NAME

DBM_Filter -- Filter DBM keys/values

SYNOPSIS

```
use DBM_Filter ;
use SDBM_File; # or DB_File, GDBM_File, NDBM_File, or ODBM_File

$db = tie %hash, ...

$db->Filter_Push(Fetch => sub {...},
Store => sub {...});

$db->Filter_Push('my_filter1');
$db->Filter_Push('my_filter2', params...);

$db->Filter_Key_Push(...) ;
$db->Filter_Value_Push(...) ;

$db->Filter_Pop();
$db->Filtered();

package DBM_Filter::my_filter1;

sub Store { ... }
sub Fetch { ... }

1;

package DBM_Filter::my_filter2;

sub Filter
{
my @opts = @_;
...
return (
sub Store { ... },
sub Fetch { ... } );
}

1;
```

DESCRIPTION

This module provides an interface that allows filters to be applied to tied Hashes associated with DBM files. It builds on the DBM Filter hooks that are present in all the *DB*_File modules included with the standard Perl source distribution from version 5.6.1 onwards. In addition to the *DB*_File modules distributed with Perl, the BerkeleyDB module, available on CPAN, supports the DBM Filter hooks. See perldbmfilter(1) for more details on the DBM Filter hooks.

What is a DBM Filter?

A DBM Filter allows the keys and/or values in a tied hash to be modified by some user-defined code just before it is written to the DBM file and just after it is read back from the DBM file. For example, this snippet of code

```
$some_hash{"abc"} = 42;
```

could potentially trigger two filters, one for the writing of the key ‘‘abc’’ and another for writing the value 42. Similarly, this snippet

```
my ($key, $value) = each %some_hash
```

will trigger two filters, one for the reading of the key and one for the reading of the value.

Like the existing DBM Filter functionality, this module arranges for the `$_` variable to be populated with the key or value that a filter will check. This usually means that most DBM filters tend to be very short.

So what's new?

The main enhancements over the standard DBM Filter hooks are:

- A cleaner interface.
- The ability to easily apply multiple filters to a single DBM file.
- The ability to create ''canned'' filters. These allow commonly used filters to be packaged into a stand-alone module.

METHODS

This module will arrange for the following methods to be available via the object returned from the `tie` call.

`$db->`*Filter_Push()* / `$db->`*Filter_Key_Push()* / `$db->`*Filter_Value_Push()*

Add a filter to filter stack for the database, `$db`. The three formats vary only in whether they apply to the DBM key, the DBM value or both.

Filter_Push
: The filter is applied to *both* keys and values.

Filter_Key_Push
: The filter is applied to the key *only*.

Filter_Value_Push
: The filter is applied to the value *only*.

`$db->`*Filter_Pop()*

Removes the last filter that was applied to the DBM file associated with `$db`, if present.

`$db->`*Filtered()*

Returns TRUE if there are any filters applied to the DBM associated with `$db`. Otherwise returns FALSE.

Writing a Filter

Filters can be created in two main ways

Immediate Filters

An immediate filter allows you to specify the filter code to be used at the point where the filter is applied to a dbm. In this mode the Filter_*_Push methods expects to receive exactly two parameters.

```
my $db = tie %hash, 'SDBM_File', ...
$db->Filter_Push( Store => sub { },
Fetch => sub { });
```

The code reference associated with `Store` will be called before any key/value is written to the database and the code reference associated with `Fetch` will be called after any key/value is read from the database.

For example, here is a sample filter that adds a trailing NULL character to all strings before they are written to the DBM file, and removes the trailing NULL when they are read from the DBM file

```
my $db = tie %hash, 'SDBM_File', ...
$db->Filter_Push( Store => sub { $_ .= "\x00" ; },
Fetch => sub { s/\x00$// ; });
```

Points to note:

1. Both the Store and Fetch filters manipulate `$_`.

Canned Filters

Immediate filters are useful for one-off situations. For more generic problems it can be useful to package the filter up in its own module.

The usage is for a canned filter is:

```
$db->Filter_Push("name", params)
```

where

"name"
: is the name of the module to load. If the string specified does not contain the package separator characters "::", it is assumed to refer to the full module name "DBM_Filter::name" This means that the full names for canned filters, "null" and "utf8", included with this module are:

```
DBM_Filter::null
DBM_Filter::utf8
```

params
: any optional parameters that need to be sent to the filter. See the encode filter for an example of a module that uses parameters.

The module that implements the canned filter can take one of two forms. Here is a template for the first

```
package DBM_Filter::null ;

use strict;
use warnings;

sub Store
{
# store code here
}

sub Fetch
{
# fetch code here
}

1;
```

Notes:

1. The package name uses the `DBM_Filter::` prefix.
2. The module *must* have both a Store and a Fetch method. If only one is present, or neither are present, a fatal error will be thrown.

The second form allows the filter to hold state information using a closure, thus:

```
package DBM_Filter::encoding ;

use strict;
use warnings;

sub Filter
{
my @params = @_ ;
```

```
...
return {
Store => sub { $_ = $encoding->encode($_) },
Fetch => sub { $_ = $encoding->decode($_) }
} ;
}

1;
```

In this instance the ''Store'' and ''Fetch'' methods are encapsulated inside a ''Filter'' method.

Filters Included

A number of canned filers are provided with this module. They cover a number of the main areas that filters are needed when interfacing with DBM files. They also act as templates for your own filters.

The filter included are:

- utf8

  This module will ensure that all data written to the DBM will be encoded in UTF-8.

  This module needs the Encode module.

- encode

  Allows you to choose the character encoding will be store in the DBM file.

- compress

  This filter will compress all data before it is written to the database and uncompressed it on reading.

  This module needs Compress::Zlib.

- int32

  This module is used when interoperating with a C/C++ application that uses a C int as either the key and/or value in the DBM file.

- null

  This module ensures that all data written to the DBM file is null terminated. This is useful when you have a perl script that needs to interoperate with a DBM file that a C program also uses. A fairly common issue is for the C application to include the terminating null in a string when it writes to the DBM file. This filter will ensure that all data written to the DBM file can be read by the C application.

NOTES

Maintain Round Trip Integrity

When writing a DBM filter it is *very* important to ensure that it is possible to retrieve all data that you have written when the DBM filter is in place. In practice, this means that whatever transformation is applied to the data in the Store method, the *exact* inverse operation should be applied in the Fetch method.

If you don't provide an exact inverse transformation, you will find that code like this will not behave as you expect.

```
while (my ($k, $v) = each %hash)
{
...
}
```

Depending on the transformation, you will find that one or more of the following will happen

1. The loop will never terminate.
2. Too few records will be retrieved.
3. Too many will be retrieved.
4. The loop will do the right thing for a while, but it will unexpectedly fail.

Don't mix filtered & non-filtered data in the same database file.

This is just a restatement of the previous section. Unless you are completely certain you know what you are doing, avoid mixing filtered & non-filtered data.

EXAMPLE

Say you need to interoperate with a legacy C application that stores keys as C ints and the values and null terminated UTF-8 strings. Here is how you would set that up

```
my $db = tie %hash, 'SDBM_File', ...

$db->Filter_Key_Push('int32') ;

$db->Filter_Value_Push('utf8');
$db->Filter_Value_Push('null');
```

SEE ALSO

<DB_File>, GDBM_File, NDBM_File, ODBM_File, SDBM_File, perldbmfilter(1)

AUTHOR

Paul Marquess <pmqs@cpan.org>