NAME

GDBM - The GNU database manager. Includes **dbm** and **ndbm** compatability. (Version 1.8.3.)

SYNOPSIS

```
#include <gdbm.h>

extern gdbm_error
gdbm_errno

extern char
*gdbm_version

GDBM_FILE
gdbm_open (name, block_size, read_write, mode, fatal_func)
char * name;
int block_size, read_write, mode;
void (*fatal_func) ();

void
gdbm_close (dbf)
GDBM_FILE dbf;

int
gdbm_store (dbf, key, content, flag)
GDBM_FILE dbf;
datum key, content;
int flag;

datum
gdbm_fetch (dbf, key)
GDBM_FILE dbf;
datum key;

int
gdbm_delete (dbf, key)
GDBM_FILE dbf;
datum key;

datum
gdbm_firstkey (dbf)
GDBM_FILE dbf;

datum
gdbm_nextkey (dbf, key)
GDBM_FILE dbf;
datum key;

int
gdbm_reorganize (dbf)
GDBM_FILE dbf;

void
gdbm_sync (dbf)
GDBM_FILE dbf;

int
gdbm_exists (dbf, key)
GDBM_FILE dbf;
datum key;

char *
gdbm_strerror (errno)
gdbm_error errno;

int
gdbm_setopt (dbf, option, value, size)
```

GDBM_FILE dbf;
int option;
int *value;
int size;

int
gdbm_fdesc (dbf)
GDBM_FILE dbf;

DBM Compatability routines:

#include <dbm.h>

int
dbminit (name)
char *name;

int
store (key, content)
datum key, content;

datum
fetch (key)
datum key;

int
delete (key)
datum key;

datum
firstkey ()

datum
nextkey (key)
datum key;

int
dbmclose ()

NDBM Compatability routines:

#include <ndbm.h>

DBM
***dbm_open (name, flags, mode)**
char *name;
int flags, mode;

void
dbm_close (file)
DBM *file;

datum
dbm_fetch (file, key)
DBM *file;
datum key;

int
dbm_store (file, key, content, flags)
DBM *file;
datum key, content;
int flags;

int
dbm_delete (file, key)
DBM *file;
datum key;

datum

dbm_firstkey (file)
DBM *file;

datum
dbm_nextkey (file)
DBM *file;

int
dbm_error (file)
DBM *file;

int
dbm_clearerr (file)
DBM *file;

int
dbm_pagfno (file)
DBM *file;

int
dbm_dirfno (file)
DBM *file;

int
dbm_rdonly (file)
DBM *file;

DESCRIPTION

GNU dbm is a library of routines that manages data files that contain key/data pairs. The access provided is that of storing, retrieval, and deletion by key and a non-sorted traversal of all keys. A process is allowed to use multiple data files at the same time.

A process that opens a gdbm file is designated as a "reader" or a "writer". Only one writer may open a gdbm file and many readers may open the file. Readers and writers can not open the gdbm file at the same time. The procedure for opening a gdbm file is:

```
GDBM_FILE dbf;

dbf = gdbm_open ( name, block_size, read_write, mode, fatal_func )
```

Name is the name of the file (the complete name, gdbm does not append any characters to this name). *Block_size* is the size of a single transfer from disk to memory. This parameter is ignored unless the file is a new file. The minimum size is 512. If it is less than 512, dbm will use the stat block size for the file system. *Read_write* can have one of the following values:
GDBM_READER reader
GDBM_WRITER writer
GDBM_WRCREAT writer - if database does not exist create new one
GDBM_NEWDB writer - create new database regardless if one exists
For the last three (writers of the database) the following may be added added to *read_write* by bitwise or: **GDBM_SYNC,** which causes all database operations to be synchronized to the disk, and **GDBM_NOLOCK,** which prevents the library from performing any locking on the database file. The option **GDBM_FAST** is now obsolete, since gdbm defaults to no-sync mode.
Mode is the file mode (see **chmod(2)** and **open(2)**) if the file is created. *(*Fatal_func) ()* is a function for dbm to call if it detects a fatal error. The only parameter of this function is a string. If the value of 0 is provided, gdbm will use a default function.

The return value *dbf* is the pointer needed by all other routines to access that gdbm file. If the return is the NULL pointer, **gdbm_open** was not successful. The errors can be found in *gdbm_errno* for gdbm errors and in *errno* for system errors. (For error codes, see gdbmerrno.h.)

In all of the following calls, the parameter *dbf* refers to the pointer returned from **gdbm_open**.

It is important that every file opened is also closed. This is needed to update the reader/writer count on the file. This is done by:

```
gdbm_close (dbf);
```

The database is used by 3 primary routines. The first stores data in the database.

```
ret = gdbm_store ( dbf, key, content, flag )
```

Dbf is the pointer returned by **gdbm_open**. *Key* is the key data. *Content* is the data to be associated with the *key*. *Flag* can have one of the following values:
GDBM_INSERT insert only, generate an error if key exists
GDBM_REPLACE replace contents if key exists.

If a reader calls **gdbm_store**, the return value will be -1. If called with GDBM_INSERT and *key* is in the database, the return value will be 1. Otherwise, the return value is 0.

NOTICE: If you store data for a key that is already in the data base, gdbm replaces the old data with the new data if called with GDBM_REPLACE. You do not get two data items for the same key and you do not get an error from gdbm_store.

NOTICE: The size in gdbm is not restricted like dbm or ndbm. Your data can be as large as you want.

To search for some data:

```
content = gdbm_fetch ( dbf, key )
```

Dbf is the pointer returned by **gdbm_open**. *Key* is the key data.

If the *dptr* element of the return value is NULL, no data was found. Otherwise the return value is a pointer to the found data. The storage space for the *dptr* element is allocated using **malloc(3C)**. **Gdbm** *does not automatically free this data. It is the programmer's responsibility to free this storage when it is no longer needed.*

To search for some data, without retrieving it:

```
ret = gdbm_exists ( dbf, key )
```

Dbf is the pointer returned by **gdbm_open**. *Key* is the key data to search for.

If the *key* is found within the database, the return value *ret* will be true. If nothing appropiate is found, *ret* will be false. This routine is useful for checking for the existance of a record, without performing the memory allocation done by **gdbm_fetch**.

To remove some data from the database:

```
ret = gdbm_delete ( dbf, key )
```

Dbf is the pointer returned by **gdbm_open**. *Key* is the key data.

The return value is -1 if the item is not present or the requester is a reader. The return value is 0 if there was a successful delete.

The next two routines allow for accessing all items in the database. This access is not key sequential, but it is guaranteed to visit every key in the database once. (The order has to do with the hash values.)

```
key = gdbm_firstkey ( dbf )

nextkey = gdbm_nextkey ( dbf, key )
```

Dbf is the pointer returned by **gdbm_open**. *Key* is the key data.

The return values are both of type **datum**. If the *dptr* element of the return value is NULL, there is no first key or next key. Again notice that *dptr* points to data allocated by **malloc(3C)** and **gdbm** will not free it for you.

These functions were intended to visit the database in read-only algorithms, for instance, to validate the database or similar operations.

File 'visiting' is based on a 'hash table'. *gdbm_delete* re-arranges the hash table to make sure that any collisions in the table do not leave some item 'un-findable'. The original key order is NOT guaranteed to remain unchanged in ALL instances. It is possible that some key will not be visited if a loop like the following is executed:

```
key = gdbm_firstkey ( dbf );
while ( key.dptr ) {
  nextkey = gdbm_nextkey ( dbf, key );
  if ( some condition ) {
    gdbm_delete ( dbf, key );
    free ( key.dptr );
  }
  key = nextkey;
}
```

The following routine should be used very infrequently.

```
ret = gdbm_reorganize ( dbf )
```

If you have had a lot of deletions and would like to shrink the space used by the **gdbm** file, this routine will reorganize the database. **Gdbm** will not shorten the length of a **gdbm** file except by using this reorganization. (Deleted file space will be reused.)

Unless your database was opened with the GDBM_SYNC flag, gdbm does not wait for writes to be flushed to the disk before continuing. The following routine can be used to guarantee that the database is physically written to the disk file.

```
gdbm_sync ( dbf )
```

It will not return until the disk file state is syncronized with the in-memory state of the database.

To convert a **gdbm** error code into English text, use this routine:

```
ret = gdbm_strerror ( errno )
```

Where *errno* is of type *gdbm_error*, usually the global variable *gdbm_errno*. The appropiate phrase is returned.

Gdbm now supports the ability to set certain options on an already open database.

```
ret = gdbm_setopt ( dbf, option, value, size )
```

Where *dbf* is the return value from a previous call to **gdbm_open**, and *option* specifies which option to set. The valid options are currently:

GDBM_CACHESIZE - Set the size of the internal bucket cache. This option may only be set once on each *GDBM_FILE* descriptor, and is set automatically to 100 upon the first access to the database.

GDBM_FASTMODE - Set **fast mode** to either on or off. This allows **fast mode** to be toggled on an already open and active database. *value* (see below) should be set to either TRUE or FALSE. *This option is now obsolete.*

GDBM_SYNCMODE - Turn on or off file system synchronization operations. This setting defaults to off; *value* (see below) should be set to either TRUE or FALSE.

GDBM_CENTFREE - Set **central free block pool** to either on or off. The default is off, which is how previous versions of **Gdbm** handled free blocks. If set, this option causes all subsequent free blocks to be placed in the **global** pool, allowing (in thoery) more file space to be reused more quickly. *value* (see below) should be set to either TRUE or FALSE.
NOTICE: This feature is still under study.

GDBM_COALESCEBLKS - Set **free block merging** to either on or off. The default is off, which is how previous versions of **Gdbm** handled free blocks. If set, this option causes adjacent free blocks to be merged. This can become a CPU expensive process with time, though, especially if used in conjunction with **GDBM_CENTFREE**. *value* (see below) should be set to either TRUE or FALSE.
NOTICE: This feature is still under study.

value is the value to set *option* to, specified as an integer pointer. *size* is the size of the data pointed to by *value*. The return value will be -1 upon failure, or 0 upon success. The global variable *gdbm_errno* will be set upon failure.

For instance, to set a database to use a cache of 10, after opening it with **gdbm_open**, but prior to accessing it in any way, the following code could be used:

```
int value = 10;

ret = gdbm_setopt( dbf, GDBM_CACHESIZE, &value, sizeof(int));
```

If the database was opened with the **GDBM_NOLOCK** flag, the user may wish to perform their own file locking on the database file in order to prevent multiple writers operating on the same file simultaneously.

In order to support this, the *gdbm_fdesc* routine is provided.

```
ret = gdbm_fdesc ( dbf )
```

Where *dbf* is the return value from a previous call to **gdbm_open**. The return value will be the file descriptor of the database.

The following two external variables may be useful:

gdbm_errno is the variable that contains more information about gdbm errors. (gdbm.h has the definitions of the error values and defines gdbm_errno as an external variable.)
gdbm_version is the string containing the version information.

There are a few more things of interest. First, **gdbm** files are not "sparse". You can copy them with the UNIX **cp(1)** command and they will not expand in the copying process. Also, there is a compatibility mode for use with programs that already use UNIX **dbm**. In this compatibility mode, no gdbm file pointer is required by the programmer, and only one file may be opened at a time. All users in compatibility mode are assumed to be writers. If the **gdbm** file is a read only, it will fail as a writer, but will also try to open it as a reader. All returned pointers in datum structures point to data that **gdbm** WILL free. They should be treated as static pointers (as standard UNIX **dbm** does).

LINKING

This library is accessed by specifying *-lgdbm* as the last parameter to the compile line, e.g.:

```
gcc -o prog prog.c -lgdbm
```

If you wish to use the **dbm** or **ndbm** compatibility routines, you must link in the *gdbm_compat* library as well. For example:

```
gcc -o prog proc.c -lgdbm -lgdbm_compat
```

BUGS

SEE ALSO

dbm, ndbm

AUTHOR

You may contact the original author by:
e-mail: phil@cs.wwu.edu
us-mail: Philip A. Nelson
Computer Science Department
Western Washington University
Bellingham, WA 98226

You may contact the current maintainer by:
e-mail: downsj@downsj.com