NAME

perltodo - Perl TO-DO List

DESCRIPTION

This is a list of wishes for Perl. The most up to date version of this file is at http://perl5.git.perl.org/perl.git/blob_plain/HEAD:/pod/perltodo.pod

The tasks we think are smaller or easier are listed first. Anyone is welcome to work on any of these, but it's a good idea to first contact *perl5-porters@perl.org* to avoid duplication of effort, and to learn from any previous attempts. By all means contact a pumpking privately first if you prefer.

Whilst patches to make the list shorter are most welcome, ideas to add to the list are also encouraged. Check the perl5-porters archives for past ideas, and any discussion about them. One set of archives may be found at:

```
    http://www.xray.mpe.mpg.de/mailing-lists/perl5-porters/
```

What can we offer you in return? Fame, fortune, and everlasting glory? Maybe not, but if your patch is incorporated, then we'll add your name to the *AUTHORS* file, which ships in the official distribution. How many other programming languages offer you 1 line of immortality?

Tasks that only need Perl knowledge

Improve Porting/cmpVERSION.pl to work from git tags

See *Porting/release_managers_guide.pod* for a bit more detail.

Migrate t/ from custom TAP generation

Many tests below *t/* still generate TAP by "hand", rather than using library functions. As explained in *"Writing a test" in perlhack*, tests in *t/* are written in a particular way to test that more complex constructions actually work before using them routinely. Hence they don't use `Test::More`, but instead there is an intentionally simpler library, *t/test.pl*. However, quite a few tests in *t/* have not been refactored to use it. Refactoring any of these tests, one at a time, is a useful thing TODO.

The subdirectories *base*, *cmd* and *comp*, that contain the most basic tests, should be excluded from this task.

Test that regen.pl was run

There are various generated files shipped with the perl distribution, for things like header files generate from data. The generation scripts are written in perl, and all can be run by *regen.pl*. However, because they're written in perl, we can't run them before we've built perl. We can't run them as part of the *Makefile*, because changing files underneath *make* confuses it completely, and we don't want to run them automatically anyway, as they change files shipped by the distribution, something we seek not do to.

If someone changes the data, but forgets to re-run *regen.pl* then the generated files are out of sync. It would be good to have a test in *t/porting* that checks that the generated files are in sync, and fails otherwise, to alert someone before they make a poor commit. I suspect that this would require adapting the scripts run from *regen.pl* to have dry-run options, and invoking them with these, or by refactoring them into a library that does the generation, which can be called by the scripts, and by the test.

Automate perldelta generation

The perldelta file accompanying each release summaries the major changes. It's mostly manually generated currently, but some of that could be automated with a bit of perl, specifically the generation of

Modules and Pragmata

New Documentation

New Tests

See *Porting/how_to_write_a_perldelta.pod* for details.

Remove duplication of test setup.

Schwern notes, that there's duplication of code - lots and lots of tests have some variation on the big block of `$Is_Foo` checks. We can safely put this into a file, change it to build an `%Is` hash and require it. Maybe just put it into *test.pl*. Throw in the handy tainting subroutines.

POD -> HTML conversion in the core still sucks

Which is crazy given just how simple POD purports to be, and how simple HTML can be. It's not actually *as* simple as it sounds, particularly with the flexibility POD allows for `=item`, but it would be good to improve the visual appeal of the HTML generated, and to avoid it having any validation errors. See also *make HTML install work*, as the layout of installation tree is needed to improve the cross-linking.

The addition of `Pod::Simple` and its related modules may make this task easier to complete.

Make ExtUtils::ParseXS use strict;

lib/ExtUtils/ParseXS.pm contains this line

```
    # use strict;  # One of these days...
```

Simply uncomment it, and fix all the resulting issues :-)

The more practical approach, to break the task down into manageable chunks, is to work your way though the code from bottom to top, or if necessary adding extra `{ ... }` blocks, and turning on strict within them.

Make Schwern poorer

We should have tests for everything. When all the core's modules are tested, Schwern has promised to donate to $500 to TPF. We may need volunteers to hold him upside down and shake vigorously in order to actually extract the cash.

Improve the coverage of the core tests

Use Devel::Cover to ascertain the core modules' test coverage, then add tests that are currently missing.

test B

A full test suite for the B module would be nice.

A decent benchmark

`perlbench` seems impervious to any recent changes made to the perl core. It would be useful to have a reasonable general benchmarking suite that roughly represented what current perl programs do, and measurably reported whether tweaks to the core improve, degrade or don't really affect performance, to guide people attempting to optimise the guts of perl. Gisle would welcome new tests for perlbench.

fix tainting bugs

Fix the bugs revealed by running the test suite with the `-t` switch (via `make test.taintwarn`).

Dual life everything

As part of the "dists" plan, anything that doesn't belong in the smallest perl distribution needs to be dual lifed. Anything else can be too. Figure out what changes would be needed to package that module and its tests up for CPAN, and do so. Test it with older perl releases, and fix the problems you find.

To make a minimal perl distribution, it's useful to look at *t/lib/commonsense.t*.

Move dual-life pod/*.PL into ext

Nearly all the dual-life modules have been moved to *ext*. However, we still need to move *pod/*.PL* into their respective directories in *ext/*. They're referenced by (at least) `plextract` in *Makefile.SH* and `utils` in *win32/Makefile* and *win32/makefile.ml*, and listed explicitly in *win32/pod.mak*, *vms/descrip_mms.template* and *utils.lst*

POSIX memory footprint

Ilya observed that use POSIX; eats memory like there's no tomorrow, and at various times worked to cut it down. There is probably still fat to cut out - for example POSIX passes Exporter some very memory hungry data structures.

embed.pl/makedef.pl

There is a script *embed.pl* that generates several header files to prefix all of Perl's symbols in a consistent way, to provide some semblance of namespace support in `C`. Functions are declared in *embed.fnc*, variables in *interpvar.h*. Quite a few of the functions and variables are conditionally declared there, using `#ifdef`. However, *embed.pl* doesn't understand the C macros, so the rules about which symbols are present when is duplicated in *makedef.pl*. Writing things twice is bad, m'kay. It would be good to teach `embed.pl` to understand the conditional compilation, and hence remove the duplication, and the mistakes it has caused.

use strict; and AutoLoad

Currently if you write

```
package Whack;
use AutoLoader 'AUTOLOAD';
use strict;
1;
__END__
sub bloop {
    print join (' ', No, strict, here), "!\n";
}
```

then `use strict;` isn't in force within the autoloaded subroutines. It would be more consistent (and less surprising) to arrange for all lexical pragmas in force at the __END__ block to be in force within each autoloaded subroutine.

There's a similar problem with SelfLoader.

profile installman

The *installman* script is slow. All it is doing text processing, which we're told is something Perl is good at. So it would be nice to know what it is doing that is taking so much CPU, and where possible address it.

enable lexical enabling/disabling of inidvidual warnings

Currently, warnings can only be enabled or disabled by category. There are times when it would be useful to quash a single warning, not a whole category.

Tasks that need a little sysadmin-type knowledge

Or if you prefer, tasks that you would learn from, and broaden your skills base...

make HTML install work

There is an `installhtml` target in the Makefile. It's marked as "experimental". It would be good to get this tested, make it work reliably, and remove the "experimental" tag. This would include

1 Checking that cross linking between various parts of the documentation works. In particular

that links work between the modules (files with POD in *lib/*) and the core documentation (files in *pod/*)

2 Work out how to split `perlfunc` into chunks, preferably one per function group, preferably with general case code that could be used elsewhere. Challenges here are correctly identifying the groups of functions that go together, and making the right named external cross-links point to the right page. Things to be aware of are `-X`, groups such as `getpwnam` to `endservent`, two or more `=items` giving the different parameter lists, such as

```
    =item substr EXPR,OFFSET,LENGTH,REPLACEMENT
    =item substr EXPR,OFFSET,LENGTH
    =item substr EXPR,OFFSET
```

and different parameter lists having different meanings. (eg `select`)

compressed man pages

Be able to install them. This would probably need a configure test to see how the system does compressed man pages (same directory/different directory? same filename/different filename), as well as tweaking the *installman* script to compress as necessary.

Add a code coverage target to the Makefile

Make it easy for anyone to run Devel::Cover on the core's tests. The steps to do this manually are roughly

- do a normal `Configure`, but include Devel::Cover as a module to install (see *INSTALL* for how to do this)

-
  ```
      make perl
  ```

-
  ```
      cd t; HARNESS_PERL_SWITCHES=-MDevel::Cover ./perl -I../lib
  harness
  ```
- Process the resulting Devel::Cover database

This just give you the coverage of the *.pm*s. To also get the C level coverage you need to

- Additionally tell `Configure` to use the appropriate C compiler flags for `gcov`

-
  ```
      make perl.gcov
  ```

  (instead of `make perl`)
- After running the tests run `gcov` to generate all the *.gcov* files. (Including down in the subdirectories of *ext/*
- (From the top level perl directory) run `gcov2perl` on all the `.gcov` files to get their stats into the cover_db directory.
- Then process the Devel::Cover database

It would be good to add a single switch to `Configure` to specify that you wanted to perform perl level coverage, and another to specify C level coverage, and have `Configure` and the *Makefile* do all the right things automatically.

Make Config.pm cope with differences between built and installed perl

Quite often vendors ship a perl binary compiled with their (pay-for) compilers. People install a free compiler, such as gcc. To work out how to build extensions, Perl interrogates `%Config`, so in this situation `%Config` describes compilers that aren't there, and extension building fails. This forces people into choosing between re-compiling perl themselves using the compiler they have, or only using modules that the vendor ships.

It would be good to find a way teach `Config.pm` about the installation setup, possibly involving probing at install time or later, so that the `%Config` in a binary distribution better describes the installed machine, when the installed machine differs from the build machine in some significant way.

linker specification files

Some platforms mandate that you provide a list of a shared library's external symbols to the linker, so the core already has the infrastructure in place to do this for generating shared perl libraries. My understanding is that the GNU toolchain can accept an optional linker specification file, and restrict visibility just to symbols declared in that file. It would be good to extend *makedef.pl* to support this format, and to provide a means within `Configure` to enable it. This would allow Unix users to test that the export list is correct, and to build a perl that does not pollute the global namespace with private symbols.

Cross-compile support

Currently `Configure` understands `-Dusecrosscompile` option. This option arranges for building `miniperl` for TARGET machine, so this `miniperl` is assumed then to be copied to TARGET machine and used as a replacement of full `perl` executable.

This could be done little differently. Namely `miniperl` should be built for HOST and then full `perl` with extensions should be compiled for TARGET. This, however, might require extra trickery for %Config: we have one config first for HOST and then another for TARGET. Tools like MakeMaker will be mightily confused. Having around two different types of executables and libraries (HOST and TARGET) makes life interesting for Makefiles and shell (and Perl) scripts. There is $Config{run}, normally empty, which can be used as an execution wrapper. Also note that in some cross-compilation/execution environments the HOST and the TARGET do not see the same filesystem(s), the $Config{run} may need to do some file/directory copying back and forth.

roffitall

Make *pod/roffitall* be updated by *pod/buildtoc*.

Split "linker" from "compiler"

Right now, Configure probes for two commands, and sets two variables:

* `cc` (in *cc.U*)

> This variable holds the name of a command to execute a C compiler which can resolve multiple global references that happen to have the same name. Usual values are *cc* and *gcc*. Fervent ANSI compilers may be called *c89*. AIX has *xlc*.

* `ld` (in *dlsrc.U*)

> This variable indicates the program to be used to link libraries for dynamic loading. On some systems, it is *ld*. On ELF systems, it should be `$cc`. Mostly, we'll try to respect the hint file setting.

There is an implicit historical assumption from around Perl5.000alpha something, that `$cc` is also the correct command for linking object files together to make an executable. This may be true on Unix, but it's not true on other platforms, and there are a maze of work arounds in other places (such as *Makefile.SH*) to cope with this.

Ideally, we should create a new variable to hold the name of the executable linker program, probe for it in *Configure*, and centralise all the special case logic there or in hints files.

A small bikeshed issue remains - what to call it, given that `$ld` is already taken (arguably for the wrong thing now, but on SunOS 4.1 it is the command for creating dynamically-loadable modules) and `$link` could be confused with the Unix command line executable of the same name, which does something completely different. Andy Dougherty makes the counter argument "In parrot, I tried to call the command used to link object files and libraries into an executable *link*, since that's what my vaguely-remembered DOS and VMS experience suggested. I don't think any real confusion has ensued, so it's probably a reasonable name for perl5 to use."

"Alas, I've always worried that introducing it would make things worse, since now the module building utilities would have to look for `$Config{link}` and institute a fall-back plan if it weren't found." Although I can see that as confusing, given that `$Config{d_link}` is true when (hard) links are available.

Configure Windows using PowerShell

Currently, Windows uses hard-coded config files based to build the config.h for compiling Perl. Makefiles are also hard-coded and need to be hand edited prior to building Perl. While this makes it easy to create a perl.exe that works across multiple Windows versions, being able to accurately configure a perl.exe for a specific Windows versions and VS C++ would be a nice enhancement. With PowerShell available on Windows XP and up, this may now be possible. Step 1 might be to investigate whether this is possible and use this to clean up our current makefile situation. Step 2 would be to see if there would be a way to use our existing metaconfig units to configure a Windows Perl or whether we go in a separate direction and make it so. Of course, we all know what step 3 is.

decouple -g and -DDEBUGGING

Currently *Configure* automatically adds `-DDEBUGGING` to the C compiler flags if it spots `-g` in the optimiser flags. The pre-processor directive `DEBUGGING` enables *perl*'s command line `-D` options, but in the process makes *perl* slower. It would be good to disentangle this logic, so that C-level debugging with `-g` and Perl level debugging with `-D` can easily be enabled independently.

Tasks that need a little C knowledge

These tasks would need a little C knowledge, but don't need any specific background or experience with XS, or how the Perl interpreter works

Weed out needless PERL_UNUSED_ARG

The C code uses the macro `PERL_UNUSED_ARG` to stop compilers warning about unused arguments. Often the arguments can't be removed, as there is an external constraint that determines the prototype of the function, so this approach is valid. However, there are some cases where `PERL_UNUSED_ARG` could be removed. Specifically

- The prototypes of (nearly all) static functions can be changed
- Unused arguments generated by short cut macros are wasteful - the short cut macro used can be changed.

Modernize the order of directories in @INC

The way @INC is laid out by default, one cannot upgrade core (dual-life) modules without overwriting files. This causes problems for binary package builders. One possible proposal is laid out in this message: *http://www.xray.mpe.mpg.de/mailing-lists/perl5-porters/2002-04/msg02380.html*.

-Duse32bit*

Natively 64-bit systems need neither -Duse64bitint nor -Duse64bitall. On these systems, it might be the default compilation mode, and there is currently no guarantee that passing no use64bitall option to the Configure process will build a 32bit perl. Implementing -Duse32bit* options would be nice for perl 5.12.

Profile Perl - am I hot or not?

The Perl source code is stable enough that it makes sense to profile it, identify and optimise the hotspots. It would be good to measure the performance of the Perl interpreter using free tools such as cachegrind, gprof, and dtrace, and work to reduce the bottlenecks they reveal.

As part of this, the idea of *pp_hot.c* is that it contains the *hot* ops, the ops that are most commonly used. The idea is that by grouping them, their object code will be adjacent in the executable, so they have a greater chance of already being in the CPU cache (or swapped in) due to being near another op already in use.

Except that it's not clear if these really are the most commonly used ops. So as part of exercising your skills with coverage and profiling tools you might want to determine what ops *really* are the most commonly used. And in turn suggest evictions and promotions to achieve a better *pp_hot.c*.

One piece of Perl code that might make a good testbed is *installman*.

Allocate OPs from arenas

Currently all new OP structures are individually malloc()ed and free()d. All `malloc` implementations have space overheads, and are now as fast as custom allocates so it would both use less memory and less CPU to allocate the various OP structures from arenas. The SV arena code can probably be re-used for this.

Note that Configuring perl with `-Accflags=-DPL_OP_SLAB_ALLOC` will use Perl_Slab_alloc() to pack optrees into a contiguous block, which is probably superior to the use of OP arenas, esp. from a cache locality standpoint. See *Profile Perl - am I hot or not?*.

Improve win32/wince.c

Currently, numerous functions look virtually, if not completely, identical in both `win32/wince.c` and `win32/win32.c` files, which can't be good.

Use secure CRT functions when building with VC8 on Win32

Visual C++ 2005 (VC++ 8.x) deprecated a number of CRT functions on the basis that they were "unsafe" and introduced differently named secure versions of them as replacements, e.g. instead of writing

```
    FILE* f = fopen(__FILE__, "r");
```

one should now write

```
    FILE* f;
    errno_t err = fopen_s(&f, __FILE__, "r");
```

Currently, the warnings about these deprecations have been disabled by adding -D_CRT_SECURE_NO_DEPRECATE to the CFLAGS. It would be nice to remove that warning suppressant and actually make use of the new secure CRT functions.

There is also a similar issue with POSIX CRT function names like fileno having been deprecated in favour of ISO C++ conformant names like _fileno. These warnings are also currently suppressed by adding -D_CRT_NONSTDC_NO_DEPRECATE. It might be nice to do as Microsoft suggest here too, although, unlike the secure functions issue, there is presumably little or no benefit in this case.

Fix POSIX::access() and chdir() on Win32

These functions currently take no account of DACLs and therefore do not behave correctly in situations where access is restricted by DACLs (as opposed to the read-only attribute).

Furthermore, POSIX::access() behaves differently for directories having the read-only attribute set depending on what CRT library is being used. For example, the _access() function in the VC6 and VC7 CRTs (wrongly) claim that such directories are not writable, whereas in fact all directories are writable unless access is denied by DACLs. (In the case of directories, the read-only attribute actually only means that the directory cannot be deleted.) This CRT bug is fixed in the VC8 and VC9 CRTs (but, of course, the directory may still not actually be writable if access is indeed denied by DACLs).

For the chdir() issue, see ActiveState bug #74552:
http://bugs.activestate.com/show_bug.cgi?id=74552

Therefore, DACLs should be checked both for consistency across CRTs and for the correct answer.

(Note that perl's -w operator should not be modified to check DACLs. It has been written so that it reflects the state of the read-only attribute, even for directories (whatever CRT is being used), for symmetry with chmod().)

strcat(), strcpy(), strncat(), strncpy(), sprintf(), vsprintf()

Maybe create a utility that checks after each libperl.a creation that none of the above (nor sprintf(), vsprintf(), or *SHUDDER* gets()) ever creep back to libperl.a.

```
  nm libperl.a | ./miniperl -alne '$o = $F[0] if /:$/; print "$o $F[1]" if
$F[0] eq "U" && $F[1] =~ /^(?:strn?c(?:at|py)|v?sprintf|gets)$/'
```

Note, of course, that this will only tell whether **your** platform is using those naughty interfaces.

-D_FORTIFY_SOURCE=2, -fstack-protector

Recent glibcs support `-D_FORTIFY_SOURCE=2` and recent gcc (4.1 onwards?) supports `-fstack-protector`, both of which give protection against various kinds of buffer overflow problems. These should probably be used for compiling Perl whenever available, Configure and/or hints files should be adjusted to probe for the availability of these features and enable them as appropriate.

Arenas for GPs? For MAGIC?

`struct gp` and `struct magic` are both currently allocated by `malloc`. It might be a speed or memory saving to change to using arenas. Or it might not. It would need some suitable benchmarking first. In particular, `GP`s can probably be changed with minimal compatibility impact (probably nothing outside of the core, or even outside of *gv.c* allocates them), but they probably aren't allocated/deallocated often enough for a speed saving. Whereas `MAGIC` is allocated/deallocated more often, but in turn, is also something more externally visible, so changing the rules here may bite external code.

Shared arenas

Several SV body structs are now the same size, notably PVMG and PVGV, PVAV and PVHV, and PVCV and PVFM. It should be possible to allocate and return same sized bodies from the same actual arena, rather than maintaining one arena for each. This could save 4-6K per thread, of memory no longer tied up in the not-yet-allocated part of an arena.

Tasks that need a knowledge of XS

These tasks would need C knowledge, and roughly the level of knowledge of the perl API that comes from writing modules that use XS to interface to C.

Write an XS cookbook

Create pod/perlxscookbook.pod with short, task-focused 'recipes' in XS that demonstrate common tasks and good practices. (Some of these might be extracted from perlguts.) The target audience should be XS novices, who need more examples than perlguts but something less overwhelming than perlapi. Recipes should provide "one pretty good way to do it" instead of TIMTOWTDI.

Rather than focusing on interfacing Perl to C libraries, such a cookbook should probably focus on how to optimize Perl routines by re-writing them in XS. This will likely be more motivating to those who mostly work in Perl but are looking to take the next step into XS.

Deconstructing and explaining some simpler XS modules could be one way to bootstrap a cookbook. (List::Util? Class::XSAccessor? Tree::Ternary_XS?) Another option could be deconstructing the implementation of some simpler functions in op.c.

Allow XSUBs to inline themselves as OPs

For a simple XSUB, often the subroutine dispatch takes more time than the XSUB itself. The tokeniser already has the ability to inline constant subroutines - it would be good to provide a way to inline other subroutines.

Specifically, simplest approach looks to be to allow an XSUB to provide an alternative implementation of itself as a custom OP. A new flag bit in `CvFLAGS()` would signal to the peephole optimiser to take an optree such as this:

```
b  <@> leave[1 ref] vKP/REFC ->(end)
1     <0> enter ->2
2     <;> nextstate(main 1 -e:1) v:{ ->3
a     <2> sassign vKS/2 ->b
8        <1> entersub[t2] sKS/TARG,1 ->9
-           <1> ex-list sK ->8
3              <0> pushmark s ->4
4              <$> const(IV 1) sM ->5
6              <1> rv2av[t1] lKM/1 ->7
5                 <$> gv(*a) s ->6
-              <1> ex-rv2cv sK ->-
7                 <$> gv(*x) s/EARLYCV ->8
-        <1> ex-rv2sv sKRM*/1 ->a
9           <$> gvsv(*b) s ->a
```

perform the symbol table lookup of `rv2cv` and `gv(*x)`, locate the pointer to the custom OP that provides the direct implementation, and re- write the optree something like:

```
b  <@> leave[1 ref] vKP/REFC ->(end)
1     <0> enter ->2
2     <;> nextstate(main 1 -e:1) v:{ ->3
a     <2> sassign vKS/2 ->b
7        <1> custom_x -> 8
-           <1> ex-list sK ->7
3              <0> pushmark s ->4
4              <$> const(IV 1) sM ->5
6              <1> rv2av[t1] lKM/1 ->7
5                 <$> gv(*a) s ->6
-              <1> ex-rv2cv sK ->-
-                 <$> ex-gv(*x) s/EARLYCV ->7
-        <1> ex-rv2sv sKRM*/1 ->a
8           <$> gvsv(*b) s ->a
```

i.e. the `gv(*)` OP has been nulled and spliced out of the execution path, and the `entersub` OP has been replaced by the custom op.

This approach should provide a measurable speed up to simple XSUBs inside tight loops. Initially one would have to write the OP alternative implementation by hand, but it's likely that this should be reasonably straightforward for the type of XSUB that would benefit the most. Longer term, once the run-time implementation is proven, it should be possible to progressively update ExtUtils::ParseXS to generate OP implementations for some XSUBs.

Remove the use of SVs as temporaries in dump.c

dump.c contains debugging routines to dump out the contains of perl data structures, such as SVs, AV s and HVs. Currently, the dumping code **uses** SVs for its temporary buffers, which was a logical initial implementation choice, as they provide ready made memory handling.

However, they also lead to a lot of confusion when it happens that what you're trying to debug is seen by the code in *dump.c*, correctly or incorrectly, as a temporary scalar it can use for a temporary buffer. It's also not possible to dump scalars before the interpreter is properly set up, such as during ithreads cloning. It would be good to progressively replace the use of scalars as string accumulation buffers with something much simpler, directly allocated by `malloc`. The *dump.c* code is (or should be) only producing 7 bit US-ASCII, so output character sets are not an issue.

Producing and proving an internal simple buffer allocation would make it easier to re-write the internals of the PerlIO subsystem to avoid using SVs for **its** buffers, use of which can cause problems similar to those of *dump.c*, at similar times.

safely supporting POSIX SA_SIGINFO

Some years ago Jarkko supplied patches to provide support for the POSIX SA_SIGINFO feature in Perl, passing the extra data to the Perl signal handler.

Unfortunately, it only works with "unsafe" signals, because under safe signals, by the time Perl gets to run the signal handler, the extra information has been lost. Moreover, it's not easy to store it somewhere, as you can't call mutexs, or do anything else fancy, from inside a signal handler.

So it strikes me that we could provide safe SA_SIGINFO support

1. Provide global variables for two file descriptors
2. When the first request is made via `sigaction` for `SA_SIGINFO`, create a pipe, store the reader in one, the writer in the other
3. In the "safe" signal handler (`Perl_csighandler()`/`S_raise_signal()`), if the `siginfo_t` pointer non-`NULL`, and the writer file handle is open,
    1. serialise signal number, `struct siginfo_t` (or at least the parts we care about) into a small auto char buff
    2. `write()` that (non-blocking) to the writer fd
        1. if it writes 100%, flag the signal in a counter of "signals on the pipe" akin to the current per-signal-number counts
        2. if it writes 0%, assume the pipe is full. Flag the data as lost?
        3. if it writes partially, croak a panic, as your OS is broken.
4. in the regular `PERL_ASYNC_CHECK()` processing, if there are "signals on the pipe", read the data out, deserialise, build the Perl structures on the stack (code in `Perl_sighandler()`, the "unsafe" handler), and call as usual.

I think that this gets us decent `SA_SIGINFO` support, without the current risk of running Perl code inside the signal handler context. (With all the dangers of things like `malloc` corruption that that currently offers us)

For more information see the thread starting with this message: http://www.xray.mpe.mpg.de/mailing-lists/perl5-porters/2008-03/msg00305.html

autovivification

Make all autovivification consistent w.r.t LVALUE/RVALUE and strict/no strict;

This task is incremental - even a little bit of work on it will help.

Unicode in Filenames

chdir, chmod, chown, chroot, exec, glob, link, lstat, mkdir, open, opendir, qx, readdir, readlink, rename, rmdir, stat, symlink, sysopen, system, truncate, unlink, utime, -X. All these could potentially accept Unicode filenames either as input or output (and in the case of system and qx Unicode in general, as input or output to/from the shell). Whether a filesystem - an operating system pair understands Unicode in filenames varies.

Known combinations that have some level of understanding include Microsoft NTFS, Apple HFS+ (In Mac OS 9 and X) and Apple UFS (in Mac OS X), NFS v4 is rumored to be Unicode, and of course Plan 9. How to create Unicode filenames, what forms of Unicode are accepted and used (UCS-2, UTF-16, UTF-8), what (if any) is the normalization form used, and so on, varies. Finding the right level of interfacing to Perl requires some thought. Remember that an OS does not implicate a filesystem.

(The Windows -C command flag "wide API support" has been at least temporarily retired in 5.8.1, and the -C has been repurposed, see *perlrun*.)

Most probably the right way to do this would be this: *Virtualize operating system access*.

Unicode in %ENV

Currently the %ENV entries are always byte strings. See *Virtualize operating system access*.

Unicode and glob()

Currently glob patterns and filenames returned from File::Glob::glob() are always byte strings. See *Virtualize operating system access*.

use less 'memory'

Investigate trade offs to switch out perl's choices on memory usage. Particularly perl should be able to give memory back.

This task is incremental - even a little bit of work on it will help.

Re-implement :unique in a way that is actually thread-safe

The old implementation made bad assumptions on several levels. A good 90% solution might be just to make `:unique` work to share the string buffer of SvPVs. That way large constant strings can be shared between ithreads, such as the configuration information in *Config*.

Make tainting consistent

Tainting would be easier to use if it didn't take documented shortcuts and allow taint to "leak" everywhere within an expression.

readpipe(LIST)

system() accepts a LIST syntax (and a PROGRAM LIST syntax) to avoid running a shell. readpipe() (the function behind qx//) could be similarly extended.

Audit the code for destruction ordering assumptions

Change 25773 notes

```
    /* Need to check SvMAGICAL, as during global destruction it may be that
       AvARYLEN(av) has been freed before av, and hence the SvANY() pointer
       is now part of the linked list of SV heads, rather than pointing to
       the original body.  */
    /* FIXME - audit the code for other bugs like this one.  */
```

adding the `SvMAGICAL` check to

```
    if (AvARYLEN(av) && SvMAGICAL(AvARYLEN(av))) {
        MAGIC *mg = mg_find (AvARYLEN(av), PERL_MAGIC_arylen);
```

Go through the core and look for similar assumptions that SVs have particular types, as all bets are off during global destruction.

Extend PerlIO and PerlIO::Scalar

PerlIO::Scalar doesn't know how to truncate(). Implementing this would require extending the PerlIO vtable.

Similarly the PerlIO vtable doesn't know about formats (write()), or about stat(), or chmod()/chown(), utime(), or flock().

(For PerlIO::Scalar it's hard to see what e.g. mode bits or ownership would mean.)

PerlIO doesn't do directories or symlinks, either: mkdir(), rmdir(), opendir(), closedir(), seekdir(), rewinddir(), glob(); symlink(), readlink().

See also *Virtualize operating system access*.

-C on the #! line

It should be possible to make -C work correctly if found on the #! line, given that all perl command line options are strict ASCII, and -C changes only the interpretation of non-ASCII characters, and not for the script file handle. To make it work needs some investigation of the ordering of function calls during startup, and (by implication) a bit of tweaking of that order.

Organize error messages

Perl's diagnostics (error messages, see *perldiag*) could use reorganizing and formalizing so that each error message has its stable-for-all-eternity unique id, categorized by severity, type, and subsystem. (The error messages would be listed in a datafile outside of the Perl source code, and the source code would only refer to the messages by the id.) This clean-up and regularizing should apply for all croak() messages.

This would enable all sorts of things: easier translation/localization of the messages (though please do keep in mind the caveats of *Locale::Maketext* about too straightforward approaches to translation), filtering by severity, and instead of grepping for a particular error message one could look for a stable error id. (Of course, changing the error messages by default would break all the existing software depending on some particular error message...)

This kind of functionality is known as *message catalogs*. Look for inspiration for example in the catgets() system, possibly even use it if available-- but **only** if available, all platforms will **not** have catgets().

For the really pure at heart, consider extending this item to cover also the warning messages (see *perllexwarn*, `warnings.pl`).

Tasks that need a knowledge of the interpreter

These tasks would need C knowledge, and knowledge of how the interpreter works, or a willingness to learn.

forbid labels with keyword names

Currently `goto keyword` "computes" the label value:

```
    $ perl -e 'goto print'
    Can't find label 1 at -e line 1.
```

It is controversial if the right way to avoid the confusion is to forbid labels with keyword names, or if it would be better to always treat bareword expressions after a "goto" as a label and never as a keyword.

truncate() prototype

The prototype of truncate() is currently $$. It should probably be `*$` instead. (This is changed in *opcode.pl*)

decapsulation of smart match argument

Currently `$foo ~~ $object` will die with the message "Smart matching a non-overloaded object breaks encapsulation". It would be nice to allow to bypass this by using explictly the syntax `$foo ~~ %$object` or `$foo ~~ @$object`.

error reporting of [$a ; $b]

Using `;` inside brackets is a syntax error, and we don't propose to change that by giving it any meaning. However, it's not reported very helpfully:

```
    $ perl -e '$a = [$b; $c];'
    syntax error at -e line 1, near "$b;"
    syntax error at -e line 1, near "$c]"
    Execution of -e aborted due to compilation errors.
```

It should be possible to hook into the tokeniser or the lexer, so that when a `;` is parsed where it is not legal as a statement terminator (ie inside `{}` used as a hashref, `[]` or `()`) it issues an error something like *';' isn't legal inside an expression - if you need multiple statements use a do {...} block*. See the thread starting at http://www.xray.mpe.mpg.de/mailing-lists/perl5-porters/2008-09/msg00573.html

lexicals used only once

This warns:

```
    $ perl -we '$pie = 42'
    Name "main::pie" used only once: possible typo at -e line 1.
```

This does not:

```
    $ perl -we 'my $pie = 42'
```

Logically all lexicals used only once should warn, if the user asks for warnings. An unworked RT ticket (#5087) has been open for almost seven years for this discrepancy.

UTF-8 revamp

The handling of Unicode is unclean in many places. For example, the regexp engine matches in Unicode semantics whenever the string or the pattern is flagged as UTF-8, but that should not be dependent on an internal storage detail of the string.

Properly Unicode safe tokeniser and pads.

The tokeniser isn't actually very UTF-8 clean. `use utf8;` is a hack - variable names are stored in stashes as raw bytes, without the utf-8 flag set. The pad API only takes a `char *` pointer, so that's all bytes too. The tokeniser ignores the UTF-8-ness of `PL_rsfp`, or any SVs returned from source filters. All this could be fixed.

state variable initialization in list context

Currently this is illegal:

```
    state ($a, $b) = foo();
```

In Perl 6, `state ($a) = foo();` and `(state $a) = foo();` have different semantics, which is tricky to implement in Perl 5 as currently they produce the same opcode trees. The Perl 6 design is firm, so it would be good to implement the necessary code in Perl 5. There are comments in `Perl_newASSIGNOP()` that show the code paths taken by various assignment constructions involving state variables.

Implement $value ~~ 0 .. $range

It would be nice to extend the syntax of the ~~ operator to also understand numeric (and maybe alphanumeric) ranges.

A does() built-in

Like ref(), only useful. It would call the `DOES` method on objects; it would also tell whether something can be dereferenced as an array/hash/etc., or used as a regexp, etc.
http://www.xray.mpe.mpg.de/mailing-lists/perl5-porters/2007-03/msg00481.html

Tied filehandles and write() don't mix

There is no method on tied filehandles to allow them to be called back by formats.

Propagate compilation hints to the debugger

Currently a debugger started with -dE on the command-line doesn't see the features enabled by -E. More generally hints (`$^H` and `%^H`) aren't propagated to the debugger. Probably it would be a good thing to propagate hints from the innermost non-`DB::` scope: this would make code eval'ed in the debugger see the features (and strictures, etc.) currently in scope.

Attach/detach debugger from running program

The old perltodo notes "With `gdb`, you can attach the debugger to a running program if you pass the process ID. It would be good to do this with the Perl debugger on a running Perl program, although I'm not sure how it would be done." ssh and screen do this with named pipes in /tmp. Maybe we can too.

LVALUE functions for lists

The old perltodo notes that lvalue functions don't work for list or hash slices. This would be good to fix.

regexp optimiser optional

The regexp optimiser is not optional. It should configurable to be, to allow its performance to be measured, and its bugs to be easily demonstrated.

/w regex modifier

That flag would enable to match whole words, and also to interpolate arrays as alternations. With it, `/P/w` would be roughly equivalent to:

```
    do { local $"='|'; /\b(?:P)\b/ }
```

See *http://www.xray.mpe.mpg.de/mailing-lists/perl5-porters/2007-01/msg00400.html* for the discussion.

optional optimizer

Make the peephole optimizer optional. Currently it performs two tasks as it walks the optree - genuine peephole optimisations, and necessary fixups of ops. It would be good to find an efficient way to switch out the optimisations whilst keeping the fixups.

You WANT *how* many

Currently contexts are void, scalar and list. split has a special mechanism in place to pass in the number of return values wanted. It would be useful to have a general mechanism for this, backwards compatible and little speed hit. This would allow proposals such as short circuiting sort to be implemented as a module on CPAN.

lexical aliases

Allow lexical aliases (maybe via the syntax `my \$alias = \$foo`.

entersub XS vs Perl

At the moment pp_entersub is huge, and has code to deal with entering both perl and XS subroutines. Subroutine implementations rarely change between perl and XS at run time, so investigate using 2 ops to enter subs (one for XS, one for perl) and swap between if a sub is redefined.

Self-ties

Self-ties are currently illegal because they caused too many segfaults. Maybe the causes of these could be tracked down and self-ties on all types reinstated.

Optimize away @_

The old perltodo notes "Look at the "reification" code in `av.c`".

Virtualize operating system access

Implement a set of "vtables" that virtualizes operating system access (open(), mkdir(), unlink(), readdir(), getenv(), etc.) At the very least these interfaces should take SVs as "name" arguments instead of bare char pointers; probably the most flexible and extensible way would be for the Perl-facing interfaces to accept HVs. The system needs to be per-operating-system and per-file-system hookable/filterable, preferably both from XS and Perl level (*"Files and Filesystems" in perlport* is good reading at this point, in fact, all of *perlport* is.)

This has actually already been implemented (but only for Win32), take a look at *iperlsys.h* and *win32/perlhost.h*. While all Win32 variants go through a set of "vtables" for operating system access, non-Win32 systems currently go straight for the POSIX/Unix-style system/library call. Similar system as for Win32 should be implemented for all platforms. The existing Win32 implementation probably does not need to survive alongside this proposed new implementation, the approaches could be merged.

What would this give us? One often-asked-for feature this would enable is using Unicode for filenames, and other "names" like %ENV, usernames, hostnames, and so forth. (See *"When Unicode Does Not Happen" in perlunicode*.)

But this kind of virtualization would also allow for things like virtual filesystems, virtual networks, and "sandboxes" (though as long as dynamic loading of random object code is allowed, not very safe sandboxes since external code of course know not of Perl's vtables). An example of a smaller "sandbox" is that this feature can be used to implement per-thread working directories: Win32 already does this.

See also *Extend PerlIO and PerlIO::Scalar*.

Investigate PADTMP hash pessimisation

The peephole optimiser converts constants used for hash key lookups to shared hash key scalars. Under ithreads, something is undoing this work. See http://www.xray.mpe.mpg.de/mailing-lists/perl5-porters/2007-09/msg00793.html

Store the current pad in the OP slab allocator

Currently we leak ops in various cases of parse failure. I suggested that we could solve this by always using the op slab allocator, and walking it to free ops. Dave comments that as some ops are already freed during optree creation one would have to mark which ops are freed, and not double free them when walking the slab. He notes that one problem with this is that for some ops you have to know which pad was current at the time of allocation, which does change. I suggested storing a pointer to the current pad in the memory allocated for the slab, and swapping to a new slab each time the pad changes. Dave thinks that this would work.

repack the optree

Repacking the optree after execution order is determined could allow removal of NULL ops, and optimal ordering of OPs with respect to cache-line filling. The slab allocator could be reused for this purpose. I think that the best way to do this is to make it an optional step just before the completed optree is attached to anything else, and to use the slab allocator unchanged, so that freeing ops is identical whether or not this step runs. Note that the slab allocator allocates ops downwards in memory, so one would have to actually "allocate" the ops in reverse-execution order to get them contiguous in memory in execution order.

See http://www.nntp.perl.org/group/perl.perl5.porters/2007/12/msg131975.html

Note that running this copy, and then freeing all the old location ops would cause their slabs to be freed, which would eliminate possible memory wastage if the previous suggestion is implemented, and we swap slabs more frequently.

eliminate incorrect line numbers in warnings

This code

```
use warnings;
my $undef;

if ($undef == 3) {
} elsif ($undef == 0) {
}
```

used to produce this output:

```
Use of uninitialized value in numeric eq (==) at wrong.pl line 4.
Use of uninitialized value in numeric eq (==) at wrong.pl line 4.
```

where the line of the second warning was misreported - it should be line 5. Rafael fixed this - the problem arose because there was no nextstate OP between the execution of the `if` and the `elsif`, hence `PL_curcop` still reports that the currently executing line is line 4. The solution was to inject a nextstate OPs for each `elsif`, although it turned out that the nextstate OP needed to be a nulled OP, rather than a live nextstate OP, else other line numbers became misreported. (Jenga!)

The problem is more general than `elsif` (although the `elsif` case is the most common and the most confusing). Ideally this code

```
use warnings;
my $undef;

my $a = $undef + 1;
my $b
  = $undef
  + 1;
```

would produce this output

```
Use of uninitialized value $undef in addition (+) at wrong.pl line 4.
Use of uninitialized value $undef in addition (+) at wrong.pl line 7.
```

(rather than lines 4 and 5), but this would seem to require every OP to carry (at least) line number information.

What might work is to have an optional line number in memory just before the BASEOP structure, with a flag bit in the op to say whether it's present. Initially during compile every OP would carry its line number. Then add a late pass to the optimiser (potentially combined with *repack the optree*) which looks at the two ops on every edge of the graph of the execution path. If the line number changes, flags the destination OP with this information. Once all paths are traced, replace every op with the flag with a nextstate-light op (that just updates `PL_curcop`), which in turn then passes control on to the true op. All ops would then be replaced by variants that do not store the line number. (Which, logically, why it would work best in conjunction with *repack the optree*, as that is already copying/reallocating all the OPs)

(Although I should note that we're not certain that doing this for the general case is worth it)

optimize tail-calls

Tail-calls present an opportunity for broadly applicable optimization; anywhere that `return foo(...)` is called, the outer return can be replaced by a goto, and foo will return directly to the outer caller, saving (conservatively) 25% of perl's call&return cost, which is relatively higher than in C. The scheme language is known to do this heavily. B::Concise provides good insight into where this optimization is possible, ie anywhere entersub,leavesub op-sequence occurs.

```
 perl -MO=Concise,-exec,a,b,-main -e 'sub a{ 1 }; sub b {a()}; b(2)'
```

Bottom line on this is probably a new pp_tailcall function which combines the code in pp_entersub, pp_leavesub. This should probably be done 1st in XS, and using B::Generate to patch the new OP into the optrees.

Big projects

Tasks that will get your name mentioned in the description of the "Highlights of 5.12"

make ithreads more robust

Generally make ithreads more robust. See also *iCOW*

This task is incremental - even a little bit of work on it will help, and will be greatly appreciated.

One bit would be to write the missing code in sv.c:Perl_dirp_dup.

Fix Perl_sv_dup, et al so that threads can return objects.

iCOW

Sarathy and Arthur have a proposal for an improved Copy On Write which specifically will be able to COW new ithreads. If this can be implemented it would be a good thing.

(?{...}) closures in regexps

Fix (or rewrite) the implementation of the `/(?{...})/` closures.

A re-entrant regexp engine

This will allow the use of a regex from inside (?{ }), (??{ }) and (?(?{ })|) constructs.

Add class set operations to regexp engine

Apparently these are quite useful. Anyway, Jeffery Friedl wants them.

demerphq has this on his todo list, but right at the bottom.

Tasks for microperl

[Each and every one of these may be obsolete, but they were listed in the old Todo.micro file]

make creating uconfig.sh automatic

make creating Makefile.micro automatic

do away with fork/exec/wait?

(system, popen should be enough?)

some of the uconfig.sh really needs to be probed (using cc) in buildtime:

(uConfigure? :-) native datatype widths and endianness come to mind