. There is one initial function for each watcher type. In the timers case, this function is called `new_timer()`, used in line 62.

Because we are on C side now, we are responsible to perform most of the necessary steps in watcher setup ourselves. First, there is an important base watcher data `ext_data`, which has to be set up by the watcher instance.

Then the callback hint is stored which is done by assigning a pointer to the callback function to the appropriate watcher data (line 66).

The watcher description is an SV and therefore has to be set up by a Perl API function (line 68). `sv_setpv()` copies the description string to the watchers description field. This optional information can be stored in watchers of all types.

The following assignments are timer specific. We want this timer to repeat (line 70). The API macro `WaREPEAT_on()` sets the repeat flag. Please note that different to the Perl side this flag is not managed "automagically" (depending on which interval attribute is used). *If the timer shall repeat, the flag has to be set explicitly.*

On Perl side, one can set either the `at` or the `interval` attribute. On C side, you *have* to set `at` (line 72). This data field is a `double` indicating when the next timer event will happen. It depends on the timers behaviour how to setup this: for a non repeating timer, pass *the events time* (which can be calculated as the sum of the current time and the remaing delay), for a repeating timer, *only pass the interval* because **Event** will add the current time itself. The current time, by the way, can be get by calling the API function `NVtime()`.

The `interval` value is only necessary if the timer repeats. It is an SV expected to hold a `double`, and because of this, the best way to set it up is to make a new SV for a `double` and to assign it. This is done by using Perls API function `newSVnv()` (line 73).

Finally, the new watcher can be started by calling `GEventAPI->start()` (line 77). This function can be used for all watcher types.

That's all!

6.4. C watcher and Perl callback

This is a team for all **Event** users who want still use Perl functions on callback side, but wish to accelerate watcher installation and maintenance.

After all we've seen before, this is a simple final step. It can directly be derived from the "all C" solution. The Perl side is very similar, except that it now contains a usual watcher callback.

```
01: # pragma
02: use strict;
03:
04: # load module
05: use Inline with => Event;
06: use Inline C    => 'DATA';
07:
08: # install a timer
09: start_timer();
10: # start the loop
11: Event::loop;
12:
13: # Perl callback
14: sub callback($)
15:  {
16:   # get event
17:   my ($event)=@_;
18:
19:   print sprintf(
20:                 "Here is the Perl callback (%s),  %d event(s) were detected. The events priority was %d, th
21:                 $event->w->desc,
22:                 $event->hits,
23:                 $event->prio,
```

```
24:                 $event->w->prio,
25:                );
26:  }
27:
28: __END__
```

Note that there's nothing special in this callback. It could have been written for an "all Perl" solution.

Now on C side, we walk the well known path by declaring a watcher instance ...

```
30: __C__
31:
32: /* declare variables */
33: pe_timer * timer;
```

Well known until we come to the callback. What, a *C* callback here? Is this not the chapter about C watchers and *Perl* callbacks, of which we already wrote one? Yes, this is true, but we need an additional step to pass event data to Perl. Fortunately, it's quite short:

```
38: static void c_callback(pe_event * event)
39:  {
40:   dSP;
41:   PUSHMARK(SP);
42:   XPUSHs(GEventAPI->event_2sv(event));
43:   PUTBACK;
44:   call_pv("main::callback", 0);
45:  }
```

You may just copy this function, it should work for *all* watcher types and *all* Perl callbacks (except that you may want to adapt the callback name in line 44).

As usual, `static` hides the function from Perl side access.

The callback initially takes the received `pe_event` structure and converts it to a *Perl Event::Event object* by the API call `event_2sv()` (line 42). This object is put on Perls stack then (lines 40 to 43). Finally, we invoke the *Perl* callback (line 44), passing it's name just as a string. (The package name could even be omitted because we already are in the `main` package here.)

The performed way of a Perl function call is documented in general in Perls *perlcall* manpage.

Well, the new part is done! All that remains is watcher setup, and this is performed the same way as in an "all C" solution.

```
47: void start_timer()
48:  {
49:   if (!timer)
50:     {
51:      /* make new watcher */
52:      timer=GEventAPI->new_timer(0, 0);
53:
54:      /* set it up */
55:      sv_setpv(timer->base.desc, "timer setup in C");
56:      timer->base.callback=(void*)c_callback;
57:      timer->base.ext_data=(void*)timer;
58:      timer->interval=newSVnv(2.0);
59:      timer->tm.at=2.0;
60:      WaREPEAT_on(timer);
61:     }
62:
63:   /* start the watcher */
64:   GEventAPI->start((pe_watcher*)timer, 0);
65:  }
```

Now you may assemble these script snippets to see the example run. ;−)

7. Application examples

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

Application
A bankers **trading system**.
The state machine library **POE** (to be found on CPAN) uses **Event** as one of several event libraries.
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."
A news scanner contacting hundreds of servers *simultaniously* to get the maximal band width.
database frontends
web backends
client/server architectures

8. Outlook

Event is constantly improved. Joshua 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*.

A. Data

Information	*Details*
name	**Event.pm** by Joshua N. Pritikin (JPRIT).
version	This tutorial is based on version 0.80.
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.

A.1. About this document

Credits Thanks to *Joshua N. Pritikin* for **Event**, his constant and immediate module maintenance, and for his patient and immediate support in all questions I asked for this document; to *Brian Ingerson* for the **Inline** module (which immediately inspired me to add the C API chapter) and his help in figuring out how to make it work together with **Event**. Special thanks to him for the **Inline** extensions of tranparent module typemapping (introduced with **Inline** 0.30). Thanks to *Marc Lehmann* for his review of the first version of the German Workshop talk (the root of this tutorial).

This tutorial is intended to be helpful. Please send comments, questions and suggestions to perl@jochen−stenzel.de or discuss them in **Event**s mailing list.