

NAME

Opcode - Disable named opcodes when compiling perl code

SYNOPSIS

```
use Opcode;
```

DESCRIPTION

Perl code is always compiled into an internal format before execution.

Evaluating perl code (e.g. via "eval" or "do 'file'") causes the code to be compiled into an internal format and then, provided there was no error in the compilation, executed. The internal format is based on many distinct *opcodes*.

By default no opmask is in effect and any code can be compiled.

The Opcode module allow you to define an *operator mask* to be in effect when perl *next* compiles any code. Attempting to compile code which contains a masked opcode will cause the compilation to fail with an error. The code will not be executed.

NOTE

The Opcode module is not usually used directly. See the ops pragma and Safe modules for more typical uses.

WARNING

The authors make **no warranty**, implied or otherwise, about the suitability of this software for safety or security purposes.

The authors shall not in any case be liable for special, incidental, consequential, indirect or other similar damages arising from the use of this software.

Your mileage will vary. If in any doubt **do not use it**.

Operator Names and Operator Lists

The canonical list of operator names is the contents of the array PL_op_name defined and initialised in file *opcode.h* of the Perl source distribution (and installed into the perl library).

Each operator has both a terse name (its opname) and a more verbose or recognisable descriptive name. The opdesc function can be used to return a list of descriptions for a list of operators.

Many of the functions and methods listed below take a list of operators as parameters. Most operator lists can be made up of several types of element. Each element can be one of

an operator name (opname)

Operator names are typically small lowercase words like enterloop, leaveloop, last, next, redo etc. Sometimes they are rather cryptic like gv2cv, i_ncmp and ftsvtx.

an operator tag name (optag)

Operator tags can be used to refer to groups (or sets) of operators. Tag names always begin with a colon. The Opcode module defines several optags and the user can define others using the define_optag function.

a negated opname or optag

An opname or optag can be prefixed with an exclamation mark, e.g., !mkdir. Negating an opname or optag means remove the corresponding ops from the accumulated set of ops at that point.

an operator set (opset)

An *opset* as a binary string of approximately 44 bytes which holds a set or zero or more operators.

The `opset` and `opset_to_ops` functions can be used to convert from a list of operators to an opset and *vice versa*.

Wherever a list of operators can be given you can use one or more opsets. See also *Manipulating Opsets* below.

Opcode Functions

The Opcode package contains functions for manipulating operator names tags and sets. All are available for export by the package.

`opcodes`

In a scalar context `opcodes` returns the number of opcodes in this version of perl (around 350 for perl-5.7.0).

In a list context it returns a list of all the operator names. (Not yet implemented, use `@names = opset_to_ops(full_opset)`.)

`opset (OP, ...)`

Returns an opset containing the listed operators.

`opset_to_ops (OPSET)`

Returns a list of operator names corresponding to those operators in the set.

`opset_to_hex (OPSET)`

Returns a string representation of an opset. Can be handy for debugging.

`full_opset`

Returns an opset which includes all operators.

`empty_opset`

Returns an opset which contains no operators.

`invert_opset (OPSET)`

Returns an opset which is the inverse set of the one supplied.

`verify_opset (OPSET, ...)`

Returns true if the supplied opset looks like a valid opset (is the right length etc) otherwise it returns false. If an optional second parameter is true then `verify_opset` will croak on an invalid opset instead of returning false.

Most of the other Opcode functions call `verify_opset` automatically and will croak if given an invalid opset.

`define_optag (OPTAG, OPSET)`

Define OPTAG as a symbolic name for OPSET. Optag names always start with a colon :.

The optag name used must not be defined already (`define_optag` will croak if it is already defined). Optag names are global to the perl process and optag definitions cannot be altered or deleted once defined.

It is strongly recommended that applications using Opcode should use a leading capital letter on their tag names since lowercase names are reserved for use by the Opcode module. If using Opcode within a module you should prefix your tags names with the name of your module to ensure uniqueness and thus avoid clashes with other modules.

opmask_add (OPSET)

Adds the supplied opset to the current opmask. Note that there is currently *no* mechanism for unmasking ops once they have been masked. This is intentional.

opmask

Returns an opset corresponding to the current opmask.

opdesc (OP, ...)

This takes a list of operator names and returns the corresponding list of operator descriptions.

opdump (PAT)

Dumps to STDOUT a two column list of op names and op descriptions. If an optional pattern is given then only lines which match the (case insensitive) pattern will be output.

It's designed to be used as a handy command line utility:

```
perl -MOpc=opdump -e opdump
perl -MOpc=opdump -e 'opdump Eval'
```

Manipulating Opsets

Opsets may be manipulated using the perl bit vector operators & (and), | (or), ^ (xor) and ~ (negate/invert).

However you should never rely on the numerical position of any opcode within the opset. In other words both sides of a bit vector operator should be opsets returned from Opcode functions.

Also, since the number of opcodes in your current version of perl might not be an exact multiple of eight, there may be unused bits in the last byte of an opset. This should not cause any problems (Opcode functions ignore those extra bits) but it does mean that using the ~ operator will typically not produce the same 'physical' opset 'string' as the invert_opset function.

TO DO (maybe)

```
$bool = opset_eq($opset1, $opset2) true if opsets are logically equiv
```

```
$yes = opset_can($opset, @ops) true if $opset has all @ops set
```

```
@diff = opset_diff($opset1, $opset2) => ('foo', '!bar', ...)
```

Predefined Opcode Tags

:base_core

null stub scalar pushmark wantarray const defined undef

rv2sv sassign

rv2av aassign aelem aelemfast aslice av2arylen

rv2hv helem hslice each values keys exists delete aeach akeys

avalues

boolkeys

preinc i_preinc predec i_predec postinc i_postinc postdec

i_postdec

int hex oct abs pow multiply i_multiply divide i_divide

```
modulo i_modulo add i_add subtract i_subtract

left_shift right_shift bit_and bit_xor bit_or negate i_negate
not complement

lt i_lt gt i_gt le i_le ge i_ge eq i_eq ne i_nencmp i_ncmp
slt sgt sle sge seq sne scmp

substr vec stringify study pos length index rindex ord chr

ucfirst lcfirst uc lc quotemeta trans chop schop chomp schomp

match split qr

list lslice splice push pop shift unshift reverse

cond_expr flip flop andassign orassign dorassign and or dor xor

warn die lineseq nextstate scope enter leave

rv2cv anoncode prototype

entersub leavesub leavesublv return method method_named -- XXX
loops via recursion?

leaveeval -- needed for Safe to operate, is safe without
entereval
```

:base_mem

These memory related ops are not included in :base_core because they can easily be used to implement a resource attack (e.g., consume all available memory).

```
concat repeat join range
```

```
anonlist anonhash
```

Note that despite the existence of this optag a memory resource attack may still be possible using only :base_core ops.

Disabling these ops is a *very* heavy handed way to attempt to prevent a memory resource attack. It's probable that a specific memory limit mechanism will be added to perl in the near future.

:base_loop

These loop ops are not included in :base_core because they can easily be used to implement a resource attack (e.g., consume all available CPU time).

```
grepstart grepwhile
mapstart mapwhile
enteriter iter
enterloop leaveloop unstack
last next redo
goto
```

:base_io

These ops enable *filehandle* (rather than filename) based input and output. These are safe on the assumption that only pre-existing filehandles are available for use. Usually, to create

new filehandles other ops such as open would need to be enabled, if you don't take into account the magical open of ARGV.

```
readline rcatline getc read

formline enterwrite leavewrite

print say sysread syswrite send recv

eof tell seek sysseek

readdir telldir seekdir rewinddir
```

:base_orig

These are a hotchpotch of opcodes still waiting to be considered

```
gvsv gv gelem

padsv padav padhv padany

once

rv2gv refgen srefgen ref

bless -- could be used to change ownership of objects
(reblessing)

pushre regcmaybe regcreset regcomp subst substcont

sprintf prtft -- can core dump

crypt

tie untie

dbmopen dbmclose
sselect select
pipe_op sockpair

getppid getpgrp setpgrp getpriority setpriority localtime gmtime

entertry leavetry -- can be used to 'hide' fatal errors

entergiven leavegiven
enterwhen leavewhen
break continue
smartmatch

custom -- where should this go
```

:base_math

These ops are not included in :base_core because of the risk of them being used to generate floating point exceptions (which would have to be caught using a \$SIG{FPE} handler).

```
atan2 sin cos exp log sqrt
```

These ops are not included in `:base_core` because they have an effect beyond the scope of the compartment.

`rand srand`

`:base_thread`

These ops are related to multi-threading.

`lock`

`:default`

A handy tag name for a *reasonable* default set of ops. (The current ops allowed are unstable while development continues. It will change.)

`:base_core :base_mem :base_loop :base_orig :base_thread`

This list used to contain `:base_io` prior to Opcode 1.07.

If safety matters to you (and why else would you be using the Opcode module?) then you should not rely on the definition of this, or indeed any other, optag!

`:filesystem_read`

`stat lstat readlink`

`ftatime ftblk ftchr ftctime ftdir fteexec fteowned fteread
ftewrite ftfile ftis ftlink ftmtime ftpipe ftrexec ftrowned
ftrread ftsgid ftsize ftsock ftsuid fttty ftzero ftrwrite ftsvtx`

`fttext ftbinary`

`fileno`

`:sys_db`

<code>ghbyname ghbyaddr gghostent shostent ehostent</code>	<code>-- hosts</code>
<code>gnbyname gnbyaddr gnetent snetent enetent</code>	<code>-- networks</code>
<code>gpbyname gpbynumber gprotoent sprotoent eprotoent</code>	<code>-- protocols</code>
<code>gsbyname gsbyport gservernt sservernt eservernt</code>	<code>-- services</code>
<code>gpwnam gpwuid gpwent spwent epwent getlogin</code>	<code>-- users</code>
<code>ggrnam ggrgid ggrent sgrent egrent</code>	<code>-- groups</code>

`:browse`

A handy tag name for a *reasonable* default set of ops beyond the `:default` optag. Like `:default` (and indeed all the other optags) its current definition is unstable while development continues. It will change.

The `:browse` tag represents the next step beyond `:default`. It is a superset of the `:default` ops and adds `:filesystem_read` the `:sys_db`. The intent being that scripts can access more (possibly sensitive) information about your system but not be able to change it.

`:default :filesystem_read :sys_db`

`:filesystem_open`

`sysopen open close
umask binmode`

`open_dir closedir -- other dir ops are in :base_io`

:filesystem_write

link unlink rename symlink truncate

mkdir rmdir

utime chmod chown

fcntl -- not strictly filesystem related, but possibly as dangerous?

:subprocess

backtick system

fork

wait waitpid

glob -- access to Cshell via <`rm *`>

:ownprocess

exec exit kill

time tms -- could be used for timing attacks (paranoid?)

:others

This tag holds groups of assorted specialist opcodes that don't warrant having optags defined for them.

SystemV Interprocess Communications:

msgctl msgget msgrcv msgsnd

semctl semget semop

shmctl shmget shmread shmwrite

:load

This tag holds opcodes related to loading modules and getting information about calling environment and args.

require dofile

caller

:still_to_be_decided

chdir

flock ioctl

socket getpeername ssocket

bind connect listen accept shutdown gsocket getsockname

sleep alarm -- changes global timer state and signal handling

sort -- assorted problems including core dumps

tied -- can be used to access object implementing a tie

pack unpack -- can be used to create/use memory pointers

`hintseval` -- constant op holding eval hints

`entereval` -- can be used to hide code from initial compile

`reset`

`dbstate` -- perl -d version of `nextstate(ment)` opcode

`:dangerous`

This tag is simply a bucket for opcodes that are unlikely to be used via a tag name but need to be tagged for completeness and documentation.

`syscall` `dump` `chroot`

SEE ALSO

ops -- perl pragma interface to Opcode module.

Safe -- Opcode and namespace limited execution compartments

AUTHORS

Originally designed and implemented by Malcolm Beattie, mbeattie@sable.ox.ac.uk as part of Safe version 1.

Split out from Safe module version 1, named opcode tags and other changes added by Tim Bunce.