NAME

DB_File - Perl5 access to Berkeley DB version 1.x

SYNOPSIS

```
 use DB_File;

 [$X =] tie %hash,  'DB_File', [$filename, $flags, $mode, $DB_HASH] ;
 [$X =] tie %hash,  'DB_File', $filename, $flags, $mode, $DB_BTREE ;
 [$X =] tie @array, 'DB_File', $filename, $flags, $mode, $DB_RECNO ;

 $status = $X->del($key [, $flags]) ;
 $status = $X->put($key, $value [, $flags]) ;
 $status = $X->get($key, $value [, $flags]) ;
 $status = $X->seq($key, $value, $flags) ;
 $status = $X->sync([$flags]) ;
 $status = $X->fd ;

 # BTREE only
 $count = $X->get_dup($key) ;
 @list  = $X->get_dup($key) ;
 %list  = $X->get_dup($key, 1) ;
 $status = $X->find_dup($key, $value) ;
 $status = $X->del_dup($key, $value) ;

 # RECNO only
 $a = $X->length;
 $a = $X->pop ;
 $X->push(list);
 $a = $X->shift;
 $X->unshift(list);
 @r = $X->splice(offset, length, elements);

 # DBM Filters
 $old_filter = $db->filter_store_key  ( sub { ... } ) ;
 $old_filter = $db->filter_store_value( sub { ... } ) ;
 $old_filter = $db->filter_fetch_key  ( sub { ... } ) ;
 $old_filter = $db->filter_fetch_value( sub { ... } ) ;

 untie %hash ;
 untie @array ;
```

DESCRIPTION

DB_File is a module which allows Perl programs to make use of the facilities provided by Berkeley DB version 1.x (if you have a newer version of DB, see *Using DB_File with Berkeley DB version 2 or greater*). It is assumed that you have a copy of the Berkeley DB manual pages at hand when reading this documentation. The interface defined here mirrors the Berkeley DB interface closely.

Berkeley DB is a C library which provides a consistent interface to a number of database formats. **DB_File** provides an interface to all three of the database types currently supported by Berkeley DB.

The file types are:

DB_HASH

This database type allows arbitrary key/value pairs to be stored in data files. This is

equivalent to the functionality provided by other hashing packages like DBM, NDBM, ODBM, GDBM, and SDBM. Remember though, the files created using DB_HASH are not compatible with any of the other packages mentioned.

A default hashing algorithm, which will be adequate for most applications, is built into Berkeley DB. If you do need to use your own hashing algorithm it is possible to write your own in Perl and have **DB_File** use it instead.

DB_BTREE

The btree format allows arbitrary key/value pairs to be stored in a sorted, balanced binary tree.

As with the DB_HASH format, it is possible to provide a user defined Perl routine to perform the comparison of keys. By default, though, the keys are stored in lexical order.

DB_RECNO

DB_RECNO allows both fixed-length and variable-length flat text files to be manipulated using the same key/value pair interface as in DB_HASH and DB_BTREE. In this case the key will consist of a record (line) number.

Using DB_File with Berkeley DB version 2 or greater

Although **DB_File** is intended to be used with Berkeley DB version 1, it can also be used with version 2, 3 or 4. In this case the interface is limited to the functionality provided by Berkeley DB 1.x. Anywhere the version 2 or greater interface differs, **DB_File** arranges for it to work like version 1. This feature allows **DB_File** scripts that were built with version 1 to be migrated to version 2 or greater without any changes.

If you want to make use of the new features available in Berkeley DB 2.x or greater, use the Perl module **BerkeleyDB** instead.

Note: The database file format has changed multiple times in Berkeley DB version 2, 3 and 4. If you cannot recreate your databases, you must dump any existing databases with either the `db_dump` or the `db_dump185` utility that comes with Berkeley DB. Once you have rebuilt DB_File to use Berkeley DB version 2 or greater, your databases can be recreated using `db_load`. Refer to the Berkeley DB documentation for further details.

Please read *COPYRIGHT* before using version 2.x or greater of Berkeley DB with DB_File.

Interface to Berkeley DB

DB_File allows access to Berkeley DB files using the tie() mechanism in Perl 5 (for full details, see *"tie()" in perlfunc*). This facility allows **DB_File** to access Berkeley DB files using either an associative array (for DB_HASH & DB_BTREE file types) or an ordinary array (for the DB_RECNO file type).

In addition to the tie() interface, it is also possible to access most of the functions provided in the Berkeley DB API directly. See *THE API INTERFACE*.

Opening a Berkeley DB Database File

Berkeley DB uses the function dbopen() to open or create a database. Here is the C prototype for dbopen():

```
DB*
dbopen (const char * file, int flags, int mode,
        DBTYPE type, const void * openinfo)
```

The parameter `type` is an enumeration which specifies which of the 3 interface methods (DB_HASH, DB_BTREE or DB_RECNO) is to be used. Depending on which of these is actually chosen, the final parameter, *openinfo* points to a data structure which allows tailoring of the specific interface method.

This interface is handled slightly differently in **DB_File**. Here is an equivalent call using **DB_File**:

```
tie %array, 'DB_File', $filename, $flags, $mode, $DB_HASH ;
```

The `filename`, `flags` and `mode` parameters are the direct equivalent of their dbopen() counterparts. The final parameter $DB_HASH performs the function of both the `type` and `openinfo` parameters in dbopen().

In the example above $DB_HASH is actually a pre-defined reference to a hash object. **DB_File** has three of these pre-defined references. Apart from $DB_HASH, there is also $DB_BTREE and $DB_RECNO.

The keys allowed in each of these pre-defined references is limited to the names used in the equivalent C structure. So, for example, the $DB_HASH reference will only allow keys called `bsize`, `cachesize`, `ffactor`, `hash`, `lorder` and `nelem`.

To change one of these elements, just assign to it like this:

```
$DB_HASH->{'cachesize'} = 10000 ;
```

The three predefined variables $DB_HASH, $DB_BTREE and $DB_RECNO are usually adequate for most applications. If you do need to create extra instances of these objects, constructors are available for each file type.

Here are examples of the constructors and the valid options available for DB_HASH, DB_BTREE and DB_RECNO respectively.

```
$a = new DB_File::HASHINFO ;
$a->{'bsize'} ;
$a->{'cachesize'} ;
$a->{'ffactor'};
$a->{'hash'} ;
$a->{'lorder'} ;
$a->{'nelem'} ;

$b = new DB_File::BTREEINFO ;
$b->{'flags'} ;
$b->{'cachesize'} ;
$b->{'maxkeypage'} ;
$b->{'minkeypage'} ;
$b->{'psize'} ;
$b->{'compare'} ;
$b->{'prefix'} ;
$b->{'lorder'} ;

$c = new DB_File::RECNOINFO ;
$c->{'bval'} ;
$c->{'cachesize'} ;
$c->{'psize'} ;
$c->{'flags'} ;
$c->{'lorder'} ;
$c->{'reclen'} ;
$c->{'bfname'} ;
```

The values stored in the hashes above are mostly the direct equivalent of their C counterpart. Like their C counterparts, all are set to a default values - that means you don't have to set *all* of the values when you only want to change one. Here is an example:

```
$a = new DB_File::HASHINFO ;
```

```
 $a->{'cachesize'} =  12345 ;
 tie %y, 'DB_File', "filename", $flags, 0777, $a ;
```

A few of the options need extra discussion here. When used, the C equivalent of the keys `hash`, `compare` and `prefix` store pointers to C functions. In **DB_File** these keys are used to store references to Perl subs. Below are templates for each of the subs:

```
    sub hash
    {
        my ($data) = @_ ;
        ...
        # return the hash value for $data
 return $hash ;
    }

    sub compare
    {
 my ($key, $key2) = @_ ;
        ...
        # return  0 if $key1 eq $key2
        #        -1 if $key1 lt $key2
        #         1 if $key1 gt $key2
        return (-1 , 0 or 1) ;
    }

    sub prefix
    {
 my ($key, $key2) = @_ ;
        ...
        # return number of bytes of $key2 which are
        # necessary to determine that it is greater than $key1
        return $bytes ;
    }
```

See *Changing the BTREE sort order* for an example of using the `compare` template.

If you are using the DB_RECNO interface and you intend making use of `bval`, you should check out *The 'bval' Option*.

Default Parameters

It is possible to omit some or all of the final 4 parameters in the call to `tie` and let them take default values. As DB_HASH is the most common file format used, the call:

```
    tie %A, "DB_File", "filename" ;
```

is equivalent to:

```
    tie %A, "DB_File", "filename", O_CREAT|O_RDWR, 0666, $DB_HASH ;
```

It is also possible to omit the filename parameter as well, so the call:

```
    tie %A, "DB_File" ;
```

is equivalent to:

```
    tie %A, "DB_File", undef, O_CREAT|O_RDWR, 0666, $DB_HASH ;
```

See *In Memory Databases* for a discussion on the use of `undef` in place of a filename.

In Memory Databases

Berkeley DB allows the creation of in-memory databases by using NULL (that is, a `(char *)0` in C) in place of the filename. **DB_File** uses `undef` instead of NULL to provide this functionality.

DB_HASH

The DB_HASH file format is probably the most commonly used of the three file formats that **DB_File** supports. It is also very straightforward to use.

A Simple Example

This example shows how to create a database, add key/value pairs to the database, delete keys/value pairs and finally how to enumerate the contents of the database.

```
use warnings ;
use strict ;
use DB_File ;
our (%h, $k, $v) ;

unlink "fruit" ;
tie %h, "DB_File", "fruit", O_RDWR|O_CREAT, 0666, $DB_HASH
    or die "Cannot open file 'fruit': $!\n";

# Add a few key/value pairs to the file
$h{"apple"} = "red" ;
$h{"orange"} = "orange" ;
$h{"banana"} = "yellow" ;
$h{"tomato"} = "red" ;

# Check for existence of a key
print "Banana Exists\n\n" if $h{"banana"} ;

# Delete a key/value pair.
delete $h{"apple"} ;

# print the contents of the file
while (($k, $v) = each %h)
  { print "$k -> $v\n" }

untie %h ;
```

here is the output:

```
Banana Exists

orange -> orange
tomato -> red
banana -> yellow
```

Note that the like ordinary associative arrays, the order of the keys retrieved is in an apparently random order.

DB_BTREE

The DB_BTREE format is useful when you want to store data in a given order. By default the keys will be stored in lexical order, but as you will see from the example shown in the next section, it is very easy to define your own sorting function.

Changing the BTREE sort order

This script shows how to override the default sorting algorithm that BTREE uses. Instead of using the normal lexical ordering, a case insensitive compare function will be used.

```
use warnings ;
use strict ;
use DB_File ;

my %h ;

sub Compare
{
    my ($key1, $key2) = @_ ;
    "\L$key1" cmp "\L$key2" ;
}

# specify the Perl sub that will do the comparison
$DB_BTREE->{'compare'} = \&Compare ;

unlink "tree" ;
tie %h, "DB_File", "tree", O_RDWR|O_CREAT, 0666, $DB_BTREE
    or die "Cannot open file 'tree': $!\n" ;

# Add a key/value pair to the file
$h{'Wall'} = 'Larry' ;
$h{'Smith'} = 'John' ;
$h{'mouse'} = 'mickey' ;
$h{'duck'}  = 'donald' ;

# Delete
delete $h{"duck"} ;

# Cycle through the keys printing them in order.
# Note it is not necessary to sort the keys as
# the btree will have kept them in order automatically.
foreach (keys %h)
  { print "$_\n" }

untie %h ;
```

Here is the output from the code above.

```
mouse
Smith
Wall
```

There are a few point to bear in mind if you want to change the ordering in a BTREE database:

1. The new compare function must be specified when you create the database.

2. You cannot change the ordering once the database has been created. Thus you must use the same compare function every time you access the database.

3 Duplicate keys are entirely defined by the comparison function. In the case-insensitive example above, the keys: 'KEY' and 'key' would be considered duplicates, and assigning to the second one would overwrite the first. If duplicates are allowed for (with the R_DUP flag discussed below), only a single copy of duplicate keys is stored in the database --- so (again with example above) assigning three values to the keys: 'KEY', 'Key', and 'key' would leave just the first key: 'KEY' in the database with three values. For some situations this results in information loss, so care should be taken to provide fully qualified comparison functions when necessary. For example, the above comparison routine could be modified to additionally compare case-sensitively if two keys are equal in the case insensitive comparison:

```
sub compare {
    my($key1, $key2) = @_;
    lc $key1 cmp lc $key2 ||
    $key1 cmp $key2;
}
```

And now you will only have duplicates when the keys themselves are truly the same. (note: in versions of the db library prior to about November 1996, such duplicate keys were retained so it was possible to recover the original keys in sets of keys that compared as equal).

Handling Duplicate Keys

The BTREE file type optionally allows a single key to be associated with an arbitrary number of values. This option is enabled by setting the flags element of $DB_BTREE to R_DUP when creating the database.

There are some difficulties in using the tied hash interface if you want to manipulate a BTREE database with duplicate keys. Consider this code:

```
use warnings ;
use strict ;
use DB_File ;

my ($filename, %h) ;

$filename = "tree" ;
unlink $filename ;

# Enable duplicate records
$DB_BTREE->{'flags'} = R_DUP ;

tie %h, "DB_File", $filename, O_RDWR|O_CREAT, 0666, $DB_BTREE
 or die "Cannot open $filename: $!\n";

# Add some key/value pairs to the file
$h{'Wall'} = 'Larry' ;
$h{'Wall'} = 'Brick' ; # Note the duplicate key
$h{'Wall'} = 'Brick' ; # Note the duplicate key and value
$h{'Smith'} = 'John' ;
```

```
    $h{'mouse'} = 'mickey' ;

    # iterate through the associative array
    # and print each key/value pair.
    foreach (sort keys %h)
      { print "$_  -> $h{$_}\n" }

    untie %h ;
```

Here is the output:

```
    Smith   -> John
    Wall    -> Larry
    Wall    -> Larry
    Wall    -> Larry
    mouse   -> mickey
```

As you can see 3 records have been successfully created with key `Wall` - the only thing is, when they are retrieved from the database they *seem* to have the same value, namely `Larry`. The problem is caused by the way that the associative array interface works. Basically, when the associative array interface is used to fetch the value associated with a given key, it will only ever retrieve the first value.

Although it may not be immediately obvious from the code above, the associative array interface can be used to write values with duplicate keys, but it cannot be used to read them back from the database.

The way to get around this problem is to use the Berkeley DB API method called `seq`. This method allows sequential access to key/value pairs. See *THE API INTERFACE* for details of both the `seq` method and the API in general.

Here is the script above rewritten using the `seq` API method.

```
    use warnings ;
    use strict ;
    use DB_File ;

    my ($filename, $x, %h, $status, $key, $value) ;

    $filename = "tree" ;
    unlink $filename ;

    # Enable duplicate records
    $DB_BTREE->{'flags'} = R_DUP ;

    $x = tie %h, "DB_File", $filename, O_RDWR|O_CREAT, 0666, $DB_BTREE
 or die "Cannot open $filename: $!\n";

    # Add some key/value pairs to the file
    $h{'Wall'} = 'Larry' ;
    $h{'Wall'} = 'Brick' ; # Note the duplicate key
    $h{'Wall'} = 'Brick' ; # Note the duplicate key and value
    $h{'Smith'} = 'John' ;
    $h{'mouse'} = 'mickey' ;
```

```
    # iterate through the btree using seq
    # and print each key/value pair.
    $key = $value = 0 ;
    for ($status = $x->seq($key, $value, R_FIRST) ;
         $status == 0 ;
         $status = $x->seq($key, $value, R_NEXT) )
      {  print "$key -> $value\n" }

    undef $x ;
    untie %h ;
```

that prints:

```
    Smith   -> John
    Wall    -> Brick
    Wall    -> Brick
    Wall    -> Larry
    mouse   -> mickey
```

This time we have got all the key/value pairs, including the multiple values associated with the key `Wall`.

To make life easier when dealing with duplicate keys, **DB_File** comes with a few utility methods.

The get_dup() Method

The `get_dup` method assists in reading duplicate values from BTREE databases. The method can take the following forms:

```
    $count = $x->get_dup($key) ;
    @list  = $x->get_dup($key) ;
    %list  = $x->get_dup($key, 1) ;
```

In a scalar context the method returns the number of values associated with the key, `$key`.

In list context, it returns all the values which match `$key`. Note that the values will be returned in an apparently random order.

In list context, if the second parameter is present and evaluates TRUE, the method returns an associative array. The keys of the associative array correspond to the values that matched in the BTREE and the values of the array are a count of the number of times that particular value occurred in the BTREE.

So assuming the database created above, we can use `get_dup` like this:

```
    use warnings ;
    use strict ;
    use DB_File ;

    my ($filename, $x, %h) ;

    $filename = "tree" ;

    # Enable duplicate records
    $DB_BTREE->{'flags'} = R_DUP ;

    $x = tie %h, "DB_File", $filename, O_RDWR|O_CREAT, 0666, $DB_BTREE
```

```
 or die "Cannot open $filename: $!\n";

    my $cnt  = $x->get_dup("Wall") ;
    print "Wall occurred $cnt times\n" ;

    my %hash = $x->get_dup("Wall", 1) ;
    print "Larry is there\n" if $hash{'Larry'} ;
    print "There are $hash{'Brick'} Brick Walls\n" ;

    my @list = sort $x->get_dup("Wall") ;
    print "Wall => [@list]\n" ;

    @list = $x->get_dup("Smith") ;
    print "Smith => [@list]\n" ;

    @list = $x->get_dup("Dog") ;
    print "Dog => [@list]\n" ;
```

and it will print:

```
    Wall occurred 3 times
    Larry is there
    There are 2 Brick Walls
    Wall => [Brick Brick Larry]
    Smith => [John]
    Dog => []
```

The find_dup() Method

```
    $status = $X->find_dup($key, $value) ;
```

This method checks for the existence of a specific key/value pair. If the pair exists, the cursor is left pointing to the pair and the method returns 0. Otherwise the method returns a non-zero value.

Assuming the database from the previous example:

```
    use warnings ;
    use strict ;
    use DB_File ;

    my ($filename, $x, %h, $found) ;

    $filename = "tree" ;

    # Enable duplicate records
    $DB_BTREE->{'flags'} = R_DUP ;

    $x = tie %h, "DB_File", $filename, O_RDWR|O_CREAT, 0666, $DB_BTREE
 or die "Cannot open $filename: $!\n";

    $found = ( $x->find_dup("Wall", "Larry") == 0 ? "" : "not") ;
    print "Larry Wall is $found there\n" ;
```

```
    $found = ( $x->find_dup("Wall", "Harry") == 0 ? "" : "not") ;
    print "Harry Wall is $found there\n" ;

    undef $x ;
    untie %h ;
```

prints this

```
    Larry Wall is  there
    Harry Wall is not there
```

The del_dup() Method

```
    $status = $X->del_dup($key, $value) ;
```

This method deletes a specific key/value pair. It returns 0 if they exist and have been deleted successfully. Otherwise the method returns a non-zero value.

Again assuming the existence of the `tree` database

```
    use warnings ;
    use strict ;
    use DB_File ;

    my ($filename, $x, %h, $found) ;

    $filename = "tree" ;

    # Enable duplicate records
    $DB_BTREE->{'flags'} = R_DUP ;

    $x = tie %h, "DB_File", $filename, O_RDWR|O_CREAT, 0666, $DB_BTREE
 or die "Cannot open $filename: $!\n";

    $x->del_dup("Wall", "Larry") ;

    $found = ( $x->find_dup("Wall", "Larry") == 0 ? "" : "not") ;
    print "Larry Wall is $found there\n" ;

    undef $x ;
    untie %h ;
```

prints this

```
    Larry Wall is not there
```

Matching Partial Keys

The BTREE interface has a feature which allows partial keys to be matched. This functionality is *only* available when the `seq` method is used along with the R_CURSOR flag.

```
    $x->seq($key, $value, R_CURSOR) ;
```

Here is the relevant quote from the dbopen man page where it defines the use of the R_CURSOR

flag with seq:

```
    Note, for the DB_BTREE access method, the returned key is not
    necessarily an exact match for the specified key. The returned key
    is the smallest key greater than or equal to the specified key,
    permitting partial key matches and range searches.
```

In the example script below, the `match` sub uses this feature to find and print the first matching key/value pair given a partial key.

```
    use warnings ;
    use strict ;
    use DB_File ;
    use Fcntl ;

    my ($filename, $x, %h, $st, $key, $value) ;

    sub match
    {
        my $key = shift ;
        my $value = 0;
        my $orig_key = $key ;
        $x->seq($key, $value, R_CURSOR) ;
        print "$orig_key\t-> $key\t-> $value\n" ;
    }

    $filename = "tree" ;
    unlink $filename ;

    $x = tie %h, "DB_File", $filename, O_RDWR|O_CREAT, 0666, $DB_BTREE
        or die "Cannot open $filename: $!\n";

    # Add some key/value pairs to the file
    $h{'mouse'} = 'mickey' ;
    $h{'Wall'} = 'Larry' ;
    $h{'Walls'} = 'Brick' ;
    $h{'Smith'} = 'John' ;

    $key = $value = 0 ;
    print "IN ORDER\n" ;
    for ($st = $x->seq($key, $value, R_FIRST) ;
  $st == 0 ;
         $st = $x->seq($key, $value, R_NEXT) )

      {  print "$key -> $value\n" }

    print "\nPARTIAL MATCH\n" ;

    match "Wa" ;
    match "A" ;
    match "a" ;

    undef $x ;
```

```
untie %h ;
```

Here is the output:

```
IN ORDER
Smith -> John
Wall  -> Larry
Walls -> Brick
mouse -> mickey

PARTIAL MATCH
Wa -> Wall  -> Larry
A  -> Smith -> John
a  -> mouse -> mickey
```

DB_RECNO

DB_RECNO provides an interface to flat text files. Both variable and fixed length records are supported.

In order to make RECNO more compatible with Perl, the array offset for all RECNO arrays begins at 0 rather than 1 as in Berkeley DB.

As with normal Perl arrays, a RECNO array can be accessed using negative indexes. The index -1 refers to the last element of the array, -2 the second last, and so on. Attempting to access an element before the start of the array will raise a fatal run-time error.

The 'bval' Option

The operation of the bval option warrants some discussion. Here is the definition of bval from the Berkeley DB 1.85 recno manual page:

```
The delimiting byte to be used to mark  the  end  of  a
record for variable-length records, and the pad charac-
ter for fixed-length records.  If no  value  is  speci-
fied,  newlines  (``\n'')  are  used to mark the end of
variable-length records and  fixed-length  records  are
padded with spaces.
```

The second sentence is wrong. In actual fact bval will only default to `"\n"` when the openinfo parameter in dbopen is NULL. If a non-NULL openinfo parameter is used at all, the value that happens to be in bval will be used. That means you always have to specify bval when making use of any of the options in the openinfo parameter. This documentation error will be fixed in the next release of Berkeley DB.

That clarifies the situation with regards Berkeley DB itself. What about **DB_File**? Well, the behavior defined in the quote above is quite useful, so **DB_File** conforms to it.

That means that you can specify other options (e.g. cachesize) and still have bval default to `"\n"` for variable length records, and space for fixed length records.

Also note that the bval option only allows you to specify a single byte as a delimiter.

A Simple Example

Here is a simple example that uses RECNO (if you are using a version of Perl earlier than 5.004_57 this example won't work -- see *Extra RECNO Methods* for a workaround).

```
use warnings ;
use strict ;
```

```
use DB_File ;

my $filename = "text" ;
unlink $filename ;

my @h ;
tie @h, "DB_File", $filename, O_RDWR|O_CREAT, 0666, $DB_RECNO
    or die "Cannot open file 'text': $!\n" ;

# Add a few key/value pairs to the file
$h[0] = "orange" ;
$h[1] = "blue" ;
$h[2] = "yellow" ;

push @h, "green", "black" ;

my $elements = scalar @h ;
print "The array contains $elements entries\n" ;

my $last = pop @h ;
print "popped $last\n" ;

unshift @h, "white" ;
my $first = shift @h ;
print "shifted $first\n" ;

# Check for existence of a key
print "Element 1 Exists with value $h[1]\n" if $h[1] ;

# use a negative index
print "The last element is $h[-1]\n" ;
print "The 2nd last element is $h[-2]\n" ;

untie @h ;
```

Here is the output from the script:

```
The array contains 5 entries
popped black
shifted white
Element 1 Exists with value blue
The last element is green
The 2nd last element is yellow
```

Extra RECNO Methods

If you are using a version of Perl earlier than 5.004_57, the tied array interface is quite limited. In the example script above `push`, `pop`, `shift`, `unshift` or determining the array length will not work with a tied array.

To make the interface more useful for older versions of Perl, a number of methods are supplied with **DB_File** to simulate the missing array operations. All these methods are accessed via the object returned from the tie call.

Here are the methods:

$X->push(list) ;

Pushes the elements of `list` to the end of the array.

$value = $X->pop ;

Removes and returns the last element of the array.

$X->shift

Removes and returns the first element of the array.

$X->unshift(list) ;

Pushes the elements of `list` to the start of the array.

$X->length

Returns the number of elements in the array.

$X->splice(offset, length, elements);

Returns a splice of the array.

Another Example

Here is a more complete example that makes use of some of the methods described above. It also makes use of the API interface directly (see *THE API INTERFACE*).

```
use warnings ;
use strict ;
my (@h, $H, $file, $i) ;
use DB_File ;
use Fcntl ;

$file = "text" ;

unlink $file ;

$H = tie @h, "DB_File", $file, O_RDWR|O_CREAT, 0666, $DB_RECNO
    or die "Cannot open file $file: $!\n" ;

# first create a text file to play with
$h[0] = "zero" ;
$h[1] = "one" ;
$h[2] = "two" ;
$h[3] = "three" ;
$h[4] = "four" ;

# Print the records in order.
#
# The length method is needed here because evaluating a tied
# array in a scalar context does not return the number of
# elements in the array.

print "\nORIGINAL\n" ;
foreach $i (0 .. $H->length - 1) {
    print "$i: $h[$i]\n" ;
}
```

```
    # use the push & pop methods
    $a = $H->pop ;
    $H->push("last") ;
    print "\nThe last record was [$a]\n" ;

    # and the shift & unshift methods
    $a = $H->shift ;
    $H->unshift("first") ;
    print "The first record was [$a]\n" ;

    # Use the API to add a new record after record 2.
    $i = 2 ;
    $H->put($i, "Newbie", R_IAFTER) ;

    # and a new record before record 1.
    $i = 1 ;
    $H->put($i, "New One", R_IBEFORE) ;

    # delete record 3
    $H->del(3) ;

    # now print the records in reverse order
    print "\nREVERSE\n" ;
    for ($i = $H->length - 1 ; $i >= 0 ; -- $i)
      { print "$i: $h[$i]\n" }

    # same again, but use the API functions instead
    print "\nREVERSE again\n" ;
    my ($s, $k, $v)  = (0, 0, 0) ;
    for ($s = $H->seq($k, $v, R_LAST) ;
             $s == 0 ;
             $s = $H->seq($k, $v, R_PREV))
      { print "$k: $v\n" }

    undef $H ;
    untie @h ;
```

and this is what it outputs:

```
    ORIGINAL
    0: zero
    1: one
    2: two
    3: three
    4: four

    The last record was [four]
    The first record was [zero]

    REVERSE
    5: last
    4: three
    3: Newbie
```

```
2: one
1: New One
0: first

REVERSE again
5: last
4: three
3: Newbie
2: one
1: New One
0: first
```

Notes:

1. Rather than iterating through the array, @h like this:

   ```
   foreach $i (@h)
   ```

   it is necessary to use either this:

   ```
   foreach $i (0 .. $H->length - 1)
   ```

   or this:

   ```
   for ($a = $H->get($k, $v, R_FIRST) ;
        $a == 0 ;
        $a = $H->get($k, $v, R_NEXT) )
   ```

2. Notice that both times the `put` method was used the record index was specified using a variable, `$i`, rather than the literal value itself. This is because `put` will return the record number of the inserted line via that parameter.

THE API INTERFACE

As well as accessing Berkeley DB using a tied hash or array, it is also possible to make direct use of most of the API functions defined in the Berkeley DB documentation.

To do this you need to store a copy of the object returned from the tie.

```
$db = tie %hash, "DB_File", "filename" ;
```

Once you have done that, you can access the Berkeley DB API functions as **DB_File** methods directly like this:

```
$db->put($key, $value, R_NOOVERWRITE) ;
```

Important: If you have saved a copy of the object returned from `tie`, the underlying database file will *not* be closed until both the tied variable is untied and all copies of the saved object are destroyed.

```
use DB_File ;
$db = tie %hash, "DB_File", "filename"
    or die "Cannot tie filename: $!" ;
...
undef $db ;
untie %hash ;
```

See *The untie() Gotcha* for more details.

All the functions defined in *dbopen* are available except for close() and dbopen() itself. The **DB_File**

method interface to the supported functions have been implemented to mirror the way Berkeley DB works whenever possible. In particular note that:

- The methods return a status value. All return 0 on success. All return -1 to signify an error and set `$!` to the exact error code. The return code 1 generally (but not always) means that the key specified did not exist in the database.

  Other return codes are defined. See below and in the Berkeley DB documentation for details. The Berkeley DB documentation should be used as the definitive source.

- Whenever a Berkeley DB function returns data via one of its parameters, the equivalent **DB_File** method does exactly the same.

- If you are careful, it is possible to mix API calls with the tied hash/array interface in the same piece of code. Although only a few of the methods used to implement the tied interface currently make use of the cursor, you should always assume that the cursor has been changed any time the tied hash/array interface is used. As an example, this code will probably not do what you expect:

```
    $X = tie %x, 'DB_File', $filename, O_RDWR|O_CREAT, 0777,
$DB_BTREE
        or die "Cannot tie $filename: $!" ;

    # Get the first key/value pair and set  the cursor
    $X->seq($key, $value, R_FIRST) ;

    # this line will modify the cursor
    $count = scalar keys %x ;

    # Get the second key/value pair.
    # oops, it didn't, it got the last key/value pair!
    $X->seq($key, $value, R_NEXT) ;
```

  The code above can be rearranged to get around the problem, like this:

```
    $X = tie %x, 'DB_File', $filename, O_RDWR|O_CREAT, 0777,
$DB_BTREE
        or die "Cannot tie $filename: $!" ;

    # this line will modify the cursor
    $count = scalar keys %x ;

    # Get the first key/value pair and set  the cursor
    $X->seq($key, $value, R_FIRST) ;

    # Get the second key/value pair.
    # worked this time.
    $X->seq($key, $value, R_NEXT) ;
```

All the constants defined in *dbopen* for use in the flags parameters in the methods defined below are also available. Refer to the Berkeley DB documentation for the precise meaning of the flags values.

Below is a list of the methods available.

$status = $X->get($key, $value [, $flags]) ;

Given a key (`$key`) this method reads the value associated with it from the database. The value read from the database is returned in the `$value` parameter.

If the key does not exist the method returns 1.

No flags are currently defined for this method.

$status = $X->put($key, $value [, $flags]) ;

Stores the key/value pair in the database.

If you use either the R_IAFTER or R_IBEFORE flags, the `$key` parameter will have the record number of the inserted key/value pair set.

Valid flags are R_CURSOR, R_IAFTER, R_IBEFORE, R_NOOVERWRITE and R_SETCURSOR.

$status = $X->del($key [, $flags]) ;

Removes all key/value pairs with key `$key` from the database.

A return code of 1 means that the requested key was not in the database.

R_CURSOR is the only valid flag at present.

$status = $X->fd ;

Returns the file descriptor for the underlying database.

See *Locking: The Trouble with fd* for an explanation for why you should not use `fd` to lock your database.

$status = $X->seq($key, $value, $flags) ;

This interface allows sequential retrieval from the database. See *dbopen* for full details.

Both the `$key` and `$value` parameters will be set to the key/value pair read from the database.

The flags parameter is mandatory. The valid flag values are R_CURSOR, R_FIRST, R_LAST, R_NEXT and R_PREV.

$status = $X->sync([$flags]) ;

Flushes any cached buffers to disk.

R_RECNOSYNC is the only valid flag at present.

DBM FILTERS

A DBM Filter is a piece of code that is be used when you *always* want to make the same transformation to all keys and/or values in a DBM database.

There are four methods associated with DBM Filters. All work identically, and each is used to install (or uninstall) a single DBM Filter. Each expects a single parameter, namely a reference to a sub. The only difference between them is the place that the filter is installed.

To summarise:

filter_store_key

If a filter has been installed with this method, it will be invoked every time you write a key to a DBM database.

filter_store_value

If a filter has been installed with this method, it will be invoked every time you write a value to a DBM database.

filter_fetch_key

If a filter has been installed with this method, it will be invoked every time you read a key from a DBM database.

filter_fetch_value

If a filter has been installed with this method, it will be invoked every time you read a value

from a DBM database.

You can use any combination of the methods, from none, to all four.

All filter methods return the existing filter, if present, or `undef` in not.

To delete a filter pass `undef` to it.

The Filter

When each filter is called by Perl, a local copy of $_ will contain the key or value to be filtered. Filtering is achieved by modifying the contents of $_. The return code from the filter is ignored.

An Example -- the NULL termination problem.

Consider the following scenario. You have a DBM database that you need to share with a third-party C application. The C application assumes that *all* keys and values are NULL terminated. Unfortunately when Perl writes to DBM databases it doesn't use NULL termination, so your Perl application will have to manage NULL termination itself. When you write to the database you will have to use something like this:

```
    $hash{"$key\0"} = "$value\0" ;
```

Similarly the NULL needs to be taken into account when you are considering the length of existing keys/values.

It would be much better if you could ignore the NULL terminations issue in the main application code and have a mechanism that automatically added the terminating NULL to all keys and values whenever you write to the database and have them removed when you read from the database. As I'm sure you have already guessed, this is a problem that DBM Filters can fix very easily.

```
    use warnings ;
    use strict ;
    use DB_File ;

    my %hash ;
    my $filename = "filt" ;
    unlink $filename ;

    my $db = tie %hash, 'DB_File', $filename, O_CREAT|O_RDWR, 0666,
$DB_HASH
      or die "Cannot open $filename: $!\n" ;

    # Install DBM Filters
    $db->filter_fetch_key  ( sub { s/\0$//    } ) ;
    $db->filter_store_key  ( sub { $_ .= "\0" } ) ;
    $db->filter_fetch_value( sub { s/\0$//    } ) ;
    $db->filter_store_value( sub { $_ .= "\0" } ) ;

    $hash{"abc"} = "def" ;
    my $a = $hash{"ABC"} ;
    # ...
    undef $db ;
    untie %hash ;
```

Hopefully the contents of each of the filters should be self-explanatory. Both "fetch" filters remove the terminating NULL, and both "store" filters add a terminating NULL.

Another Example -- Key is a C int.

Here is another real-life example. By default, whenever Perl writes to a DBM database it always writes the key and value as strings. So when you use this:

```
    $hash{12345} = "something" ;
```

the key 12345 will get stored in the DBM database as the 5 byte string "12345". If you actually want the key to be stored in the DBM database as a C int, you will have to use `pack` when writing, and `unpack` when reading.

Here is a DBM Filter that does it:

```
    use warnings ;
    use strict ;
    use DB_File ;
    my %hash ;
    my $filename = "filt" ;
    unlink $filename ;

    my $db = tie %hash, 'DB_File', $filename, O_CREAT|O_RDWR, 0666,
$DB_HASH
      or die "Cannot open $filename: $!\n" ;

    $db->filter_fetch_key  ( sub { $_ = unpack("i", $_) } ) ;
    $db->filter_store_key  ( sub { $_ = pack ("i", $_) } ) ;
    $hash{123} = "def" ;
    # ...
    undef $db ;
    untie %hash ;
```

This time only two filters have been used -- we only need to manipulate the contents of the key, so it wasn't necessary to install any value filters.

HINTS AND TIPS

Locking: The Trouble with fd

Until version 1.72 of this module, the recommended technique for locking **DB_File** databases was to flock the filehandle returned from the "fd" function. Unfortunately this technique has been shown to be fundamentally flawed (Kudos to David Harris for tracking this down). Use it at your own peril!

The locking technique went like this.

```
    $db = tie(%db, 'DB_File', 'foo.db', O_CREAT|O_RDWR, 0644)
        || die "dbcreat foo.db $!";
    $fd = $db->fd;
    open(DB_FH, "+<&=$fd") || die "dup $!";
    flock (DB_FH, LOCK_EX) || die "flock: $!";
    ...
    $db{"Tom"} = "Jerry" ;
    ...
    flock(DB_FH, LOCK_UN);
    undef $db;
    untie %db;
    close(DB_FH);
```

In simple terms, this is what happens:

1. Use "tie" to open the database.
2. Lock the database with fd & flock.
3. Read & Write to the database.
4. Unlock and close the database.

Here is the crux of the problem. A side-effect of opening the **DB_File** database in step 2 is that an initial block from the database will get read from disk and cached in memory.

To see why this is a problem, consider what can happen when two processes, say "A" and "B", both want to update the same **DB_File** database using the locking steps outlined above. Assume process "A" has already opened the database and has a write lock, but it hasn't actually updated the database yet (it has finished step 2, but not started step 3 yet). Now process "B" tries to open the same database - step 1 will succeed, but it will block on step 2 until process "A" releases the lock. The important thing to notice here is that at this point in time both processes will have cached identical initial blocks from the database.

Now process "A" updates the database and happens to change some of the data held in the initial buffer. Process "A" terminates, flushing all cached data to disk and releasing the database lock. At this point the database on disk will correctly reflect the changes made by process "A".

With the lock released, process "B" can now continue. It also updates the database and unfortunately it too modifies the data that was in its initial buffer. Once that data gets flushed to disk it will overwrite some/all of the changes process "A" made to the database.

The result of this scenario is at best a database that doesn't contain what you expect. At worst the database will corrupt.

The above won't happen every time competing process update the same **DB_File** database, but it does illustrate why the technique should not be used.

Safe ways to lock a database

Starting with version 2.x, Berkeley DB has internal support for locking. The companion module to this one, **BerkeleyDB**, provides an interface to this locking functionality. If you are serious about locking Berkeley DB databases, I strongly recommend using **BerkeleyDB**.

If using **BerkeleyDB** isn't an option, there are a number of modules available on CPAN that can be used to implement locking. Each one implements locking differently and has different goals in mind. It is therefore worth knowing the difference, so that you can pick the right one for your application. Here are the three locking wrappers:

Tie::DB_Lock

A **DB_File** wrapper which creates copies of the database file for read access, so that you have a kind of a multiversioning concurrent read system. However, updates are still serial. Use for databases where reads may be lengthy and consistency problems may occur.

Tie::DB_LockFile

A **DB_File** wrapper that has the ability to lock and unlock the database while it is being used. Avoids the tie-before-flock problem by simply re-tie-ing the database when you get or drop a lock. Because of the flexibility in dropping and re-acquiring the lock in the middle of a session, this can be massaged into a system that will work with long updates and/or reads if the application follows the hints in the POD documentation.

DB_File::Lock

An extremely lightweight **DB_File** wrapper that simply flocks a lockfile before tie-ing the database and drops the lock after the untie. Allows one to use the same lockfile for multiple

databases to avoid deadlock problems, if desired. Use for databases where updates are reads are quick and simple flock locking semantics are enough.

Sharing Databases With C Applications

There is no technical reason why a Berkeley DB database cannot be shared by both a Perl and a C application.

The vast majority of problems that are reported in this area boil down to the fact that C strings are NULL terminated, whilst Perl strings are not. See *DBM FILTERS* for a generic way to work around this problem.

Here is a real example. Netscape 2.0 keeps a record of the locations you visit along with the time you last visited them in a DB_HASH database. This is usually stored in the file *~/.netscape/history.db*. The key field in the database is the location string and the value field is the time the location was last visited stored as a 4 byte binary value.

If you haven't already guessed, the location string is stored with a terminating NULL. This means you need to be careful when accessing the database.

Here is a snippet of code that is loosely based on Tom Christiansen's *ggh* script (available from your nearest CPAN archive in *authors/id/TOMC/scripts/nshist.gz*).

```
use warnings ;
use strict ;
use DB_File ;
use Fcntl ;

my ($dotdir, $HISTORY, %hist_db, $href, $binary_time, $date) ;
$dotdir = $ENV{HOME} || $ENV{LOGNAME};

$HISTORY = "$dotdir/.netscape/history.db";

tie %hist_db, 'DB_File', $HISTORY
    or die "Cannot open $HISTORY: $!\n" ;;

# Dump the complete database
while ( ($href, $binary_time) = each %hist_db ) {

    # remove the terminating NULL
    $href =~ s/\x00$// ;

    # convert the binary time into a user friendly string
    $date = localtime unpack("V", $binary_time);
    print "$date $href\n" ;
}

# check for the existence of a specific key
# remember to add the NULL
if ( $binary_time = $hist_db{"http://mox.perl.com/\x00"} ) {
    $date = localtime unpack("V", $binary_time) ;
    print "Last visited mox.perl.com on $date\n" ;
}
else {
    print "Never visited mox.perl.com\n"
}
```

```
    untie %hist_db ;
```

The untie() Gotcha

If you make use of the Berkeley DB API, it is *very* strongly recommended that you read *"The untie Gotcha" in perltie*.

Even if you don't currently make use of the API interface, it is still worth reading it.

Here is an example which illustrates the problem from a **DB_File** perspective:

```
    use DB_File ;
    use Fcntl ;

    my %x ;
    my $X ;

    $X = tie %x, 'DB_File', 'tst.fil' , O_RDWR|O_TRUNC
        or die "Cannot tie first time: $!" ;

    $x{123} = 456 ;

    untie %x ;

    tie %x, 'DB_File', 'tst.fil' , O_RDWR|O_CREAT
        or die "Cannot tie second time: $!" ;

    untie %x ;
```

When run, the script will produce this error message:

```
    Cannot tie second time: Invalid argument at bad.file line 14.
```

Although the error message above refers to the second tie() statement in the script, the source of the problem is really with the untie() statement that precedes it.

Having read *perltie* you will probably have already guessed that the error is caused by the extra copy of the tied object stored in $X. If you haven't, then the problem boils down to the fact that the **DB_File** destructor, DESTROY, will not be called until *all* references to the tied object are destroyed. Both the tied variable, %x, and $X above hold a reference to the object. The call to untie() will destroy the first, but $X still holds a valid reference, so the destructor will not get called and the database file *tst.fil* will remain open. The fact that Berkeley DB then reports the attempt to open a database that is already open via the catch-all "Invalid argument" doesn't help.

If you run the script with the `-w` flag the error message becomes:

```
    untie attempted while 1 inner references still exist at bad.file line
12.
    Cannot tie second time: Invalid argument at bad.file line 14.
```

which pinpoints the real problem. Finally the script can now be modified to fix the original problem by destroying the API object before the untie:

```
    ...
    $x{123} = 456 ;
```

```
undef $X ;
untie %x ;

$X = tie %x, 'DB_File', 'tst.fil' , O_RDWR|O_CREAT
...
```

COMMON QUESTIONS

Why is there Perl source in my database?

If you look at the contents of a database file created by DB_File, there can sometimes be part of a Perl script included in it.

This happens because Berkeley DB uses dynamic memory to allocate buffers which will subsequently be written to the database file. Being dynamic, the memory could have been used for anything before DB malloced it. As Berkeley DB doesn't clear the memory once it has been allocated, the unused portions will contain random junk. In the case where a Perl script gets written to the database, the random junk will correspond to an area of dynamic memory that happened to be used during the compilation of the script.

Unless you don't like the possibility of there being part of your Perl scripts embedded in a database file, this is nothing to worry about.

How do I store complex data structures with DB_File?

Although **DB_File** cannot do this directly, there is a module which can layer transparently over **DB_File** to accomplish this feat.

Check out the MLDBM module, available on CPAN in the directory *modules/by-module/MLDBM*.

What does "Invalid Argument" mean?

You will get this error message when one of the parameters in the `tie` call is wrong. Unfortunately there are quite a few parameters to get wrong, so it can be difficult to figure out which one it is.

Here are a couple of possibilities:

1. Attempting to reopen a database without closing it.
2. Using the O_WRONLY flag.

What does "Bareword 'DB_File' not allowed" mean?

You will encounter this particular error message when you have the `strict 'subs'` pragma (or the full strict pragma) in your script. Consider this script:

```
use warnings ;
use strict ;
use DB_File ;
my %x ;
tie %x, DB_File, "filename" ;
```

Running it produces the error in question:

```
Bareword "DB_File" not allowed while "strict subs" in use
```

To get around the error, place the word `DB_File` in either single or double quotes, like this:

```
tie %x, "DB_File", "filename" ;
```

Although it might seem like a real pain, it is really worth the effort of having a `use strict` in all your scripts.

REFERENCES

Articles that are either about **DB_File** or make use of it.

1. *Full-Text Searching in Perl*, Tim Kientzle (tkientzle@ddj.com), Dr. Dobb's Journal, Issue 295, January 1999, pp 34-41

HISTORY

Moved to the Changes file.

BUGS

Some older versions of Berkeley DB had problems with fixed length records using the RECNO file format. This problem has been fixed since version 1.85 of Berkeley DB.

I am sure there are bugs in the code. If you do find any, or can suggest any enhancements, I would welcome your comments.

AVAILABILITY

DB_File comes with the standard Perl source distribution. Look in the directory *ext/DB_File*. Given the amount of time between releases of Perl the version that ships with Perl is quite likely to be out of date, so the most recent version can always be found on CPAN (see *"CPAN" in perlmodlib* for details), in the directory *modules/by-module/DB_File*.

This version of **DB_File** will work with either version 1.x, 2.x or 3.x of Berkeley DB, but is limited to the functionality provided by version 1.

The official web site for Berkeley DB is *http://www.oracle.com/technology/products/berkeley-db/db/index.html*. All versions of Berkeley DB are available there.

Alternatively, Berkeley DB version 1 is available at your nearest CPAN archive in *src/misc/db.1.85.tar.gz*.

COPYRIGHT

Although **DB_File** is covered by the Perl license, the library it makes use of, namely Berkeley DB, is not. Berkeley DB has its own copyright and its own license. Please take the time to read it.

Here are are few words taken from the Berkeley DB FAQ (at *http://www.oracle.com/technology/products/berkeley-db/db/index.html*) regarding the license:

```
Do I have to license DB to use it in Perl scripts?

No. The Berkeley DB license requires that software that uses
Berkeley DB be freely redistributable. In the case of Perl, that
software is Perl, and not your scripts. Any Perl scripts that you
write are your property, including scripts that make use of
Berkeley DB. Neither the Perl license nor the Berkeley DB license
place any restriction on what you may do with them.
```

If you are in any doubt about the license situation, contact either the Berkeley DB authors or the author of DB_File. See *AUTHOR* for details.

SEE ALSO

perl, *dbopen(3)*, *hash(3)*, *recno(3)*, *btree(3)*, *perldbmfilter*

AUTHOR

The DB_File interface was written by Paul Marquess <pmqs@cpan.org>.