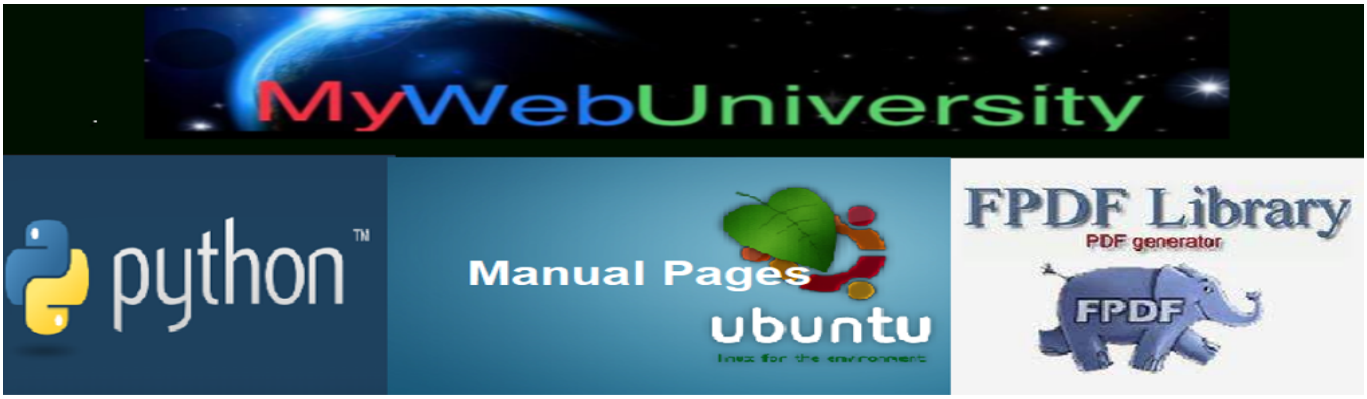

Rocky Enterprise Linux 9.2 Manual Pages on command 'DBI::DBD.3pm'

$ man DBI::DBD.3pm

DBI::DBD(3pm) User Contributed Perl Documentation DBI::DBD(3pm)

NAME

DBI::DBD - Perl DBI Database Driver Writer's Guide

SYNOPSIS

perldoc DBI::DBD

Version and volatility

This document is still a minimal draft which is in need of further work.

Please read the DBI documentation first and fully. Then look at the implementation of some high-profile and regularly maintained drivers like DBD::Oracle, DBD::ODBC, DBD::Pg etc. (Those are no no particular order.)

Then reread the DBI specification and the code of those drivers again as you're reading this. It'll help. Where this document and the driver code differ it's likely that the driver code is more correct, especially if multiple drivers do the same thing.

This document is a patchwork of contributions from various authors. More contributions (preferably as patches) are very welcome.

DESCRIPTION

This document is primarily intended to help people writing new database drivers for the Perl Database Interface (Perl DBI). It may also help others interested in discovering why the internals of a DBD driver are written the way they are.

This is a guide. Few (if any) of the statements in it are completely authoritative under all possible circumstances. This means you will need to use judgement in applying the guidelines in this document. If in any doubt at all, please do contact the dbi-dev mailing list (details given below) where Tim Bunce and other driver authors can help.

CREATING A NEW DRIVER

The first rule for creating a new database driver for the Perl DBI is very simple: DON'T!

There is usually a driver already available for the database you want to use, almost regardless of which database you choose. Very often, the database will provide an ODBC driver interface, so you can often use DBD::ODBC to access the database. This is typically less convenient on a Unix box than on a Microsoft Windows box, but there are numerous options for ODBC driver managers on Unix too, and very often the ODBC driver is provided by the database supplier.

Before deciding that you need to write a driver, do your homework to ensure that you are not wasting your energies.

[As of December 2002, the consensus is that if you need an ODBC driver manager on Unix, then the unixODBC driver (available from <http://www.unixodbc.org/>) is the way to go.]

The second rule for creating a new database driver for the Perl DBI is also very simple: Don't -- get someone else to do it for you!

Nevertheless, there are occasions when it is necessary to write a new driver, often to use a proprietary language or API to access the database more swiftly, or more comprehensively, than an ODBC driver can. Then you should read this document very carefully, but with a suitably sceptical eye.

If there is something in here that does not make any sense, question it. You might be right that the information is bogus, but don't come to that conclusion too quickly.

URLs and mailing lists

The primary web-site for locating DBI software and information is

http://dbi.perl.org/

There are two main and one auxiliary mailing lists for people working with DBI. The primary lists are dbi-users@perl.org for general users of DBI and DBD drivers, and dbi-dev@perl.org mainly for DBD driver writers (don't join the dbi-dev list unless you have a good reason). The auxiliary list is dbi-announce@perl.org for announcing new releases of DBI or DBD drivers.

You can join these lists by accessing the web-site <http://dbi.perl.org/>. The lists are closed so you cannot send email to any of the lists unless you join the list first.

You should also consider monitoring the comp.lang.perl.* newsgroups, especially comp.lang.perl.modules.

The definitive book on Perl DBI is the Cheetah book, so called because of the picture on the cover. Its proper title is 'Programming the Perl DBI: Database programming with Perl' by Alligator Descartes and Tim Bunce, published by O'Reilly Associates, February 2000, ISBN 1-56592-699-4. Buy it now if you have not already done so, and read it.

Locating drivers

Before writing a new driver, it is in your interests to find out whether there already is a driver for your database. If there is such a driver, it would be much easier to make use of it than to write your own!

The primary web-site for locating Perl software is <http://search.cpan.org/>. You should look under the various modules listings for the software you are after. For example:

http://search.cpan.org/modlist/Database_Interfaces

Follow the DBD:: and DBIx:: links at the top to see those subsets.

See the DBI docs for information on DBI web sites and mailing lists.

Registering a new driver

Before going through any official registration process, you will need to establish that there is no driver already in the works. You'll do that by asking the DBI mailing lists whether there is such a driver available, or whether anybody is working on one.

When you get the go ahead, you will need to establish the name of the driver and a prefix for the driver. Typically, the name is based on the name of the database software it uses, and the prefix is a contraction of that. Hence, DBD::Oracle has the name Oracle and the prefix 'ora_'. The prefix must be lowercase and contain no underscores other than the one at the end.

This information will be recorded in the DBI module. Apart from documentation purposes, registration is a prerequisite for installing private methods.

If you are writing a driver which will not be distributed on CPAN, then you should choose a prefix beginning with 'x_', to avoid potential prefix collisions with drivers registered in the future. Thus, if you wrote a non-CPAN distributed driver called DBD::CustomDB, the prefix might be 'x_cdb_'.

This document assumes you are writing a driver called DBD::Driver, and that the prefix 'drv_' is assigned to the driver.

Two styles of database driver

There are two distinct styles of database driver that can be written to work with the Perl DBI.

Your driver can be written in pure Perl, requiring no C compiler. When feasible, this is the best solution, but most databases are not written in such a way that this can be done. Some examples of pure Perl drivers are DBD::File and DBD::CSV.

Alternatively, and most commonly, your driver will need to use some C code to gain access to the database. This will be classified as a C/XS driver.

What code will you write?

There are a number of files that need to be written for either a pure Perl driver or a C/XS driver. There are no extra files needed only by a pure Perl driver, but there are several extra files needed only by a C/XS driver.

Files common to pure Perl and C/XS drivers

Assuming that your driver is called DBD::Driver, these files are:

- Makefile.PL
- META.yml
- README
- MANIFEST
- Driver.pm
- lib/Bundle/DBD/Driver.pm
- lib/DBD/Driver/Summary.pm
- t/*.t

The first four files are mandatory. Makefile.PL is used to control how the driver is built and installed. The README file tells people who download the file about how to build the module and any prerequisite software that must be installed. The MANIFEST file is used by the standard Perl module distribution mechanism. It lists all the source files that need to be distributed with your module. Driver.pm is what is loaded by the DBI code; it contains the methods peculiar to your driver.

Although the META.yml file is not required you are advised to create one. Of particular importance are the build_requires and configure_requires attributes which newer CPAN modules understand. You use these to tell the CPAN module (and CPANPLUS) that your build and configure mechanisms require DBI. The best reference for META.yml (at the time of writing) is <http://module-build.sourceforge.net/META-spec-v1.4.html>. You can find a reasonable example of a META.yml in DBD::ODBC.

The lib/Bundle/DBD/Driver.pm file allows you to specify other Perl modules on which yours depends in a format that allows someone to type a simple command and ensure that all the

pre-requisites are in place as well as building your driver.

The lib/DBD/Driver/Summary.pm file contains (an updated version of) the information that was included - or that would have been included - in the appendices of the Cheetah book as a summary of the abilities of your driver and the associated database.

The files in the t subdirectory are unit tests for your driver. You should write your tests as stringently as possible, while taking into account the diversity of installations that you can encounter:

? Your tests should not casually modify operational databases.

? You should never damage existing tables in a database.

? You should code your tests to use a constrained name space within the database. For example, the tables (and all other named objects) that are created could all begin with 'dbd_drv_'.

? At the end of a test run, there should be no testing objects left behind in the database.

? If you create any databases, you should remove them.

? If your database supports temporary tables that are automatically removed at the end of a session, then exploit them as often as possible.

? Try to make your tests independent of each other. If you have a test t/t11dowhat.t that depends upon the successful running of t/t10thingamy.t, people cannot run the single test case t/t11dowhat.t. Further, running t/t11dowhat.t twice in a row is likely to fail (at least, if t/t11dowhat.t modifies the database at all) because the database at the start of the second run is not what you saw at the start of the first run.

? Document in your README file what you do, and what privileges people need to do it.

? You can, and probably should, sequence your tests by including a test number before an abbreviated version of the test name; the tests are run in the order in which the names are expanded by shell-style globbing.

? It is in your interests to ensure that your tests work as widely as possible.

Many drivers also install sub-modules DBD::Driver::SubModule for any of a variety of different reasons, such as to support the metadata methods (see the discussion of "METADATA METHODS" below). Such sub-modules are conventionally stored in the directory lib/DBD/Driver. The module itself would usually be in a file SubModule.pm. All such sub-modules should themselves be version stamped (see the discussions far below).

Extra files needed by C/XS drivers

The software for a C/XS driver will typically contain at least four extra files that are not relevant to a pure Perl driver.

- Driver.xs
- Driver.h
- dbdimp.h
- dbdimp.c

The Driver.xs file is used to generate C code that Perl can call to gain access to the C functions you write that will, in turn, call down onto your database software.

The Driver.h header is a stylized header that ensures you can access the necessary Perl and DBI macros, types, and function declarations.

The dbdimp.h is used to specify which functions have been implemented by your driver.

The dbdimp.c file is where you write the C code that does the real work of translating between Perl-ish data types and what the database expects to use and return.

There are some (mainly small, but very important) differences between the contents of Makefile.PL and Driver.pm for pure Perl and C/XS drivers, so those files are described both in the section on creating a pure Perl driver and in the section on creating a C/XS driver.

Obviously, you can add extra source code files to the list.

Requirements on a driver and driver writer

To be remotely useful, your driver must be implemented in a format that allows it to be distributed via CPAN, the Comprehensive Perl Archive Network (<http://www.cpan.org/> and <http://search.cpan.org>). Of course, it is easier if you do not have to meet this criterion, but you will not be able to ask for much help if you do not do so, and no-one is likely to want to install your module if they have to learn a new installation mechanism.

CREATING A PURE PERL DRIVER

Writing a pure Perl driver is surprisingly simple. However, there are some problems you should be aware of. The best option is of course picking up an existing driver and carefully modifying one method after the other.

Also look carefully at DBD::AnyData and DBD::Template.

As an example we take a look at the DBD::File driver, a driver for accessing plain files as tables, which is part of the DBD::CSV package.

The minimal set of files we have to implement are Makefile.PL, README, MANIFEST and Driver.pm.

Pure Perl version of Makefile.PL

You typically start with writing Makefile.PL, a Makefile generator. The contents of this file are described in detail in the ExtUtils::MakeMaker man pages. It is definitely a good idea if you start reading them. At least you should know about the variables CONFIGURE, DEFINED, PM, DIR, EXE_FILES, INC, LIBS, LINKTYPE, NAME, OPTIMIZE, PL_FILES, VERSION, VERSION_FROM, clean, depend, realclean from the ExtUtils::MakeMaker man page: these are used in almost any Makefile.PL.

Additionally read the section on Overriding MakeMaker Methods and the descriptions of the distcheck, disttest and dist targets: They will definitely be useful for you.

Of special importance for DBI drivers is the postamble method from the ExtUtils::MM_Unix man page.

For Emacs users, I recommend the libscan method, which removes Emacs backup files (file names which end with a tilde '~') from lists of files.

Now an example, I use the word "Driver" wherever you should insert your driver's name:

```
# -*- perl -*-
use ExtUtils::MakeMaker;
WriteMakefile(
    dbd_edit_mm_attribs( {
        'NAME'         => 'DBD::Driver',
        'VERSION_FROM' => 'Driver.pm',
        'INC'          => '',
        'dist'         => { 'SUFFIX'   => '.gz',
                            'COMPRESS' => 'gzip -9f' },
        'realclean'    => { FILES => '*.xsi' },
        'PREREQ_PM'    => '1.03',
        'CONFIGURE'    => sub {
            eval {require DBI::DBD;};
            if ($@) {
                warn $@;
                exit 0;
            }
```

```
            my $dbi_arch_dir = dbd_dbi_arch_dir();
            if (exists($opts{INC})) {
                return {INC => "$opts{INC} -I$dbi_arch_dir"};
            } else {
                return {INC => "-I$dbi_arch_dir"};
            }
        }
    },
    { create_pp_tests => 1})
);
package MY;
sub postamble { return main::dbd_postamble(@_); }
sub libscan {
    my ($self, $path) = @_;
    ($path =~ m/\~$/) ? undef : $path;
}
```

Note the calls to "dbd_edit_mm_attribs()" and "dbd_postamble()".

The second hash reference in the call to "dbd_edit_mm_attribs()" (containing "create_pp_tests()") is optional; you should not use it unless your driver is a pure Perl driver (that is, it does not use C and XS code). Therefore, the call to "dbd_edit_mm_attribs()" is not relevant for C/XS drivers and may be omitted; simply use the (single) hash reference containing NAME etc as the only argument to "WriteMakefile()".

Note that the "dbd_edit_mm_attribs()" code will fail if you do not have a t sub-directory containing at least one test case.

PREREQ_PM tells MakeMaker that DBI (version 1.03 in this case) is required for this module. This will issue a warning that DBI 1.03 is missing if someone attempts to install your DBD without DBI 1.03. See CONFIGURE below for why this does not work reliably in stopping cpan testers failing your module if DBI is not installed.

CONFIGURE is a subroutine called by MakeMaker during "WriteMakefile". By putting the "require DBI::DBD" in this section we can attempt to load DBI::DBD but if it is missing we exit with success. As we exit successfully without creating a Makefile when DBI::DBD is missing cpan testers will not report a failure. This may seem at odds with PREREQ_PM but PREREQ_PM does not cause "WriteMakefile" to fail (unless you also specify PREREQ_FATAL

which is strongly discouraged by MakeMaker) so "WriteMakefile" would continue to call "dbd_dbi_arch_dir" and fail.

All drivers must use "dbd_postamble()" or risk running into problems.

Note the specification of VERSION_FROM; the named file (Driver.pm) will be scanned for the first line that looks like an assignment to $VERSION, and the subsequent text will be used to determine the version number. Note the commentary in ExtUtils::MakeMaker on the subject of correctly formatted version numbers.

If your driver depends upon external software (it usually will), you will need to add code to ensure that your environment is workable before the call to "WriteMakefile()". If you need to check for the existence of an external library and perhaps modify INC to include the paths to where the external library header files are located and you cannot find the library or header files make sure you output a message saying they cannot be found but "exit 0" (success) before calling "WriteMakefile" or CPAN testers will fail your module if the external library is not found.

A full-fledged Makefile.PL can be quite large (for example, the files for DBD::Oracle and DBD::Informix are both over 1000 lines long, and the Informix one uses - and creates - auxiliary modules too).

See also ExtUtils::MakeMaker and ExtUtils::MM_Unix. Consider using CPAN::MakeMaker in place of ExtUtils::MakeMaker.

README

The README file should describe what the driver is for, the pre-requisites for the build process, the actual build process, how to report errors, and who to report them to.

Users will find ways of breaking the driver build and test process which you would never even have dreamed to be possible in your worst nightmares. Therefore, you need to write this document defensively, precisely and concisely.

As always, use the README from one of the established drivers as a basis for your own; the version in DBD::Informix is worth a look as it has been quite successful in heading off problems.

? Note that users will have versions of Perl and DBI that are both older and newer than you expected, but this will seldom cause much trouble. When it does, it will be because you are using features of DBI that are not supported in the version they are using.

? Note that users will have versions of the database software that are both older and

newer than you expected. You will save yourself time in the long run if you can identify the range of versions which have been tested and warn about versions which are not known to be OK.

? Note that many people trying to install your driver will not be experts in the database software.

? Note that many people trying to install your driver will not be experts in C or Perl.

MANIFEST

The MANIFEST will be used by the Makefile's dist target to build the distribution tar file that is uploaded to CPAN. It should list every file that you want to include in your distribution, one per line.

lib/Bundle/DBD/Driver.pm

The CPAN module provides an extremely powerful bundle mechanism that allows you to specify pre-requisites for your driver.

The primary pre-requisite is Bundle::DBI; you may want or need to add some more. With the bundle set up correctly, the user can type:

```
perl -MCPAN -e 'install Bundle::DBD::Driver'
```

and Perl will download, compile, test and install all the Perl modules needed to build your driver.

The prerequisite modules are listed in the "CONTENTS" section, with the official name of the module followed by a dash and an informal name or description.

? Listing Bundle::DBI as the main pre-requisite simplifies life.

? Don't forget to list your driver.

? Note that unless the DBMS is itself a Perl module, you cannot list it as a pre-requisite in this file.

? You should keep the version of the bundle the same as the version of your driver.

? You should add configuration management, copyright, and licencing information at the top.

A suitable skeleton for this file is shown below.

```
package Bundle::DBD::Driver;
$VERSION = '0.01';
1;
__END__
=head1 NAME
```

Bundle::DBD::Driver - A bundle to install all DBD::Driver related modules

=head1 SYNOPSIS

C<perl -MCPAN -e 'install Bundle::DBD::Driver'>

=head1 CONTENTS

Bundle::DBI - Bundle for DBI by TIMB (Tim Bunce)

DBD::Driver - DBD::Driver by YOU (Your Name)

=head1 DESCRIPTION

This bundle includes all the modules used by the Perl Database

Interface (DBI) driver for Driver (DBD::Driver), assuming the

use of DBI version 1.13 or later, created by Tim Bunce.

If you've not previously used the CPAN module to install any

bundles, you will be interrogated during its setup phase.

But when you've done it once, it remembers what you told it.

You could start by running:

  C<perl -MCPAN -e 'install Bundle::CPAN'>

=head1 SEE ALSO

Bundle::DBI

=head1 AUTHOR

Your Name E<lt>F<you@yourdomain.com>E<gt>

=head1 THANKS

This bundle was created by ripping off Bundle::libnet created by

Graham Barr E<lt>F<gbarr@ti.com>E<gt>, and radically simplified

with some information from Jochen Wiedmann E<lt>F<joe@ispsoft.de>E<gt>.

The template was then included in the DBI::DBD documentation by

Jonathan Leffler E<lt>F<jleffler@informix.com>E<gt>.

=cut

lib/DBD/Driver/Summary.pm

There is no substitute for taking the summary file from a driver that was documented in the Perl book (such as DBD::Oracle or DBD::Informix or DBD::ODBC, to name but three), and adapting it to describe the facilities available via DBD::Driver when accessing the Driver database.

Pure Perl version of Driver.pm

The Driver.pm file defines the Perl module DBD::Driver for your driver. It will define a

package DBD::Driver along with some version information, some variable definitions, and a function "driver()" which will have a more or less standard structure.

It will also define three sub-packages of DBD::Driver:

DBD::Driver::dr

    with methods "connect()", "data_sources()" and "disconnect_all()";

DBD::Driver::db

    with methods such as "prepare()";

DBD::Driver::st

    with methods such as "execute()" and "fetch()".

The Driver.pm file will also contain the documentation specific to DBD::Driver in the format used by perldoc.

In a pure Perl driver, the Driver.pm file is the core of the implementation. You will need to provide all the key methods needed by DBI.

Now let's take a closer look at an excerpt of File.pm as an example. We ignore things that are common to any module (even non-DBI modules) or really specific to the DBD::File package.

The DBD::Driver package

The header

```
  package DBD::File;
  use strict;
  use vars qw($VERSION $drh);
  $VERSION = "1.23.00"  # Version number of DBD::File
```

This is where the version number of your driver is specified, and is where Makefile.PL looks for this information. Please ensure that any other modules added with your driver are also version stamped so that CPAN does not get confused.

It is recommended that you use a two-part (1.23) or three-part (1.23.45) version number. Also consider the CPAN system, which gets confused and considers version 1.10 to precede version 1.9, so that using a raw CVS, RCS or SCCS version number is probably not appropriate (despite being very common).

For Subversion you could use:

```
  $VERSION = "12.012346";
```

(use lots of leading zeros on the second portion so if you move the code to a shared repository like svn.perl.org the much larger revision numbers won't cause a problem, at

least not for a few years). For RCS or CVS you can use:

```
  $VERSION = "11.22";
```

which pads out the fractional part with leading zeros so all is well (so long as you don't go past x.99)

```
  $drh = undef;         # holds driver handle once initialized
```

This is where the driver handle will be stored, once created. Note that you may assume there is only one handle for your driver.

The driver constructor

The "driver()" method is the driver handle constructor. Note that the "driver()" method is in the DBD::Driver package, not in one of the sub-packages DBD::Driver::dr, DBD::Driver::db, or DBD::Driver::db.

```
  sub driver
  {
      return $drh if $drh;      # already created - return same one
      my ($class, $attr) = @_;
      $class .= "::dr";
      DBD::Driver::db->install_method('drv_example_dbh_method');
      DBD::Driver::st->install_method('drv_example_sth_method');
      # not a 'my' since we use it above to prevent multiple drivers
      $drh = DBI::_new_drh($class, {
              'Name'        => 'File',
              'Version'     => $VERSION,
              'Attribution' => 'DBD::File by Jochen Wiedmann',
           })
          or return undef;
      return $drh;
  }
```

This is a reasonable example of how DBI implements its handles. There are three kinds: driver handles (typically stored in $drh; from now on called drh or $drh), database handles (from now on called dbh or $dbh) and statement handles (from now on called sth or $sth).

The prototype of "DBI::_new_drh()" is

```
  $drh = DBI::_new_drh($class, $public_attrs, $private_attrs);
```

with the following arguments:

$class

is typically the class for your driver, (for example, "DBD::File::dr"), passed as the first argument to the "driver()" method.

$public_attrs

is a hash ref to attributes like Name, Version, and Attribution. These are processed and used by DBI. You had better not make any assumptions about them nor should you add private attributes here.

$private_attrs

This is another (optional) hash ref with your private attributes. DBI will store them and otherwise leave them alone.

The "DBI::_new_drh()" method and the "driver()" method both return "undef" for failure (in which case you must look at $DBI::err and $DBI::errstr for the failure information, because you have no driver handle to use).

Using install_method() to expose driver-private methods

DBD::Foo::db->install_method($method_name, \%attr);

Installs the driver-private method named by $method_name into the DBI method dispatcher so it can be called directly, avoiding the need to use the func() method.

It is called as a static method on the driver class to which the method belongs. The method name must begin with the corresponding registered driver-private prefix. For example, for DBD::Oracle $method_name must being with "'ora_'", and for DBD::AnyData it must begin with "'ad_'".

The "\%attr" attributes can be used to provide fine control over how the DBI dispatcher handles the dispatching of the method. However it's undocumented at the moment. See the IMA_* #define's in DBI.xs and the O=>0x000x values in the initialization of %DBI::DBI_methods in DBI.pm. (Volunteers to polish up and document the interface are very welcome to get in touch via dbi-dev@perl.org).

Methods installed using install_method default to the standard error handling behaviour for DBI methods: clearing err and errstr before calling the method, and checking for errors to trigger RaiseError etc. on return. This differs from the default behaviour of func().

Note for driver authors: The DBD::Foo::xx->install_method call won't work until the class-hierarchy has been setup. Normally the DBI looks after that just after the driver is

loaded. This means install_method() can't be called at the time the driver is loaded unless the class-hierarchy is set up first. The way to do that is to call the setup_driver() method:

```
    DBI->setup_driver('DBD::Foo');
```

before using install_method().

The CLONE special subroutine

Also needed here, in the DBD::Driver package, is a "CLONE()" method that will be called by perl when an interpreter is cloned. All your "CLONE()" method needs to do, currently, is clear the cached $drh so the new interpreter won't start using the cached $drh from the old interpreter:

```
  sub CLONE {
    undef $drh;
  }
```

See <http://search.cpan.org/dist/perl/pod/perlmod.pod#Making_your_module_threadsafe> for details.

The DBD::Driver::dr package

The next lines of code look as follows:

```
  package DBD::Driver::dr; # ====== DRIVER ======
  $DBD::Driver::dr::imp_data_size = 0;
```

Note that no @ISA is needed here, or for the other DBD::Driver::* classes, because the DBI takes care of that for you when the driver is loaded.

FIX ME Explain what the imp_data_size is, so that implementors aren't practicing cargo-cult programming.

The database handle constructor

The database handle constructor is the driver's (hence the changed namespace) "connect()" method:

```
  sub connect
  {
      my ($drh, $dr_dsn, $user, $auth, $attr) = @_;
      # Some database specific verifications, default settings
      # and the like can go here. This should only include
      # syntax checks or similar stuff where it's legal to
      # 'die' in case of errors.
```

```
    # For example, many database packages requires specific
    # environment variables to be set; this could be where you
    # validate that they are set, or default them if they are not set.
    my $driver_prefix = "drv_"; # the assigned prefix for this driver
    # Process attributes from the DSN; we assume ODBC syntax
    # here, that is, the DSN looks like var1=val1;...;varN=valN
    foreach my $var ( split /;/, $dr_dsn ) {
        my ($attr_name, $attr_value) = split '=', $var, 2;
        return $drh->set_err($DBI::stderr, "Can't parse DSN part '$var'")
            unless defined $attr_value;
        # add driver prefix to attribute name if it doesn't have it already
        $attr_name = $driver_prefix.$attr_name
            unless $attr_name =~ /^$driver_prefix/o;
        # Store attribute into %$attr, replacing any existing value.
        # The DBI will STORE() these into $dbh after we've connected
        $attr->{$attr_name} = $attr_value;
    }
    # Get the attributes we'll use to connect.
    # We use delete here because these no need to STORE them
    my $db = delete $attr->{drv_database} || delete $attr->{drv_db}
        or return $drh->set_err($DBI::stderr, "No database name given in DSN '$dr_dsn'");
    my $host = delete $attr->{drv_host} || 'localhost';
    my $port = delete $attr->{drv_port} || 123456;
    # Assume you can attach to your database via drv_connect:
    my $connection = drv_connect($db, $host, $port, $user, $auth)
        or return $drh->set_err($DBI::stderr, "Can't connect to $dr_dsn: ...");
    # create a 'blank' dbh (call superclass constructor)
    my ($outer, $dbh) = DBI::_new_dbh($drh, { Name => $dr_dsn });
    $dbh->STORE('Active', 1 );
    $dbh->{drv_connection} = $connection;
    return $outer;
}
```

This is mostly the same as in the driver handle constructor above. The arguments are

described in DBI.

The constructor "DBI::_new_dbh()" is called, returning a database handle. The constructor's prototype is:

```
  ($outer, $inner) = DBI::_new_dbh($drh, $public_attr, $private_attr);
```

with similar arguments to those in the driver handle constructor, except that the $class is replaced by $drh. The Name attribute is a standard DBI attribute (see "Database Handle Attributes" in DBI).

In scalar context, only the outer handle is returned.

Note the use of the "STORE()" method for setting the dbh attributes. That's because within the driver code, the handle object you have is the 'inner' handle of a tied hash, not the outer handle that the users of your driver have.

Because you have the inner handle, tie magic doesn't get invoked when you get or set values in the hash. This is often very handy for speed when you want to get or set simple non-special driver-specific attributes.

However, some attribute values, such as those handled by the DBI like PrintError, don't actually exist in the hash and must be read via "$h->FETCH($attrib)" and set via "$h->STORE($attrib, $value)". If in any doubt, use these methods.

The data_sources() method

The "data_sources()" method must populate and return a list of valid data sources, prefixed with the "dbi:Driver" incantation that allows them to be used in the first argument of the "DBI->connect()" method. An example of this might be scanning the $HOME/.odbcini file on Unix for ODBC data sources (DSNs).

As a trivial example, consider a fixed list of data sources:

```
  sub data_sources
  {
      my($drh, $attr) = @_;
      my(@list) = ();
      # You need more sophisticated code than this to set @list...
      push @list, "dbi:Driver:abc";
      push @list, "dbi:Driver:def";
      push @list, "dbi:Driver:ghi";
      # End of code to set @list
      return @list;
```

```
}
```

The disconnect_all() method

If you need to release any resources when the driver is unloaded, you can provide a disconnect_all method.

Other driver handle methods

If you need any other driver handle methods, they can follow here.

Error handling

It is quite likely that something fails in the connect method. With DBD::File for example, you might catch an error when setting the current directory to something not existent by using the (driver-specific) f_dir attribute.

To report an error, you use the "set_err()" method:

```
$h->set_err($err, $errmsg, $state);
```

This will ensure that the error is recorded correctly and that RaiseError and PrintError etc are handled correctly.

Typically you'll always use the method instance, aka your method's first argument.

As "set_err()" always returns "undef" your error handling code can usually be simplified to something like this:

```
return $h->set_err($err, $errmsg, $state) if ...;
```

The DBD::Driver::db package

```
package DBD::Driver::db; # ====== DATABASE ======
$DBD::Driver::db::imp_data_size = 0;
```

The statement handle constructor

There's nothing much new in the statement handle constructor, which is the "prepare()" method:

```
sub prepare
{
    my ($dbh, $statement, @attribs) = @_;
    # create a 'blank' sth
    my ($outer, $sth) = DBI::_new_sth($dbh, { Statement => $statement });
    $sth->STORE('NUM_OF_PARAMS', ($statement =~ tr/?//));
    $sth->{drv_params} = [];
    return $outer;
}
```

This is still the same -- check the arguments and call the super class constructor "DBI::_new_sth()". Again, in scalar context, only the outer handle is returned. The Statement attribute should be cached as shown.

Note the prefix drv_ in the attribute names: it is required that all your private attributes use a lowercase prefix unique to your driver. As mentioned earlier in this document, the DBI contains a registry of known driver prefixes and may one day warn about unknown attributes that don't have a registered prefix.

Note that we parse the statement here in order to set the attribute NUM_OF_PARAMS. The technique illustrated is not very reliable; it can be confused by question marks appearing in quoted strings, delimited identifiers or in SQL comments that are part of the SQL statement. We could set NUM_OF_PARAMS in the "execute()" method instead because the DBI specification explicitly allows a driver to defer this, but then the user could not call "bind_param()".

Transaction handling

Pure Perl drivers will rarely support transactions. Thus your "commit()" and "rollback()" methods will typically be quite simple:

```
 sub commit
 {
     my ($dbh) = @_;
     if ($dbh->FETCH('Warn')) {
         warn("Commit ineffective while AutoCommit is on");
     }
     0;
 }
 sub rollback {
     my ($dbh) = @_;
     if ($dbh->FETCH('Warn')) {
         warn("Rollback ineffective while AutoCommit is on");
     }
     0;
 }
```

Or even simpler, just use the default methods provided by the DBI that do nothing except return "undef".

The DBI's default "begin_work()" method can be used by inheritance.

The STORE() and FETCH() methods

These methods (that we have already used, see above) are called for you, whenever the user does a:

```
  $dbh->{$attr} = $val;
```

or, respectively,

```
  $val = $dbh->{$attr};
```

See perltie for details on tied hash refs to understand why these methods are required.

The DBI will handle most attributes for you, in particular attributes like RaiseError or PrintError. All you have to do is handle your driver's private attributes and any attributes, like AutoCommit and ChopBlanks, that the DBI can't handle for you.

A good example might look like this:

```
  sub STORE
 {
     my ($dbh, $attr, $val) = @_;
     if ($attr eq 'AutoCommit') {
         # AutoCommit is currently the only standard attribute we have
         # to consider.
         if (!$val) { die "Can't disable AutoCommit"; }
         return 1;
     }
     if ($attr =~ m/^drv_/) {
         # Handle only our private attributes here
         # Note that we could trigger arbitrary actions.
         # Ideally we should warn about unknown attributes.
         $dbh->{$attr} = $val; # Yes, we are allowed to do this,
         return 1;             # but only for our private attributes
     }
     # Else pass up to DBI to handle for us
     $dbh->SUPER::STORE($attr, $val);
 }
  sub FETCH
 {
```

```
    my ($dbh, $attr) = @_;
    if ($attr eq 'AutoCommit') { return 1; }
    if ($attr =~ m/^drv_/) {
        # Handle only our private attributes here
        # Note that we could trigger arbitrary actions.
        return $dbh->{$attr}; # Yes, we are allowed to do this,
                              # but only for our private attributes
    }
    # Else pass up to DBI to handle
    $dbh->SUPER::FETCH($attr);
}
```

The DBI will actually store and fetch driver-specific attributes (with all lowercase names) without warning or error, so there's actually no need to implement driver-specific any code in your "FETCH()" and "STORE()" methods unless you need extra logic/checks, beyond getting or setting the value.

Unless your driver documentation indicates otherwise, the return value of the "STORE()" method is unspecified and the caller shouldn't use that value.

Other database handle methods

As with the driver package, other database handle methods may follow here. In particular you should consider a (possibly empty) "disconnect()" method and possibly a "quote()" method if DBI's default isn't correct for you. You may also need the "type_info_all()" and "get_info()" methods, as described elsewhere in this document.

Where reasonable use "$h->SUPER::foo()" to call the DBI's method in some or all cases and just wrap your custom behavior around that.

If you want to use private trace flags you'll probably want to be able to set them by name. To do that you'll need to define a "parse_trace_flag()" method (note that's "parse_trace_flag", singular, not "parse_trace_flags", plural).

```
sub parse_trace_flag {
    my ($h, $name) = @_;
    return 0x01000000 if $name eq 'foo';
    return 0x02000000 if $name eq 'bar';
    return 0x04000000 if $name eq 'baz';
    return 0x08000000 if $name eq 'boo';
```

```
    return 0x10000000 if $name eq 'bop';
    return $h->SUPER::parse_trace_flag($name);
}
```

All private flag names must be lowercase, and all private flags must be in the top 8 of the 32 bits.

The DBD::Driver::st package

This package follows the same pattern the others do:

```
package DBD::Driver::st;
$DBD::Driver::st::imp_data_size = 0;
```

The execute() and bind_param() methods

This is perhaps the most difficult method because we have to consider parameter bindings here. In addition to that, there are a number of statement attributes which must be set for inherited DBI methods to function correctly (see "Statement attributes" below).

We present a simplified implementation by using the drv_params attribute from above:

```
sub bind_param
{
    my ($sth, $pNum, $val, $attr) = @_;
    my $type = (ref $attr) ? $attr->{TYPE} : $attr;
    if ($type) {
        my $dbh = $sth->{Database};
        $val = $dbh->quote($sth, $type);
    }
    my $params = $sth->{drv_params};
    $params->[$pNum-1] = $val;
    1;
}
sub execute
{
    my ($sth, @bind_values) = @_;
    # start of by finishing any previous execution if still active
    $sth->finish if $sth->FETCH('Active');
    my $params = (@bind_values) ?
        \@bind_values : $sth->{drv_params};
```

```
    my $numParam = $sth->FETCH('NUM_OF_PARAMS');
    return $sth->set_err($DBI::stderr, "Wrong number of parameters")
        if @$params != $numParam;
    my $statement = $sth->{'Statement'};
    for (my $i = 0;  $i < $numParam;  $i++) {
        $statement =~ s/?/$params->[$i]/; # XXX doesn't deal with quoting etc!
    }
    # Do anything ... we assume that an array ref of rows is
    # created and store it:
    $sth->{'drv_data'} = $data;
    $sth->{'drv_rows'} = @$data; # number of rows
    $sth->STORE('NUM_OF_FIELDS') = $numFields;
    $sth->{Active} = 1;
    @$data || '0E0';
}
```

There are a number of things you should note here.

We initialize the NUM_OF_FIELDS and Active attributes here, because they are essential for "bind_columns()" to work.

We use attribute "$sth->{Statement}" which we created within "prepare()". The attribute "$sth->{Database}", which is nothing else than the dbh, was automatically created by DBI.

Finally, note that (as specified in the DBI specification) we return the string '0E0' instead of the number 0, so that the result tests true but equal to zero.

```
 $sth->execute() or die $sth->errstr;
```

The execute_array(), execute_for_fetch() and bind_param_array() methods

In general, DBD's only need to implement "execute_for_fetch()" and "bind_param_array". DBI's default "execute_array()" will invoke the DBD's "execute_for_fetch()" as needed.

The following sequence describes the interaction between DBI "execute_array" and a DBD's "execute_for_fetch":

1. App calls "$sth->execute_array(\%attrs, @array_of_arrays)"
2. If @array_of_arrays was specified, DBI processes @array_of_arrays by calling DBD's "bind_param_array()". Alternately, App may have directly called "bind_param_array()"
3. DBD validates and binds each array
4. DBI retrieves the validated param arrays from DBD's ParamArray attribute

5. DBI calls DBD's "execute_for_fetch($fetch_tuple_sub, \@tuple_status)", where &$fetch_tuple_sub is a closure to iterate over the returned ParamArray values, and "\@tuple_status" is an array to receive the disposition status of each tuple.
6. DBD iteratively calls &$fetch_tuple_sub to retrieve parameter tuples to be added to its bulk database operation/request.
7. when DBD reaches the limit of tuples it can handle in a single database operation/request, or the &$fetch_tuple_sub indicates no more tuples by returning undef, the DBD executes the bulk operation, and reports the disposition of each tuple in \@tuple_status.
8. DBD repeats steps 6 and 7 until all tuples are processed.

E.g., here's the essence of DBD::Oracle's execute_for_fetch:

```
    while (1) {
        my @tuple_batch;
        for (my $i = 0; $i < $batch_size; $i++) {
            push @tuple_batch, [ @{$fetch_tuple_sub->() || last} ];
        }
        last unless @tuple_batch;
        my $res = ora_execute_array($sth, \@tuple_batch,
          scalar(@tuple_batch), $tuple_batch_status);
        push @$tuple_status, @$tuple_batch_status;
    }
```

Note that DBI's default execute_array()/execute_for_fetch() implementation requires the use of positional (i.e., '?') placeholders. Drivers which require named placeholders must either emulate positional placeholders (e.g., see DBD::Oracle), or must implement their own execute_array()/execute_for_fetch() methods to properly sequence bound parameter arrays.

Fetching data

Only one method needs to be written for fetching data, "fetchrow_arrayref()". The other methods, "fetchrow_array()", "fetchall_arrayref()", etc, as well as the database handle's "select*" methods are part of DBI, and call "fetchrow_arrayref()" as necessary.

```
 sub fetchrow_arrayref
 {
    my ($sth) = @_;
```

```
    my $data = $sth->{drv_data};
    my $row = shift @$data;
    if (!$row) {
        $sth->STORE(Active => 0); # mark as no longer active
        return undef;
    }
    if ($sth->FETCH('ChopBlanks')) {
        map { $_ =~ s/\s+$//; } @$row;
    }
    return $sth->_set_fbav($row);
}
 *fetch = \&fetchrow_arrayref; # required alias for fetchrow_arrayref
```

Note the use of the method "_set_fbav()" -- this is required so that "bind_col()" and "bind_columns()" work.

If an error occurs which leaves the $sth in a state where remaining rows can't be fetched then Active should be turned off before the method returns.

The "rows()" method for this driver can be implemented like this:

```
 sub rows { shift->{drv_rows} }
```

because it knows in advance how many rows it has fetched. Alternatively you could delete that method and so fallback to the DBI's own method which does the right thing based on the number of calls to "_set_fbav()".

The more_results method

If your driver doesn't support multiple result sets, then don't even implement this method.

Otherwise, this method needs to get the statement handle ready to fetch results from the next result set, if there is one. Typically you'd start with:

```
  $sth->finish;
```

then you should delete all the attributes from the attribute cache that may no longer be relevant for the new result set:

```
  delete $sth->{$_}
      for qw(NAME TYPE PRECISION SCALE ...);
```

for drivers written in C use:

```
  hv_delete((HV*)SvRV(sth), "NAME", 4, G_DISCARD);
```

```
    hv_delete((HV*)SvRV(sth), "NULLABLE", 8, G_DISCARD);
    hv_delete((HV*)SvRV(sth), "NUM_OF_FIELDS", 13, G_DISCARD);
    hv_delete((HV*)SvRV(sth), "PRECISION", 9, G_DISCARD);
    hv_delete((HV*)SvRV(sth), "SCALE", 5, G_DISCARD);
    hv_delete((HV*)SvRV(sth), "TYPE", 4, G_DISCARD);
```

Don't forget to also delete, or update, any driver-private attributes that may not be correct for the next resultset.

The NUM_OF_FIELDS attribute is a special case. It should be set using STORE:

```
    $sth->STORE(NUM_OF_FIELDS => 0); /* for DBI <= 1.53 */
    $sth->STORE(NUM_OF_FIELDS => $new_value);
```

for drivers written in C use this incantation:

```
    /* Adjust NUM_OF_FIELDS - which also adjusts the row buffer size */
    DBIc_NUM_FIELDS(imp_sth) = 0; /* for DBI <= 1.53 */
    DBIc_STATE(imp_xxh)->set_attr_k(sth, sv_2mortal(newSVpvn("NUM_OF_FIELDS",13)), 0,
        sv_2mortal(newSViv(mysql_num_fields(imp_sth->result)))
    );
```

For DBI versions prior to 1.54 you'll also need to explicitly adjust the number of elements in the row buffer array ("DBIc_FIELDS_AV(imp_sth)") to match the new result set. Fill any new values with newSV(0) not &sv_undef. Alternatively you could free DBIc_FIELDS_AV(imp_sth) and set it to null, but that would mean bind_columns() wouldn't work across result sets.

Statement attributes

The main difference between dbh and sth attributes is, that you should implement a lot of attributes here that are required by the DBI, such as NAME, NULLABLE, TYPE, etc. See "Statement Handle Attributes" in DBI for a complete list.

Pay attention to attributes which are marked as read only, such as NUM_OF_PARAMS. These attributes can only be set the first time a statement is executed. If a statement is prepared, then executed multiple times, warnings may be generated.

You can protect against these warnings, and prevent the recalculation of attributes which might be expensive to calculate (such as the NAME and NAME_* attributes):

```
    my $storedNumParams = $sth->FETCH('NUM_OF_PARAMS');
    if (!defined $storedNumParams or $storedNumFields < 0) {
        $sth->STORE('NUM_OF_PARAMS') = $numParams;
```

```
        # Set other useful attributes that only need to be set once
        # for a statement, like $sth->{NAME} and $sth->{TYPE}
    }
```

One particularly important attribute to set correctly (mentioned in "ATTRIBUTES COMMON TO ALL HANDLES" in DBI is Active. Many DBI methods, including "bind_columns()", depend on this attribute.

Besides that the "STORE()" and "FETCH()" methods are mainly the same as above for dbh's.

Other statement methods

A trivial "finish()" method to discard stored data, reset any attributes (such as Active) and do "$sth->SUPER::finish()".

If you've defined a "parse_trace_flag()" method in ::db you'll also want it in ::st, so just alias it in:

```
  *parse_trace_flag = \&DBD::foo:db::parse_trace_flag;
```

And perhaps some other methods that are not part of the DBI specification, in particular to make metadata available. Remember that they must have names that begin with your drivers registered prefix so they can be installed using "install_method()".

If "DESTROY()" is called on a statement handle that's still active ("$sth->{Active}" is true) then it should effectively call "finish()".

```
    sub DESTROY {
        my $sth = shift;
        $sth->finish if $sth->FETCH('Active');
    }
```

Tests

The test process should conform as closely as possibly to the Perl standard test harness. In particular, most (all) of the tests should be run in the t sub-directory, and should simply produce an "ok" when run under "make test". For details on how this is done, see the Camel book and the section in Chapter 7, "The Standard Perl Library" on Test::Harness.

The tests may need to adapt to the type of database which is being used for testing, and to the privileges of the user testing the driver. For example, the DBD::Informix test code has to adapt in a number of places to the type of database to which it is connected as different Informix databases have different capabilities: some of the tests are for databases without transaction logs; others are for databases with a transaction log; some versions of the server have support for blobs, or stored procedures, or user-defined data

types, and others do not.

When a complete file of tests must be skipped, you can provide a reason in a pseudo-comment:

```
if ($no_transactions_available)
{
    print "1..0 # Skip: No transactions available\n";
    exit 0;
}
```

Consider downloading the DBD::Informix code and look at the code in DBD/Informix/TestHarness.pm which is used throughout the DBD::Informix tests in the t sub-directory.

CREATING A C/XS DRIVER

Please also see the section under "CREATING A PURE PERL DRIVER" regarding the creation of the Makefile.PL.

Creating a new C/XS driver from scratch will always be a daunting task. You can and should greatly simplify your task by taking a good reference driver implementation and modifying that to match the database product for which you are writing a driver.

The de facto reference driver has been the one for DBD::Oracle written by Tim Bunce, who is also the author of the DBI package. The DBD::Oracle module is a good example of a driver implemented around a C-level API.

Nowadays it it seems better to base on DBD::ODBC, another driver maintained by Tim and Jeff Urlwin, because it offers a lot of metadata and seems to become the guideline for the future development. (Also as DBD::Oracle digs deeper into the Oracle 8 OCI interface it'll get even more hairy than it is now.)

The DBD::Informix driver is one driver implemented using embedded SQL instead of a function-based API. DBD::Ingres may also be worth a look.

C/XS version of Driver.pm

A lot of the code in the Driver.pm file is very similar to the code for pure Perl modules - see above. However, there are also some subtle (and not so subtle) differences, including:

- The variables $DBD::Driver::{dr|db|st}::imp_data_size are not defined here, but in the XS code, because they declare the size of certain C structures.
- Some methods are typically moved to the XS code, in particular "prepare()",

"execute()", "disconnect()", "disconnect_all()" and the "STORE()" and "FETCH()" methods.

? Other methods are still part of Driver.pm, but have callbacks to the XS code.

? If the driver-specific parts of the imp_drh_t structure need to be formally initialized (which does not seem to be a common requirement), then you need to add a call to an appropriate XS function in the driver method of "DBD::Driver::driver()", and you define the corresponding function in Driver.xs, and you define the C code in dbdimp.c and the prototype in dbdimp.h.

For example, DBD::Informix has such a requirement, and adds the following call after the call to "_new_drh()" in Informix.pm:

```
DBD::Informix::dr::driver_init($drh);
```

and the following code in Informix.xs:

```
# Initialize the DBD::Informix driver data structure
void
driver_init(drh)
    SV *drh
    CODE:
    ST(0) = dbd_ix_dr_driver_init(drh) ? &sv_yes : &sv_no;
```

and the code in dbdimp.h declares:

```
extern int dbd_ix_dr_driver_init(SV *drh);
```

and the code in dbdimp.ec (equivalent to dbdimp.c) defines:

```
/* Formally initialize the DBD::Informix driver structure */
int
dbd_ix_dr_driver(SV *drh)
{
    D_imp_drh(drh);
    imp_drh->n_connections = 0;       /* No active connections */
    imp_drh->current_connection = 0;  /* No current connection */
    imp_drh->multipleconnections = (ESQLC_VERSION >= 600) ? True : False;
    dbd_ix_link_newhead(&imp_drh->head);  /* Empty linked list of connections */
    return 1;
}
```

DBD::Oracle has a similar requirement but gets around it by checking whether the

private data part of the driver handle is all zeroed out, rather than add extra functions.

Now let's take a closer look at an excerpt from Oracle.pm (revised heavily to remove idiosyncrasies) as an example, ignoring things that were already discussed for pure Perl drivers.

The connect method

The connect method is the database handle constructor. You could write either of two versions of this method: either one which takes connection attributes (new code) and one which ignores them (old code only).

If you ignore the connection attributes, then you omit all mention of the $auth variable (which is a reference to a hash of attributes), and the XS system manages the differences for you.

```
 sub connect
 {
     my ($drh, $dbname, $user, $auth, $attr) = @_;
     # Some database specific verifications, default settings
     # and the like following here. This should only include
     # syntax checks or similar stuff where it's legal to
     # 'die' in case of errors.
     my $dbh = DBI::_new_dbh($drh, {
           'Name'   => $dbname,
         })
         or return undef;
     # Call the driver-specific function _login in Driver.xs file which
     # calls the DBMS-specific function(s) to connect to the database,
     # and populate internal handle data.
     DBD::Driver::db::_login($dbh, $dbname, $user, $auth, $attr)
         or return undef;
     $dbh;
 }
```

This is mostly the same as in the pure Perl case, the exception being the use of the private "_login()" callback, which is the function that will really connect to the database. It is implemented in Driver.xst (you should not implement it) and calls

"dbd_db_login6()" or "dbd_db_login6_sv" from dbdimp.c. See below for details.

If your driver has driver-specific attributes which may be passed in the connect method and hence end up in $attr in "dbd_db_login6" then it is best to delete any you process so DBI does not send them again via STORE after connect. You can do this in C like this:

```
DBD_ATTRIB_DELETE(attr, "my_attribute_name",
                  strlen("my_attribute_name"));
```

However, prior to DBI subversion version 11605 (and fixed post 1.607) DBD_ATTRIB_DELETE segfaulted so if you cannot guarantee the DBI version will be post 1.607 you need to use:

```
hv_delete((HV*)SvRV(attr), "my_attribute_name",
                   strlen("my_attribute_name"), G_DISCARD);
```

FIX ME Discuss removing attributes in Perl code.

The disconnect_all method

FIX ME T.B.S

The data_sources method

If your "data_sources()" method can be implemented in pure Perl, then do so because it is easier than doing it in XS code (see the section above for pure Perl drivers).

If your "data_sources()" method must call onto compiled functions, then you will need to define dbd_dr_data_sources in your dbdimp.h file, which will trigger Driver.xst (in DBI v1.33 or greater) to generate the XS code that calls your actual C function (see the discussion below for details) and you do not code anything in Driver.pm to handle it.

The prepare method

The prepare method is the statement handle constructor, and most of it is not new. Like the "connect()" method, it now has a C callback:

```
package DBD::Driver::db; # ====== DATABASE ======
use strict;
sub prepare
{
    my ($dbh, $statement, $attribs) = @_;
    # create a 'blank' sth
    my $sth = DBI::_new_sth($dbh, {
        'Statement' => $statement,
        })
        or return undef;
```

```
    # Call the driver-specific function _prepare in Driver.xs file
    # which calls the DBMS-specific function(s) to prepare a statement
    # and populate internal handle data.
    DBD::Driver::st::_prepare($sth, $statement, $attribs)
        or return undef;
    $sth;
}
```

The execute method

FIX ME T.B.S

The fetchrow_arrayref method

FIX ME T.B.S

Other methods?

FIX ME T.B.S

Driver.xs

Driver.xs should look something like this:

```
#include "Driver.h"
DBISTATE_DECLARE;
INCLUDE: Driver.xsi
MODULE = DBD::Driver    PACKAGE = DBD::Driver::dr
/* Non-standard drh XS methods following here, if any.       */
/* If none (the usual case), omit the MODULE line above too. */
MODULE = DBD::Driver    PACKAGE = DBD::Driver::db
/* Non-standard dbh XS methods following here, if any.       */
/* Currently this includes things like _list_tables from     */
/* DBD::mSQL and DBD::mysql.                                 */
MODULE = DBD::Driver    PACKAGE = DBD::Driver::st
/* Non-standard sth XS methods following here, if any.       */
/* In particular this includes things like _list_fields from */
/* DBD::mSQL and DBD::mysql for accessing metadata.          */
```

Note especially the include of Driver.xsi here: DBI inserts stub functions for almost all private methods here which will typically do much work for you.

Wherever you really have to implement something, it will call a private function in dbdimp.c, and this is what you have to implement.

You need to set up an extra routine if your driver needs to export constants of its own, analogous to the SQL types available when you say:

```
use DBI qw(:sql_types);
```

FIX ME T.B.S

Driver.h

Driver.h is very simple and the operational contents should look like this:

```
#ifndef DRIVER_H_INCLUDED
#define DRIVER_H_INCLUDED
#define NEED_DBIXS_VERSION 93    /* 93 for DBI versions 1.00 to 1.51+ */
#define PERL_NO_GET_CONTEXT      /* if used require DBI 1.51+ */
#include <DBIXS.h>      /* installed by the DBI module  */
#include "dbdimp.h"
#include "dbivport.h"   /* see below                    */
#include <dbd_xsh.h>    /* installed by the DBI module  */
#endif /* DRIVER_H_INCLUDED */
```

The DBIXS.h header defines most of the interesting information that the writer of a driver needs.

The file dbd_xsh.h header provides prototype declarations for the C functions that you might decide to implement. Note that you should normally only define one of "dbd_db_login()", "dbd_db_login6()" or "dbd_db_login6_sv" unless you are intent on supporting really old versions of DBI (prior to DBI 1.06) as well as modern versions. The only standard, DBI-mandated functions that you need write are those specified in the dbd_xsh.h header. You might also add extra driver-specific functions in Driver.xs.

The dbivport.h file should be copied from the latest DBI release into your distribution each time you modify your driver. Its job is to allow you to enhance your code to work with the latest DBI API while still allowing your driver to be compiled and used with older versions of the DBI (for example, when the "DBIh_SET_ERR_CHAR()" macro was added to DBI 1.41, an emulation of it was added to dbivport.h). This makes users happy and your life easier. Always read the notes in dbivport.h to check for any limitations in the emulation that you should be aware of.

With DBI v1.51 or better I recommend that the driver defines PERL_NO_GET_CONTEXT before DBIXS.h is included. This can significantly improve efficiency when running under a thread enabled perl. (Remember that the standard perl in most Linux distributions is built with

threads enabled. So is ActiveState perl for Windows, and perl built for Apache mod_perl2.) If you do this there are some things to keep in mind:

- If PERL_NO_GET_CONTEXT is defined, then every function that calls the Perl API will need to start out with a "dTHX;" declaration.
- You'll know which functions need this, because the C compiler will complain that the undeclared identifier "my_perl" is used if and only if the perl you are using to develop and test your driver has threads enabled.
- If you don't remember to test with a thread-enabled perl before making a release it's likely that you'll get failure reports from users who are.
- For driver private functions it is possible to gain even more efficiency by replacing "dTHX;" with "pTHX_" prepended to the parameter list and then "aTHX_" prepended to the argument list where the function is called.

See "How multiple interpreters and concurrency are supported" in perlguts for additional information about PERL_NO_GET_CONTEXT.

Implementation header dbdimp.h

This header file has two jobs:

First it defines data structures for your private part of the handles. Note that the DBI provides many common fields for you. For example the statement handle (imp_sth) already has a row_count field with an IV type that accessed via the DBIc_ROW_COUNT(imp_sth) macro. Using this is strongly recommended as it's built in to some DBI internals so the DBI can 'just work' in more cases and you'll have less driver-specific code to write. Study DBIXS.h to see what's included with each type of handle.

Second it defines macros that rename the generic names like "dbd_db_login()" to database specific names like "ora_db_login()". This avoids name clashes and enables use of different drivers when you work with a statically linked perl.

It also will have the important task of disabling XS methods that you don't want to implement.

Finally, the macros will also be used to select alternate implementations of some functions. For example, the "dbd_db_login()" function is not passed the attribute hash. Since DBI v1.06, if a "dbd_db_login6()" macro is defined (for a function with 6 arguments), it will be used instead with the attribute hash passed as the sixth argument. Since DBI post v1.607, if a "dbd_db_login6_sv()" macro is defined (for a function like dbd_db_login6 but with scalar pointers for the dbname, username and password), it will be

used instead. This will allow your login6 function to see if there are any Unicode characters in the dbname.

Similarly defining dbd_db_do4_iv is preferred over dbd_db_do4, dbd_st_rows_iv over dbd_st_rows, and dbd_st_execute_iv over dbd_st_execute. The *_iv forms are declared to return the IV type instead of an int.

People used to just pick Oracle's dbdimp.c and use the same names, structures and types. I strongly recommend against that. At first glance this saves time, but your implementation will be less readable. It was just hell when I had to separate DBI specific parts, Oracle specific parts, mSQL specific parts and mysql specific parts in DBD::mysql's dbdimp.h and dbdimp.c. (DBD::mysql was a port of DBD::mSQL which was based on DBD::Oracle.) [Seconded, based on the experience taking DBD::Informix apart, even though the version inherited in 1996 was only based on DBD::Oracle.]

This part of the driver is your exclusive part. Rewrite it from scratch, so it will be clean and short: in other words, a better piece of code. (Of course keep an eye on other people's work.)

```
 struct imp_drh_st {
    dbih_drc_t com;         /* MUST be first element in structure   */
    /* Insert your driver handle attributes here */
 };
 struct imp_dbh_st {
    dbih_dbc_t com;         /* MUST be first element in structure   */
    /* Insert your database handle attributes here */
 };
 struct imp_sth_st {
    dbih_stc_t com;         /* MUST be first element in structure   */
    /* Insert your statement handle attributes here */
 };
 /*  Rename functions for avoiding name clashes; prototypes are  */
 /*  in dbd_xsh.h                                        */
 #define dbd_init           drv_dr_init
 #define dbd_db_login6_sv    drv_db_login_sv
 #define dbd_db_do          drv_db_do
 ... many more here ...
```

These structures implement your private part of the handles.

You have to use the name "imp_dbh_{dr|db|st}" and the first field must be of type dbih_drc_t|_dbc_t|_stc_t and must be called "com".

You should never access these fields directly, except by using the DBIc_xxx() macros below.

Implementation source dbdimp.c

Conventionally, dbdimp.c is the main implementation file (but DBD::Informix calls the file dbdimp.ec). This section includes a short note on each function that is used in the Driver.xsi template and thus has to be implemented.

Of course, you will probably also need to implement other support functions, which should usually be file static if they are placed in dbdimp.c. If they are placed in other files, you need to list those files in Makefile.PL (and MANIFEST) to handle them correctly.

It is wise to adhere to a namespace convention for your functions to avoid conflicts. For example, for a driver with prefix drv_, you might call externally visible functions dbd_drv_xxxx. You should also avoid non-constant global variables as much as possible to improve the support for threading.

Since Perl requires support for function prototypes (ANSI or ISO or Standard C), you should write your code using function prototypes too.

It is possible to use either the unmapped names such as "dbd_init()" or the mapped names such as "dbd_ix_dr_init()" in the dbdimp.c file. DBD::Informix uses the mapped names which makes it easier to identify where to look for linkage problems at runtime (which will report errors using the mapped names).

Most other drivers, and in particular DBD::Oracle, use the unmapped names in the source code which makes it a little easier to compare code between drivers and eases discussions on the dbi-dev mailing list. The majority of the code fragments here will use the unmapped names.

Ultimately, you should provide implementations for most of the functions listed in the dbd_xsh.h header. The exceptions are optional functions (such as "dbd_st_rows()") and those functions with alternative signatures, such as "dbd_db_login6_sv", "dbd_db_login6()" and dbd_db_login(). Then you should only implement one of the alternatives, and generally the newer one of the alternatives.

The dbd_init method

```
  #include "Driver.h"
```

```
  DBISTATE_DECLARE;
  void dbd_init(dbistate_t* dbistate)
  {
      DBISTATE_INIT;  /*  Initialize the DBI macros  */
  }
```

The "dbd_init()" function will be called when your driver is first loaded; the bootstrap command in "DBD::Driver::dr::driver()" triggers this, and the call is generated in the BOOT section of Driver.xst. These statements are needed to allow your driver to use the DBI macros. They will include your private header file dbdimp.h in turn. Note that DBISTATE_INIT requires the name of the argument to "dbd_init()" to be called "dbistate()".

The dbd_drv_error method

You need a function to record errors so DBI can access them properly. You can call it whatever you like, but we'll call it "dbd_drv_error()" here.

The argument list depends on your database software; different systems provide different ways to get at error information.

```
  static void dbd_drv_error(SV *h, int rc, const char *what)
  {
```

Note that h is a generic handle, may it be a driver handle, a database or a statement handle.

```
      D_imp_xxh(h);
```

This macro will declare and initialize a variable imp_xxh with a pointer to your private handle pointer. You may cast this to to imp_drh_t, imp_dbh_t or imp_sth_t.

To record the error correctly, equivalent to the "set_err()" method, use one of the "DBIh_SET_ERR_CHAR(...)" or "DBIh_SET_ERR_SV(...)" macros, which were added in DBI 1.41:

```
  DBIh_SET_ERR_SV(h, imp_xxh, err, errstr, state, method);
  DBIh_SET_ERR_CHAR(h, imp_xxh, err_c, err_i, errstr, state, method);
```

For "DBIh_SET_ERR_SV" the err, errstr, state, and method parameters are "SV*" (use &sv_undef instead of NULL).

For "DBIh_SET_ERR_CHAR" the err_c, errstr, state, method parameters are "char*".

The err_i parameter is an "IV" that's used instead of err_c if err_c is "Null".

The method parameter can be ignored.

The "DBIh_SET_ERR_CHAR" macro is usually the simplest to use when you just have an integer error code and an error message string:

```
  DBIh_SET_ERR_CHAR(h, imp_xxh, Nullch, rc, what, Nullch, Nullch);
```

As you can see, any parameters that aren't relevant to you can be "Null".

To make drivers compatible with DBI < 1.41 you should be using dbivport.h as described in "Driver.h" above.

The (obsolete) macros such as "DBIh_EVENT2" should be removed from drivers.

The names "dbis" and "DBIS", which were used in previous versions of this document, should be replaced with the "DBIc_DBISTATE(imp_xxh)" macro.

The name "DBILOGFP", which was also used in previous versions of this document, should be replaced by "DBIc_LOGPIO(imp_xxh)".

Your code should not call the C "<stdio.h>" I/O functions; you should use "PerlIO_printf()" as shown:

```
    if (DBIc_TRACE_LEVEL(imp_xxh) >= 2)
        PerlIO_printf(DBIc_LOGPIO(imp_xxh), "foobar %s: %s\n",
            foo, neatsvpv(errstr,0));
```

That's the first time we see how tracing works within a DBI driver. Make use of this as often as you can, but don't output anything at a trace level less than 3. Levels 1 and 2 are reserved for the DBI.

You can define up to 8 private trace flags using the top 8 bits of "DBIc_TRACE_FLAGS(imp)", that is: 0xFF000000. See the "parse_trace_flag()" method elsewhere in this document.

The dbd_dr_data_sources method

This method is optional; the support for it was added in DBI v1.33.

As noted in the discussion of Driver.pm, if the data sources can be determined by pure Perl code, do it that way. If, as in DBD::Informix, the information is obtained by a C function call, then you need to define a function that matches the prototype:

```
  extern AV *dbd_dr_data_sources(SV *drh, imp_drh_t *imp_drh, SV *attrs);
```

An outline implementation for DBD::Informix follows, assuming that the "sqgetdbs()" function call shown will return up to 100 databases names, with the pointers to each name in the array dbsname and the name strings themselves being stores in dbsarea.

```
  AV *dbd_dr_data_sources(SV *drh, imp_drh_t *imp_drh, SV *attr)
  {
      int ndbs;
      int i;
```

```
    char *dbsname[100];
    char  dbsarea[10000];
    AV *av = Nullav;
    if (sqgetdbs(&ndbs, dbsname, 100, dbsarea, sizeof(dbsarea)) == 0)
    {
        av = NewAV();
        av_extend(av, (I32)ndbs);
        sv_2mortal((SV *)av);
        for (i = 0; i < ndbs; i++)
          av_store(av, i, newSVpvf("dbi:Informix:%s", dbsname[i]));
    }
    return(av);
}
```

The actual DBD::Informix implementation has a number of extra lines of code, logs function entry and exit, reports the error from "sqgetdbs()", and uses "#define"'d constants for the array sizes.

The dbd_db_login6 method

```
int dbd_db_login6_sv(SV* dbh, imp_dbh_t* imp_dbh, SV* dbname,
                     SV* user, SV* auth, SV *attr);
```

or

```
int dbd_db_login6(SV* dbh, imp_dbh_t* imp_dbh, char* dbname,
                  char* user, char* auth, SV *attr);
```

This function will really connect to the database. The argument dbh is the database handle. imp_dbh is the pointer to the handles private data, as is imp_xxx in "dbd_drv_error()" above. The arguments dbname, user, auth and attr correspond to the arguments of the driver handle's "connect()" method.

You will quite often use database specific attributes here, that are specified in the DSN. I recommend you parse the DSN (using Perl) within the "connect()" method and pass the segments of the DSN via the attributes parameter through "_login()" to "dbd_db_login6()". Here's how you fetch them; as an example we use hostname attribute, which can be up to 12 characters long excluding null terminator:

```
SV** svp;
STRLEN len;
```

```
  char* hostname;
  if ( (svp = DBD_ATTRIB_GET_SVP(attr, "drv_hostname", 12)) && SvTRUE(*svp)) {
      hostname = SvPV(*svp, len);
      DBD_ATTRIB_DELETE(attr, "drv_hostname", 12); /* avoid later STORE */
  } else {
      hostname = "localhost";
  }
```

If you handle any driver specific attributes in the dbd_db_login6 method you probably want to delete them from "attr" (as above with DBD_ATTRIB_DELETE). If you don't delete your handled attributes DBI will call "STORE" for each attribute after the connect/login and this is at best redundant for attributes you have already processed.

Note: Until revision 11605 (post DBI 1.607), there was a problem with DBD_ATTRIBUTE_DELETE so unless you require a DBI version after 1.607 you need to replace each DBD_ATTRIBUTE_DELETE call with:

```
  hv_delete((HV*)SvRV(attr), key, key_len, G_DISCARD)
```

Note that you can also obtain standard attributes such as AutoCommit and ChopBlanks from the attributes parameter, using "DBD_ATTRIB_GET_IV" for integer attributes.

If, for example, your database does not support transactions but AutoCommit is set off (requesting transaction support), then you can emulate a 'failure to connect'.

Now you should really connect to the database. In general, if the connection fails, it is best to ensure that all allocated resources are released so that the handle does not need to be destroyed separately. If you are successful (and possibly even if you fail but you have allocated some resources), you should use the following macros:

```
  DBIc_IMPSET_on(imp_dbh);
```

This indicates that the driver (implementor) has allocated resources in the imp_dbh structure and that the implementors private "dbd_db_destroy()" function should be called when the handle is destroyed.

```
  DBIc_ACTIVE_on(imp_dbh);
```

This indicates that the handle has an active connection to the server and that the "dbd_db_disconnect()" function should be called before the handle is destroyed.

Note that if you do need to fail, you should report errors via the drh or imp_drh rather than via dbh or imp_dbh because imp_dbh will be destroyed by the failure, so errors recorded in that handle will not be visible to DBI, and hence not the user either.

Note too, that the function is passed dbh and imp_dbh, and there is a macro "D_imp_drh_from_dbh" which can recover the imp_drh from the imp_dbh. However, there is no DBI macro to provide you with the drh given either the imp_dbh or the dbh or the imp_drh (and there's no way to recover the dbh given just the imp_dbh).

This suggests that, despite the above notes about "dbd_drv_error()" taking an "SV *", it may be better to have two error routines, one taking imp_dbh and one taking imp_drh instead. With care, you can factor most of the formatting code out so that these are small routines calling a common error formatter. See the code in DBD::Informix 1.05.00 for more information.

The "dbd_db_login6()" function should return TRUE for success, FALSE otherwise.

Drivers implemented long ago may define the five-argument function "dbd_db_login()" instead of "dbd_db_login6()". The missing argument is the attributes. There are ways to work around the missing attributes, but they are ungainly; it is much better to use the 6-argument form. Even later drivers will use "dbd_db_login6_sv()" which provides the dbname, username and password as SVs.

The dbd_db_commit and dbd_db_rollback methods

```
  int dbd_db_commit(SV *dbh, imp_dbh_t *imp_dbh);
  int dbd_db_rollback(SV* dbh, imp_dbh_t* imp_dbh);
```

These are used for commit and rollback. They should return TRUE for success, FALSE for error.

The arguments dbh and imp_dbh are the same as for "dbd_db_login6()" above; I will omit describing them in what follows, as they appear always.

These functions should return TRUE for success, FALSE otherwise.

The dbd_db_disconnect method

This is your private part of the "disconnect()" method. Any dbh with the ACTIVE flag on must be disconnected. (Note that you have to set it in "dbd_db_connect()" above.)

```
  int dbd_db_disconnect(SV* dbh, imp_dbh_t* imp_dbh);
```

The database handle will return TRUE for success, FALSE otherwise. In any case it should do a:

```
  DBIc_ACTIVE_off(imp_dbh);
```

before returning so DBI knows that "dbd_db_disconnect()" was executed.

Note that there's nothing to stop a dbh being disconnected while it still have active children. If your database API reacts badly to trying to use an sth in this situation then

you'll need to add code like this to all sth methods:

```
  if (!DBIc_ACTIVE(DBIc_PARENT_COM(imp_sth)))
    return 0;
```

Alternatively, you can add code to your driver to keep explicit track of the statement handles that exist for each database handle and arrange to destroy those handles before disconnecting from the database. There is code to do this in DBD::Informix. Similar comments apply to the driver handle keeping track of all the database handles.

Note that the code which destroys the subordinate handles should only release the associated database resources and mark the handles inactive; it does not attempt to free the actual handle structures.

This function should return TRUE for success, FALSE otherwise, but it is not clear what anything can do about a failure.

The dbd_db_discon_all method

```
  int dbd_discon_all (SV *drh, imp_drh_t *imp_drh);
```

This function may be called at shutdown time. It should make best-efforts to disconnect all database handles - if possible. Some databases don't support that, in which case you can do nothing but return 'success'.

This function should return TRUE for success, FALSE otherwise, but it is not clear what anything can do about a failure.

The dbd_db_destroy method

This is your private part of the database handle destructor. Any dbh with the IMPSET flag on must be destroyed, so that you can safely free resources. (Note that you have to set it in "dbd_db_connect()" above.)

```
  void dbd_db_destroy(SV* dbh, imp_dbh_t* imp_dbh)
  {
      DBIc_IMPSET_off(imp_dbh);
  }
```

The DBI Driver.xst code will have called "dbd_db_disconnect()" for you, if the handle is still 'active', before calling "dbd_db_destroy()".

Before returning the function must switch IMPSET to off, so DBI knows that the destructor was called.

A DBI handle doesn't keep references to its children. But children do keep references to their parents. So a database handle won't be "DESTROY"'d until all its children have been

"DESTROY"'d.

The dbd_db_STORE_attrib method

This function handles

```
  $dbh->{$key} = $value;
```

Its prototype is:

```
  int dbd_db_STORE_attrib(SV* dbh, imp_dbh_t* imp_dbh, SV* keysv,
                          SV* valuesv);
```

You do not handle all attributes; on the contrary, you should not handle DBI attributes here: leave this to DBI. (There are two exceptions, AutoCommit and ChopBlanks, which you should care about.)

The return value is TRUE if you have handled the attribute or FALSE otherwise. If you are handling an attribute and something fails, you should call "dbd_drv_error()", so DBI can raise exceptions, if desired. If "dbd_drv_error()" returns, however, you have a problem: the user will never know about the error, because he typically will not check "$dbh->errstr()".

I cannot recommend a general way of going on, if "dbd_drv_error()" returns, but there are examples where even the DBI specification expects that you "croak()". (See the AutoCommit method in DBI.)

If you have to store attributes, you should either use your private data structure imp_xxx, the handle hash (via "(HV*)SvRV(dbh)"), or use the private imp_data.

The first is best for internal C values like integers or pointers and where speed is important within the driver. The handle hash is best for values the user may want to get/set via driver-specific attributes. The private imp_data is an additional "SV" attached to the handle. You could think of it as an unnamed handle attribute. It's not normally used.

The dbd_db_FETCH_attrib method

This is the counterpart of "dbd_db_STORE_attrib()", needed for:

```
  $value = $dbh->{$key};
```

Its prototype is:

```
  SV* dbd_db_FETCH_attrib(SV* dbh, imp_dbh_t* imp_dbh, SV* keysv);
```

Unlike all previous methods this returns an "SV" with the value. Note that you should normally execute "sv_2mortal()", if you return a nonconstant value. (Constant values are &sv_undef, &sv_no and &sv_yes.)

Note, that DBI implements a caching algorithm for attribute values. If you think, that an attribute may be fetched, you store it in the dbh itself:

```
  if (cacheit) /* cache value for later DBI 'quick' fetch? */
      hv_store((HV*)SvRV(dbh), key, kl, cachesv, 0);
```

The dbd_st_prepare method

This is the private part of the "prepare()" method. Note that you must not really execute the statement here. You may, however, preparse and validate the statement, or do similar things.

```
  int dbd_st_prepare(SV* sth, imp_sth_t* imp_sth, char* statement,
                     SV* attribs);
```

A typical, simple, possibility is to do nothing and rely on the perl "prepare()" code that set the Statement attribute on the handle. This attribute can then be used by "dbd_st_execute()".

If the driver supports placeholders then the NUM_OF_PARAMS attribute must be set correctly by "dbd_st_prepare()":

```
  DBIc_NUM_PARAMS(imp_sth) = ...
```

If you can, you should also setup attributes like NUM_OF_FIELDS, NAME, etc. here, but DBI doesn't require that - they can be deferred until execute() is called. However, if you do, document it.

In any case you should set the IMPSET flag, as you did in "dbd_db_connect()" above:

```
  DBIc_IMPSET_on(imp_sth);
```

The dbd_st_execute method

This is where a statement will really be executed.

```
  int dbd_st_execute(SV* sth, imp_sth_t* imp_sth);
```

"dbd_st_execute" should return -2 for any error, -1 if the number of rows affected is unknown else it should be the number of affected (updated, inserted) rows.

Note that you must be aware a statement may be executed repeatedly. Also, you should not expect that "finish()" will be called between two executions, so you might need code, like the following, near the start of the function:

```
  if (DBIc_ACTIVE(imp_sth))
      dbd_st_finish(h, imp_sth);
```

If your driver supports the binding of parameters (it should!), but the database doesn't, you must do it here. This can be done as follows:

```
  SV *svp;
  char* statement = DBD_ATTRIB_GET_PV(h, "Statement", 9, svp, "");
  int numParam = DBIc_NUM_PARAMS(imp_sth);
  int i;
  for (i = 0; i < numParam; i++)
  {
      char* value = dbd_db_get_param(sth, imp_sth, i);
      /* It is your drivers task to implement dbd_db_get_param,    */
      /* it must be setup as a counterpart of dbd_bind_ph.         */
      /* Look for '?' and replace it with 'value'.  Difficult      */
      /* task, note that you may have question marks inside         */
      /* quotes and comments the like ...  :-(                     */
      /* See DBD::mysql for an example. (Don't look too deep into  */
      /* the example, you will notice where I was lazy ...)        */
  }
```

The next thing is to really execute the statement.

Note that you must set the attributes NUM_OF_FIELDS, NAME, etc when the statement is successfully executed if the driver has not already done so: they may be used even before a potential "fetchrow()". In particular you have to tell DBI the number of fields that the statement has, because it will be used by DBI internally. Thus the function will typically ends with:

```
  if (isSelectStatement) {
      DBIc_NUM_FIELDS(imp_sth) = numFields;
      DBIc_ACTIVE_on(imp_sth);
  }
```

It is important that the ACTIVE flag only be set for "SELECT" statements (or any other statements that can return many values from the database using a cursor-like mechanism). See "dbd_db_connect()" above for more explanations.

There plans for a preparse function to be provided by DBI, but this has not reached fruition yet. Meantime, if you want to know how ugly it can get, try looking at the "dbd_ix_preparse()" in DBD::Informix dbdimp.ec and the related functions in iustoken.c and sqltoken.c.

The dbd_st_fetch method

This function fetches a row of data. The row is stored in in an array, of "SV"'s that DBI prepares for you. This has two advantages: it is fast (you even reuse the "SV"'s, so they don't have to be created after the first "fetchrow()"), and it guarantees that DBI handles "bind_cols()" for you.

What you do is the following:

```
  AV* av;
  int numFields = DBIc_NUM_FIELDS(imp_sth); /* Correct, if NUM_FIELDS
     is constant for this statement. There are drivers where this is
     not the case! */
  int chopBlanks = DBIc_is(imp_sth, DBIcf_ChopBlanks);
  int i;
  if (!fetch_new_row_of_data(...)) {
     ... /* check for error or end-of-data */
     DBIc_ACTIVE_off(imp_sth); /* turn off Active flag automatically */
     return Nullav;
  }
  /* get the fbav (field buffer array value) for this row       */
  /* it is very important to only call this after you know      */
  /* that you have a row of data to return.                     */
  av = DBIc_DBISTATE(imp_sth)->get_fbav(imp_sth);
  for (i = 0; i < numFields; i++) {
     SV* sv = fetch_a_field(..., i);
     if (chopBlanks && SvOK(sv) && type_is_blank_padded(field_type[i])) {
        /*  Remove white space from end (only) of sv  */
     }
     sv_setsv(AvARRAY(av)[i], sv); /* Note: (re)use! */
  }
  return av;
```

There's no need to use a "fetch_a_field()" function returning an "SV*". It's more common to use your database API functions to fetch the data as character strings and use code like this:

```
  sv_setpvn(AvARRAY(av)[i], char_ptr, char_count);
```

"NULL" values must be returned as "undef". You can use code like this:

```
  SvOK_off(AvARRAY(av)[i]);
```

The function returns the "AV" prepared by DBI for success or "Nullav" otherwise.

FIX ME Discuss what happens when there's no more data to fetch.

Are errors permitted if another fetch occurs after the first fetch

that reports no more data. (Permitted, not required.)

If an error occurs which leaves the $sth in a state where remaining rows can't be fetched then Active should be turned off before the method returns.

The dbd_st_finish3 method

The "$sth->finish()" method can be called if the user wishes to indicate that no more rows will be fetched even if the database has more rows to offer, and the DBI code can call the function when handles are being destroyed. See the DBI specification for more background details.

In both circumstances, the DBI code ends up calling the "dbd_st_finish3()" method (if you provide a mapping for "dbd_st_finish3()" in dbdimp.h), or "dbd_st_finish()" otherwise.

The difference is that "dbd_st_finish3()" takes a third argument which is an "int" with the value 1 if it is being called from a "destroy()" method and 0 otherwise.

Note that DBI v1.32 and earlier test on "dbd_db_finish3()" to call "dbd_st_finish3()"; if you provide "dbd_st_finish3()", either define "dbd_db_finish3()" too, or insist on DBI v1.33 or later.

All it needs to do is turn off the Active flag for the sth. It will only be called by Driver.xst code, if the driver has set ACTIVE to on for the sth.

Outline example:

```
  int dbd_st_finish3(SV* sth, imp_sth_t* imp_sth, int from_destroy) {
      if (DBIc_ACTIVE(imp_sth))
      {
          /* close cursor or equivalent action */
          DBIc_ACTIVE_off(imp_sth);
      }
      return 1;
  }
```

The from_destroy parameter is true if "dbd_st_finish3()" is being called from "DESTROY()" - and so the statement is about to be destroyed. For many drivers there is no point in doing anything more than turning off the Active flag in this case.

The function returns TRUE for success, FALSE otherwise, but there isn't a lot anyone can do to recover if there is an error.

The dbd_st_destroy method

This function is the private part of the statement handle destructor.

```
  void dbd_st_destroy(SV* sth, imp_sth_t* imp_sth) {
      ... /* any clean-up that's needed */
      DBIc_IMPSET_off(imp_sth); /* let DBI know we've done it   */
  }
```

The DBI Driver.xst code will call "dbd_st_finish()" for you, if the sth has the ACTIVE flag set, before calling "dbd_st_destroy()".

The dbd_st_STORE_attrib and dbd_st_FETCH_attrib methods

These functions correspond to "dbd_db_STORE()" and "dbd_db_FETCH()" attrib above, except that they are for statement handles. See above.

```
  int dbd_st_STORE_attrib(SV* sth, imp_sth_t* imp_sth, SV* keysv,
                          SV* valuesv);
  SV* dbd_st_FETCH_attrib(SV* sth, imp_sth_t* imp_sth, SV* keysv);
```

The dbd_bind_ph method

This function is internally used by the "bind_param()" method, the "bind_param_inout()" method and by the DBI Driver.xst code if "execute()" is called with any bind parameters.

```
  int dbd_bind_ph (SV *sth, imp_sth_t *imp_sth, SV *param,
                   SV *value, IV sql_type, SV *attribs,
                   int is_inout, IV maxlen);
```

The param argument holds an "IV" with the parameter number (1, 2, ...). The value argument is the parameter value and sql_type is its type.

If your driver does not support "bind_param_inout()" then you should ignore maxlen and croak if is_inout is TRUE.

If your driver does support "bind_param_inout()" then you should note that value is the "SV" after dereferencing the reference passed to "bind_param_inout()".

In drivers of simple databases the function will, for example, store the value in a parameter array and use it later in "dbd_st_execute()". See the DBD::mysql driver for an example.

Implementing bind_param_inout support

To provide support for parameters bound by reference rather than by value, the driver must

do a number of things. First, and most importantly, it must note the references and stash them in its own driver structure. Secondly, when a value is bound to a column, the driver must discard any previous reference bound to the column. On each execute, the driver must evaluate the references and internally bind the values resulting from the references.

This is only applicable if the user writes:

```
$sth->execute;
```

If the user writes:

```
$sth->execute(@values);
```

then DBI automatically calls the binding code for each element of @values. These calls are indistinguishable from explicit user calls to "bind_param()".

C/XS version of Makefile.PL

The Makefile.PL file for a C/XS driver is similar to the code needed for a pure Perl driver, but there are a number of extra bits of information needed by the build system.

For example, the attributes list passed to "WriteMakefile()" needs to specify the object files that need to be compiled and built into the shared object (DLL). This is often, but not necessarily, just dbdimp.o (unless that should be dbdimp.obj because you're building on MS Windows).

Note that you can reliably determine the extension of the object files from the $Config{obj_ext} values, and there are many other useful pieces of configuration information lurking in that hash. You get access to it with:

```
use Config;
```

Methods which do not need to be written

The DBI code implements the majority of the methods which are accessed using the notation "DBI->function()", the only exceptions being "DBI->connect()" and "DBI->data_sources()" which require support from the driver.

The DBI code implements the following documented driver, database and statement functions which do not need to be written by the DBD driver writer.

$dbh->do()

The default implementation of this function prepares, executes and destroys the statement. This can be replaced if there is a better way to implement this, such as "EXECUTE IMMEDIATE" which can sometimes be used if there are no parameters.

$h->errstr()

$h->err()

$h->state()

$h->trace()

The DBD driver does not need to worry about these routines at all.

$h->{ChopBlanks}

This attribute needs to be honored during "fetch()" operations, but does not need to be handled by the attribute handling code.

$h->{RaiseError}

The DBD driver does not need to worry about this attribute at all.

$h->{PrintError}

The DBD driver does not need to worry about this attribute at all.

$sth->bind_col()

Assuming the driver uses the "DBIc_DBISTATE(imp_xxh)->get_fbav()" function (C drivers, see below), or the "$sth->_set_fbav($data)" method (Perl drivers) the driver does not need to do anything about this routine.

$sth->bind_columns()

Regardless of whether the driver uses "DBIc_DBISTATE(imp_xxh)->get_fbav()", the driver does not need to do anything about this routine as it simply iteratively calls "$sth->bind_col()".

The DBI code implements a default implementation of the following functions which do not need to be written by the DBD driver writer unless the default implementation is incorrect for the Driver.

$dbh->quote()

This should only be written if the database does not accept the ANSI SQL standard for quoting strings, with the string enclosed in single quotes and any embedded single quotes replaced by two consecutive single quotes.

For the two argument form of quote, you need to implement the "type_info()" method to provide the information that quote needs.

$dbh->ping()

This should be implemented as a simple efficient way to determine whether the connection to the database is still alive. Typically code like this:

```
sub ping {
    my $dbh = shift;
    $sth = $dbh->prepare_cached(q{
```

```
            select * from A_TABLE_NAME where 1=0
        }) or return 0;
        $sth->execute or return 0;
        $sth->finish;
        return 1;
    }
```

where A_TABLE_NAME is the name of a table that always exists (such as a database system catalogue).

$drh->default_user

The default implementation of default_user will get the database username and password fields from $ENV{DBI_USER} and $ENV{DBI_PASS}. You can override this method. It is called as follows:

```
($user, $pass) = $drh->default_user($user, $pass, $attr)
```

METADATA METHODS

The exposition above ignores the DBI MetaData methods. The metadata methods are all associated with a database handle.

Using DBI::DBD::Metadata

The DBI::DBD::Metadata module is a good semi-automatic way for the developer of a DBD module to write the "get_info()" and "type_info()" functions quickly and accurately.

Generating the get_info method

Prior to DBI v1.33, this existed as the method "write_getinfo_pm()" in the DBI::DBD module. From DBI v1.33, it exists as the method "write_getinfo_pm()" in the DBI::DBD::Metadata module. This discussion assumes you have DBI v1.33 or later.

You examine the documentation for "write_getinfo_pm()" using:

```
perldoc DBI::DBD::Metadata
```

To use it, you need a Perl DBI driver for your database which implements the "get_info()" method. In practice, this means you need to install DBD::ODBC, an ODBC driver manager, and an ODBC driver for your database.

With the pre-requisites in place, you might type:

```
perl -MDBI::DBD::Metadata -we \
    "write_getinfo_pm (qw{ dbi:ODBC:foo_db username password Driver })"
```

The procedure writes to standard output the code that should be added to your Driver.pm file and the code that should be written to lib/DBD/Driver/GetInfo.pm.

You should review the output to ensure that it is sensible.

Generating the type_info method

Given the idea of the "write_getinfo_pm()" method, it was not hard to devise a parallel method, "write_typeinfo_pm()", which does the analogous job for the DBI "type_info_all()" metadata method. The "write_typeinfo_pm()" method was added to DBI v1.33.

You examine the documentation for "write_typeinfo_pm()" using:

```
perldoc DBI::DBD::Metadata
```

The setup is exactly analogous to the mechanism described in "Generating the get_info method".

With the pre-requisites in place, you might type:

```
perl -MDBI::DBD::Metadata -we \
  "write_typeinfo_pm (qw{ dbi:ODBC:foo_db username password Driver })"
```

The procedure writes to standard output the code that should be added to your Driver.pm file and the code that should be written to lib/DBD/Driver/TypeInfo.pm.

You should review the output to ensure that it is sensible.

Writing DBD::Driver::db::get_info

If you use the DBI::DBD::Metadata module, then the code you need is generated for you.

If you decide not to use the DBI::DBD::Metadata module, you should probably borrow the code from a driver that has done so (eg DBD::Informix from version 1.05 onwards) and crib the code from there, or look at the code that generates that module and follow that. The method in Driver.pm will be very simple; the method in lib/DBD/Driver/GetInfo.pm is not very much more complex unless your DBMS itself is much more complex.

Note that some of the DBI utility methods rely on information from the "get_info()" method to perform their operations correctly. See, for example, the "quote_identifier()" and quote methods, discussed below.

Writing DBD::Driver::db::type_info_all

If you use the "DBI::DBD::Metadata" module, then the code you need is generated for you.

If you decide not to use the "DBI::DBD::Metadata" module, you should probably borrow the code from a driver that has done so (eg "DBD::Informix" from version 1.05 onwards) and crib the code from there, or look at the code that generates that module and follow that. The method in Driver.pm will be very simple; the method in lib/DBD/Driver/TypeInfo.pm is not very much more complex unless your DBMS itself is much more complex.

Writing DBD::Driver::db::type_info

The guidelines on writing this method are still not really clear. No sample implementation is available.

Writing DBD::Driver::db::table_info

FIX ME The guidelines on writing this method have not been written yet.

No sample implementation is available.

Writing DBD::Driver::db::column_info

FIX ME The guidelines on writing this method have not been written yet.

No sample implementation is available.

Writing DBD::Driver::db::primary_key_info

FIX ME The guidelines on writing this method have not been written yet.

No sample implementation is available.

Writing DBD::Driver::db::primary_key

FIX ME The guidelines on writing this method have not been written yet.

No sample implementation is available.

Writing DBD::Driver::db::foreign_key_info

FIX ME The guidelines on writing this method have not been written yet.

No sample implementation is available.

Writing DBD::Driver::db::tables

This method generates an array of names in a format suitable for being embedded in SQL statements in places where a table name is expected.

If your database hews close enough to the SQL standard or if you have implemented an appropriate "table_info()" function and and the appropriate "quote_identifier()" function, then the DBI default version of this method will work for your driver too.

Otherwise, you have to write a function yourself, such as:

```
sub tables
{
    my($dbh, $cat, $sch, $tab, $typ) = @_;
    my(@res);
    my($sth) = $dbh->table_info($cat, $sch, $tab, $typ);
    my(@arr);
    while (@arr = $sth->fetchrow_array)
    {
        push @res, $dbh->quote_identifier($arr[0], $arr[1], $arr[2]);
```

```
    }
    return @res;
}
```

See also the default implementation in DBI.pm.

Writing DBD::Driver::db::quote

This method takes a value and converts it into a string suitable for embedding in an SQL statement as a string literal.

If your DBMS accepts the SQL standard notation for strings (single quotes around the string as a whole with any embedded single quotes doubled up), then you do not need to write this method as DBI provides a default method that does it for you.

If your DBMS uses an alternative notation or escape mechanism, then you need to provide an equivalent function. For example, suppose your DBMS used C notation with double quotes around the string and backslashes escaping both double quotes and backslashes themselves. Then you might write the function as:

```
sub quote
{
    my($dbh, $str) = @_;
    $str =~ s/["\\]/\\$&/gmo;
    return qq{"$str"};
}
```

Handling newlines and other control characters is left as an exercise for the reader.

This sample method ignores the $data_type indicator which is the optional second argument to the method.

Writing DBD::Driver::db::quote_identifier

This method is called to ensure that the name of the given table (or other database object) can be embedded into an SQL statement without danger of misinterpretation. The result string should be usable in the text of an SQL statement as the identifier for a table.

If your DBMS accepts the SQL standard notation for quoted identifiers (which uses double quotes around the identifier as a whole, with any embedded double quotes doubled up) and accepts "schema"."identifier" (and "catalog"."schema"."identifier" when a catalog is specified), then you do not need to write this method as DBI provides a default method that does it for you.

In fact, even if your DBMS does not handle exactly that notation but you have implemented the "get_info()" method and it gives the correct responses, then it will work for you. If your database is fussier, then you need to implement your own version of the function. For example, DBD::Informix has to deal with an environment variable DELIMIDENT. If it is not set, then the DBMS treats names enclosed in double quotes as strings rather than names, which is usually a syntax error. Additionally, the catalog portion of the name is separated from the schema and table by a different delimiter (colon instead of dot), and the catalog portion is never enclosed in quotes. (Fortunately, valid strings for the catalog will never contain weird characters that might need to be escaped, unless you count dots, dashes, slashes and at-signs as weird.) Finally, an Informix database can contain objects that cannot be accessed because they were created by a user with the DELIMIDENT environment variable set, but the current user does not have it set. By design choice, the "quote_identifier()" method encloses those identifiers in double quotes anyway, which generally triggers a syntax error, and the metadata methods which generate lists of tables etc omit those identifiers from the result sets.

```
    sub quote_identifier
    {
        my($dbh, $cat, $sch, $obj) = @_;
        my($rv) = "";
        my($qq) = (defined $ENV{DELIMIDENT}) ? "" : '';
        $rv .= qq{$cat:} if (defined $cat);
        if (defined $sch)
        {
            if ($sch !~ m/^\w+$/o)
            {
                $qq = "";
                $sch =~ s/$qq/$qq$qq/gm;
            }
            $rv .= qq{$qq$sch$qq.};
        }
        if (defined $obj)
        {
            if ($obj !~ m/^\w+$/o)
```

```
            {
                $qq = "";
                $obj =~ s/$qq/$qq$qq/gm;
            }
            $rv .= qq{$qq$obj$qq};
        }
        return $rv;
    }
```

Handling newlines and other control characters is left as an exercise for the reader.

Note that there is an optional fourth parameter to this function which is a reference to a hash of attributes; this sample implementation ignores that.

This sample implementation also ignores the single-argument variant of the method.

TRACING

Tracing in DBI is controlled with a combination of a trace level and a set of flags which together are known as the trace settings. The trace settings are stored in a single integer and divided into levels and flags by a set of masks ("DBIc_TRACE_LEVEL_MASK" and "DBIc_TRACE_FLAGS_MASK").

Each handle has it's own trace settings and so does the DBI. When you call a method the DBI merges the handles settings into its own for the duration of the call: the trace flags of the handle are OR'd into the trace flags of the DBI, and if the handle has a higher trace level then the DBI trace level is raised to match it. The previous DBI trace settings are restored when the called method returns.

Trace Level

The trace level is the first 4 bits of the trace settings (masked by "DBIc_TRACE_FLAGS_MASK") and represents trace levels of 1 to 15. Do not output anything at trace levels less than 3 as they are reserved for DBI.

For advice on what to output at each level see "Trace Levels" in DBI.

To test for a trace level you can use the "DBIc_TRACE_LEVEL" macro like this:

```
 if (DBIc_TRACE_LEVEL(imp_xxh) >= 2) {
     PerlIO_printf(DBIc_LOGPIO(imp_xxh), "foobar");
 }
```

Also note the use of PerlIO_printf which you should always use for tracing and never the C "stdio.h" I/O functions.

Trace Flags

Trace flags are used to enable tracing of specific activities within the DBI and drivers.

The DBI defines some trace flags and drivers can define others. DBI trace flag names begin with a capital letter and driver specific names begin with a lowercase letter. For a list of DBI defined trace flags see "Trace Flags" in DBI.

If you want to use private trace flags you'll probably want to be able to set them by name. Drivers are expected to override the parse_trace_flag (note the singular) and check if $trace_flag_name is a driver specific trace flags and, if not, then call the DBIs default parse_trace_flag(). To do that you'll need to define a parse_trace_flag() method like this:

```
sub parse_trace_flag {
    my ($h, $name) = @_;
    return 0x01000000 if $name eq 'foo';
    return 0x02000000 if $name eq 'bar';
    return 0x04000000 if $name eq 'baz';
    return 0x08000000 if $name eq 'boo';
    return 0x10000000 if $name eq 'bop';
    return $h->SUPER::parse_trace_flag($name);
}
```

All private flag names must be lowercase, and all private flags must be in the top 8 of the 32 bits of "DBIc_TRACE_FLAGS(imp)" i.e., 0xFF000000.

If you've defined a parse_trace_flag() method in ::db you'll also want it in ::st, so just alias it in:

```
*parse_trace_flag = \&DBD::foo::db::parse_trace_flag;
```

You may want to act on the current 'SQL' trace flag that DBI defines to output SQL prepared/executed as DBI currently does not do SQL tracing.

Trace Macros

Access to the trace level and trace flags is via a set of macros.

DBIc_TRACE_SETTINGS(imp) returns the trace settings

DBIc_TRACE_LEVEL(imp) returns the trace level

DBIc_TRACE_FLAGS(imp) returns the trace flags

DBIc_TRACE(imp, flags, flaglevel, level)

e.g.,

DBIc_TRACE(imp, 0, 0, 4)

if level >= 4

DBIc_TRACE(imp, DBDtf_FOO, 2, 4)

if tracing DBDtf_FOO & level>=2 or level>=4

DBIc_TRACE(imp, DBDtf_FOO, 2, 0)

as above but never trace just due to level

WRITING AN EMULATION LAYER FOR AN OLD PERL INTERFACE

Study Oraperl.pm (supplied with DBD::Oracle) and Ingperl.pm (supplied with DBD::Ingres) and the corresponding dbdimp.c files for ideas.

Note that the emulation code sets "$dbh->{CompatMode} = 1;" for each connection so that the internals of the driver can implement behaviour compatible with the old interface when dealing with those handles.

Setting emulation perl variables

For example, ingperl has a $sql_rowcount variable. Rather than try to manually update this in Ingperl.pm it can be done faster in C code. In "dbd_init()":

```
sql_rowcount = perl_get_sv("Ingperl::sql_rowcount", GV_ADDMULTI);
```

In the relevant places do:

```
if (DBIc_COMPAT(imp_sth))     /* only do this for compatibility mode handles */
    sv_setiv(sql_rowcount, the_row_count);
```

OTHER MISCELLANEOUS INFORMATION

The imp_xyz_t types

Any handle has a corresponding C structure filled with private data. Some of this data is reserved for use by DBI (except for using the DBIc macros below), some is for you. See the description of the dbdimp.h file above for examples. Most functions in dbdimp.c are passed both the handle "xyz" and a pointer to "imp_xyz". In rare cases, however, you may use the following macros:

D_imp_dbh(dbh)

Given a function argument dbh, declare a variable imp_dbh and initialize it with a pointer to the handles private data. Note: This must be a part of the function header, because it declares a variable.

D_imp_sth(sth)

Likewise for statement handles.

D_imp_xxx(h)

Given any handle, declare a variable imp_xxx and initialize it with a pointer to the handles private data. It is safe, for example, to cast imp_xxx to "imp_dbh_t*", if "DBIc_TYPE(imp_xxx) == DBIt_DB". (You can also call "sv_derived_from(h, "DBI::db")", but that's much slower.)

D_imp_dbh_from_sth

Given a imp_sth, declare a variable imp_dbh and initialize it with a pointer to the parent database handle's implementors structure.

Using DBIc_IMPSET_on

The driver code which initializes a handle should use "DBIc_IMPSET_on()" as soon as its state is such that the cleanup code must be called. When this happens is determined by your driver code.

Failure to call this can lead to corruption of data structures.

For example, DBD::Informix maintains a linked list of database handles in the driver, and within each handle, a linked list of statements. Once a statement is added to the linked list, it is crucial that it is cleaned up (removed from the list). When DBIc_IMPSET_on() was being called too late, it was able to cause all sorts of problems.

Using DBIc_is(), DBIc_has(), DBIc_on() and DBIc_off()

Once upon a long time ago, the only way of handling the internal DBI boolean flags/attributes was through macros such as:

```
DBIc_WARN       DBIc_WARN_on        DBIc_WARN_off
DBIc_COMPAT     DBIc_COMPAT_on      DBIc_COMPAT_off
```

Each of these took an imp_xxh pointer as an argument.

Since then, new attributes have been added such as ChopBlanks, RaiseError and PrintError, and these do not have the full set of macros. The approved method for handling these is now the four macros:

```
DBIc_is(imp, flag)
DBIc_has(imp, flag)       an alias for DBIc_is
DBIc_on(imp, flag)
DBIc_off(imp, flag)
DBIc_set(imp, flag, on)   set if on is true, else clear
```

Consequently, the "DBIc_XXXXX" family of macros is now mostly deprecated and new drivers should avoid using them, even though the older drivers will probably continue to do so for quite a while yet. However...

There is an important exception to that. The ACTIVE and IMPSET flags should be set via the "DBIc_ACTIVE_on()" and "DBIc_IMPSET_on()" macros, and unset via the "DBIc_ACTIVE_off()" and "DBIc_IMPSET_off()" macros.

Using the get_fbav() method

THIS IS CRITICAL for C/XS drivers.

The "$sth->bind_col()" and "$sth->bind_columns()" documented in the DBI specification do not have to be implemented by the driver writer because DBI takes care of the details for you.

However, the key to ensuring that bound columns work is to call the function "DBIc_DBISTATE(imp_xxh)->get_fbav()" in the code which fetches a row of data.

This returns an "AV", and each element of the "AV" contains the "SV" which should be set to contain the returned data.

The pure Perl equivalent is the "$sth->_set_fbav($data)" method, as described in the part on pure Perl drivers.

Casting strings to Perl types based on a SQL type

DBI from 1.611 (and DBIXS_REVISION 13606) defines the sql_type_cast_svpv method which may be used to cast a string representation of a value to a more specific Perl type based on a SQL type. You should consider using this method when processing bound column data as it provides some support for the TYPE bind_col attribute which is rarely used in drivers.

```
int sql_type_cast_svpv(pTHX_ SV *sv, int sql_type, U32 flags, void *v)
```

"sv" is what you would like cast, "sql_type" is one of the DBI defined SQL types (e.g., "SQL_INTEGER") and "flags" is a bitmask as follows:

DBIstcf_STRICT

: If set this indicates you want an error state returned if the cast cannot be performed.

DBIstcf_DISCARD_STRING

: If set and the pv portion of the "sv" is cast then this will cause sv's pv to be freed up.

sql_type_cast_svpv returns the following states:

-2 sql_type is not handled - sv not changed

-1 sv is undef, sv not changed

0 sv could not be cast cleanly and DBIstcf_STRICT was specified

1 sv could not be case cleanly and DBIstcf_STRICT was not specified

2 sv was cast ok

The current implementation of sql_type_cast_svpv supports "SQL_INTEGER", "SQL_DOUBLE" and "SQL_NUMERIC". "SQL_INTEGER" uses sv_2iv and hence may set IV, UV or NV depending on the number. "SQL_DOUBLE" uses sv_2nv so may set NV and "SQL_NUMERIC" will set IV or UV or NV.

DBIstcf_STRICT should be implemented as the StrictlyTyped attribute and DBIstcf_DISCARD_STRING implemented as the DiscardString attribute to the bind_col method and both default to off.

See DBD::Oracle for an example of how this is used.

SUBCLASSING DBI DRIVERS

This is definitely an open subject. It can be done, as demonstrated by the DBD::File driver, but it is not as simple as one might think.

(Note that this topic is different from subclassing the DBI. For an example of that, see the t/subclass.t file supplied with the DBI.)

The main problem is that the dbh's and sth's that your "connect()" and "prepare()" methods return are not instances of your DBD::Driver::db or DBD::Driver::st packages, they are not even derived from it. Instead they are instances of the DBI::db or DBI::st classes or a derived subclass. Thus, if you write a method "mymethod()" and do a

```
$dbh->mymethod()
```

then the autoloader will search for that method in the package DBI::db. Of course you can instead to a

```
$dbh->func('mymethod')
```

and that will indeed work, even if "mymethod()" is inherited, but not without additional work. Setting @ISA is not sufficient.

Overwriting methods

The first problem is, that the "connect()" method has no idea of subclasses. For example, you cannot implement base class and subclass in the same file: The "install_driver()" method wants to do a

```
require DBD::Driver;
```

In particular, your subclass has to be a separate driver, from the view of DBI, and you cannot share driver handles.

Of course that's not much of a problem. You should even be able to inherit the base classes "connect()" method. But you cannot simply overwrite the method, unless you do something like this, quoted from DBD::CSV:

```
sub connect ($$;$$$) {
    my ($drh, $dbname, $user, $auth, $attr) = @_;
    my $this = $drh->DBD::File::dr::connect($dbname, $user, $auth, $attr);
    if (!exists($this->{csv_tables})) {
        $this->{csv_tables} = {};
    }
    $this;
}
```

Note that we cannot do a

```
 $drh->SUPER::connect($dbname, $user, $auth, $attr);
```

as we would usually do in a an OO environment, because $drh is an instance of DBI::dr. And note, that the "connect()" method of DBD::File is able to handle subclass attributes. See the description of Pure Perl drivers above.

It is essential that you always call superclass method in the above manner. However, that should do.

Attribute handling

Fortunately the DBI specifications allow a simple, but still performant way of handling attributes. The idea is based on the convention that any driver uses a prefix driver_ for its private methods. Thus it's always clear whether to pass attributes to the super class or not. For example, consider this "STORE()" method from the DBD::CSV class:

```
sub STORE {
    my ($dbh, $attr, $val) = @_;
    if ($attr !~ /^driver_/) {
        return $dbh->DBD::File::db::STORE($attr, $val);
    }
    if ($attr eq 'driver_foo') {
    ...
}
```

AUTHORS

Jonathan Leffler <jleffler@us.ibm.com> (previously <jleffler@informix.com>), Jochen Wiedmann <joe@ispsoft.de>, Steffen Goeldner <sgoeldner@cpan.org>, and Tim Bunce <dbi-users@perl.org>.