NAME

GDBM - The GNU database manager. Includes **dbm** and **ndbm** compatibility.

SYNOPSIS

#include <gdbm.h>

extern gdbm_error *gdbm_errno***;**
extern char ****gdbm_version*;**
extern int *gdbm_version[3]***;**
GDBM_FILE gdbm_open (const char ****name*, int** *block_size***,**
int *flags***, int** *mode***,**
void (**fatal_func***)(const char *))***;*
int gdbm_close (GDBM_FILE *dbf***);**
int gdbm_store (GDBM_FILE *dbf***, datum** *key***, datum** *content***, int** *flag***);**
datum gdbm_fetch (GDBM_FILE *dbf***, datum** *key***);**
int gdbm_delete (GDBM_FILE *dbf***, datum** *key***);**
datum gdbm_firstkey (GDBM_FILE *dbf***);**
datum gdbm_nextkey (GDBM_FILE *dbf***, datum** *key***);**
int gdbm_recover (GDBM_FILE *dbf***, gdbm_recovery ****rcvr***, int***flags***);**
int gdbm_reorganize (GDBM_FILE *dbf***);**
int gdbm_sync (GDBM_FILE *dbf***);**
int gdbm_exists (GDBM_FILE *dbf***, datum** *key***);**
const char *gdbm_strerror (gdbm_error *errno***);**
int gdbm_setopt (GDBM_FILE *dbf***, int** *option***, int** *value***, int** *size***);**
int gdbm_fdesc (GDBM_FILE *dbf***);**
int gdbm_count (GDBM_FILE *dbf***, gdbm_count_t ****pcount***);**
int gdbm_bucket_count (GDBM_FILE *dbf***, size_t ****pcount***);**
int gdbm_avail_verify (GDBM_FILE *dbf***);**

Crash Tolerance (see below):
int gdbm_failure_atomic (GDBM_FILE *dbf***, const char ****even***, const char ****odd***);**
int gdbm_latest_snapshot (const char ****even*, const char ****odd***, const char *****result***);**

NOTICE

This manpage is a short description of the **GDBM** library. For a detailed discussion, including examples and usage recommendations, refer to the **GDBM Manual** available in Texinfo format. To access it, run:

```
info gdbm
```

The documentation is also available online at

https://www.gnu.org/software/gdbm/manual

Should any discrepancies occur between this manpage and the **GDBM Manual**, the later shall be considered the authoritative source.

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.

Opening a database

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 gdbm_open (const char ****name*, **int *block_size*,
int *flags*, **int** *mode*,
void (**fatal_func***)(const char *))***;*

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. If the value is less than 512, the file system block size is used instead. The size is adjusted so that the block can hold exact number of directory entries, so that the effective block size can be slightly greater than requested. This adjustment is disabled if the **GDBM_BSEXACT** *flag* is used.

The *flags* parameter is a bitmask, composed of the *access mode* and one or more modifier flags. The *access mode* bit designates the process as a reader or writer and must be one of the following:

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

Additional flags (*modifiers*) can be combined with these values by bitwise **OR**. Not all of them are meaningful with all access modes.

Flags that are valid for any value of access mode are:

GDBM_CLOEXEC
Set the *close-on-exec* flag on the database file descriptor.

GDBM_NOLOCK
Prevents the library from performing any locking on the database file.

GDBM_NOMMAP
Instructs **gdbm_open** to disable the use of **mmap**(2).

GDBM_PREREAD
When mapping GDBM file to memory, read its contents immediately, instead of when needed (*prefault reading*). This can be advantageous if you open a *read-only* database and are going to do a lot of look-ups on it. In this case entire database will be read at once and searches will operate on an in-memory copy. In contrast, **GDBM_PREREAD** should not be used if you open a database (even in read-only mode) only to retrieve a couple of keys.

Finally, never use **GDBM_PREREAD** when opening a database for updates, especially for inserts: this will degrade performance.

This flag has no effect if **GDBM_NOMMAP** is given, or if the operating system does not support prefault reading. It is known to work on Linux and FreeBSD kernels.

GDBM_XVERIFY
Enable additional consistency checks. With this flag, eventual corruptions of the database are discovered when opening it, instead of when a corrupted structure is read during normal operation. However, on large databases, it can slow down the opening process.

The following additional flags are valid when the database is opened for writing (**GDBM_WRITER**, **GDBM_WRCREAT**, or **GDBM_NEWDB**):

GDBM_SYNC
Causes all database operations to be synchronized to the disk.

NOTE: this option entails severe performance degradation and does not necessarily ensure that the resulting database state is consistent, therefore we discourage its use. For a discussion of how to ensure database consistency with minimal performance overhead, see **CRASH TOLERANCE** below.

GDBM_FAST

A reverse of **GDBM_SYNC**: synchronize writes only when needed. This is the default. This flag is provided only for compatibility with previous versions of GDBM.

The following flags can be used together with **GDBM_NEWDB**. They also take effect when used with **GDBM_WRCREAT**, if the requested database file doesn't exist:

GDBM_BSEXACT

If this flag is set and the requested *block_size* value cannot be used, **gdbm_open** will refuse to create the database. In this case it will set the **gdbm_errno** variable to **GDBM_BLOCK_SIZE_ERROR** and return **NULL**.

Without this flag, **gdbm_open** will silently adjust the *block_size* to a usable value, as described above.

GDBM_NUMSYNC

Create new database in *extended database format*, a format best suited for effective crash recovery. For a detailed discussion, see the **CRASH RECOVERY** chapter below.

Mode is the file mode (see **chmod**(2) and **open**(2)). It is used if the file is created.

Fatal_func is a function to be called when **gdbm** if it encounters a fatal error. This parameter is deprecated and must always be **NULL**.

The return value 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. In this case, the reason of the failure can be found in the *gdbm_errno* variable. If the following call returns *true* (non-zero value):

```
gdbm_check_syserr(gdbm_open)
```

the system *errno* variable must be examined in order to obtain more detail about the failure.

GDBM_FILE gdbm_fd_open (int *FD*, **const char ****name*, **int** *block_size*,
int *flags*, **int** *mode*,
void (**fatal_func*)**(const char *))***;*

This is an alternative entry point to **gdbm_open**. *FD* is a valid file descriptor obtained as a result of a call to **open**(2) or **creat**(2). The function opens (or creates) a DBM database this descriptor refers to. The descriptor is not **dup**'ed, and will be closed when the returned **GDBM_FILE** is closed. Use **dup (2)** if that is not desirable.

In case of error, the function behaves like **gdbm_open** and **does not close** *FD*. This can be altered by the following value passed in *flags*:

GDBM_CLOERROR
Close *FD* before exiting on error.

The rest of arguments are the same as for **gdbm_open**.

Calling convention

All **GDBM** functions take as their first parameter the *database handle* (**GDBM_FILE**), returned from **gdbm_open** or **gdbm_fd_open**.

Any value stored in the **GDBM** database is described by *datum*, an aggregate type defined as:

```
typedef struct
{
  char *dptr;
  int   dsize;
} datum;
```

The *dptr* field points to the actual data. Its type is **char *** for historical reasons. Actually it should have been typed **void ***. Programmers are free to store data of arbitrary complexity, both scalar and aggregate, in this field.

The *dsize* field contains the number of bytes stored in **dptr**.

The **datum** type is used to describe both *keys* and *content* (values) in the database. Values of this type can be passed as arguments or returned from **GDBM** function calls.

GDBM functions that return **datum** indicate failure by setting its *dptr* field to **NULL**.

Functions returning integer value, indicate success by returning 0 and failure by returning a non-zero value (the only exception to this rule is **gdbm_exists**, for which the return value is reversed).

If the returned value indicates failure, the **gdbm_errno** variable contains an integer value indicating what went wrong. A similar value is associated with the *dbf* handle and can be accessed using the **gdbm_last_errno** function. Immediately after return from a function, both values are exactly equal. Subsequent **GDBM** calls with another *dbf* as argument may alter the value of the global **gdbm_errno**, but the value returned by **gdbm_last_errno** will always indicate the most recent code of an error that occurred for *that particular database*. Programmers are encouraged to use such per-database error codes.

Sometimes the actual reason of the failure can be clarified by examining the system **errno** value. To make sure its value is meaningful for a given **GDBM** error code, use the **gdbm_check_syserr** function. The function takes error code as argument and returns 1 if the **errno** is meaningful for that error, or 0 if it is irrelevant.

Similarly to **gdbm_errno**, the latest **errno** value associated with a particular database can be obtained using the **gdbm_last_syserr** function.

The **gdbm_clear_error** clears the error indicator (both **GDBM** and system error codes) associated with a database handle.

Some critical errors leave the database in a *structurally inconsistent state*. If that happens, all subsequent **GDBM** calls accessing that database will fail with the **GDBM** error code of **GDBM_NEED_RECOVERY** (a special function **gdbm_needs_recovery** is also provided, which returns true if the database handle given as its argument is structurally inconsistent). To return such databases to consistent state, use the **gdbm_recover** function (see below).

The **GDBM_NEED_RECOVERY** error cannot be cleared using **gdbm_clear_error**.

Error functions

This section describes the error handling functions outlined above.

gdbm_error gdbm_last_errno (GDBM_FILE *dbf***)**

Returns the error code of the most recent failure encountered when operating on *dbf*.

int gdbm_last_syserr (GDBM_FILE *dbf***)**

Returns the value of the system **errno** variable associated with the most recent failure that occurred on *dbf*.

Notice that not all **gdbm_error** codes have a relevant system error code. Use the following function to determine if a given code has.

int gdbm_check_syserr (gdbm_error *err***)**

Returns **1**, if system **errno** value should be checked to get more info on the error described by GDBM code *err*.

void gdbm_clear_error (GDBM_FILE *dbf***)**

Clears the error state for the database *dbf*. This function is called implicitly upon entry to any GDBM function that operates on **GDBM_FILE**.

The **GDBM_NEED_RECOVERY** error cannot be cleared.

int gdbm_needs_recovery (GDBM_FILE *dbf***)**

Return **1** if the database file *dbf* is in inconsistent state and needs recovery.

const char *gdbm_strerror (gdbm_error *err***)**

Returns a textual description of the error code *err*.

const char *gdbm_db_strerror (GDBM_FILE *dbf***)**

Returns a textual description of the recent error in database *dbf*. This description includes the system **errno** value, if relevant.

Closing the database

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

int gdbm_close (GDBM_FILE *dbf***);**

Database lookups

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

If the *key* is found within the database, the return value will be *true* (**1**). If nothing appropriate is found, *false* (**0**) is returned and **gdbm_errno** set to **GDBM_NO_ERROR**.

On error, returns 0 and sets **gdbm_errno**.

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

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

If the *dptr* element of the return value is **NULL**, the **gdbm_errno** variable should be examined. The value of **GDBM_ITEM_NOT_FOUND** means no data was found for that *key*. Other value means an error occurred.

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

Iterating over the database

The following two routines allow for iterating over all items in the database. Such iteration is not key sequential, but it is guaranteed to visit every key in the database exactly once. (The order has to do with the hash values.)

datum gdbm_firstkey (GDBM_FILE *dbf***);**

Returns first key in the database.

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

Given a *key*, returns the database key that follows it. End of iteration is marked by returning *datum* with *dptr* field set to **NULL** and setting the **gdbm_errno** value to **GDBM_ITEM_NOT_FOUND**.

After successful return from both functions, *dptr* points to data allocated by **malloc**(3). It is the caller responsibility to free the data when no longer needed.

A typical iteration loop looks like:

```
datum key, nextkey, content;
key = gdbm_firstkey (dbf);
while (key.dptr)
  {
    content = gdbm_fetch (dbf, key);
    /* Do something with key and/or content */
    nextkey = gdbm_nextkey (dbf, key);
    free (key.dptr);
    key = nextkey;
  }
```

These functions are intended to visit the database in read-only algorithms. Avoid any database modifications within the iteration loop. File *visiting* is based on a hash table. The **gdbm_delete** and, in most cases, **gdbm_store**, functions rearrange the hash table to make sure that any collisions in the table do not leave some item 'un-findable'. Thus, a call to either of these functions changes the order in which the keys are ordered. Therefore, these functions should not be used when iterating over all the keys in the database. For example, the following loop is wrong: it is possible that some keys will not be visited or will be visited twice if it 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;
  }
```

Updating the database

int gdbm_store (GDBM_FILE *dbf***, datum** *key***, datum** *content***, int** *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.

The function returns 0 on success and -1 on failure. If the key already exists in the database and the *flag* is **GDBM_INSERT**, the function does not modify the database. It sets **gdbm_errno** to **GDBM_CANNOT_REPLACE** and returns 1.

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

Looks up and deletes the given *key* from the database *dbf*.

The return value is 0 if there was a successful delete or -1 on error. In the latter case, the **gdbm_errno** value **GDBM_ITEM_NOT_FOUND** indicates that the key is not present in the database. Other **gdbm_errno** values indicate failure.

Recovering structural consistency

If a function leaves the database in structurally inconsistent state, it can be recovered using the **gdbm_recover** function.

int gdbm_recover (GDBM_FILE *dbf***, gdbm_recovery *** *rcvr***, int** *flags***)**

Check the database file DBF and fix eventual inconsistencies. The *rcvr* argument can be used both to control the recovery and to return additional statistics about the process, as indicated by *flags*. For a detailed discussion of these arguments and their usage, see the **GDBM Manual**, chapter

Recovering structural consistency.

You can pass **NULL** as *rcvr* and **0** as *flags*, if no such control is needed.

By default, this function first checks the database for inconsistencies and attempts recovery only if some were found. The special *flags* bit **GDBM_RCVR_FORCE** instructs **gdbm_recovery** to skip this check and to perform database recovery unconditionally.

Export and import

GDBM database files can be exported (dumped) to so called *flat files* or imported (loaded) from them. A flat file contains exactly the same data as the original database, but it cannot be used for searches or updates. Its purpose is to keep the data from the database for restoring it when the need arrives. As such, flat files are used for backup purposes, and for sending databases over the wire.

As of **GDBM** version 1.21, there are two flat file formats. The **ASCII** file format encodes all data in Base64 and stores not only key/data pairs, but also the original database file metadata, such as file name, mode and ownership. Files in this format can be sent without additional encapsulation over transmission channels that normally allow only ASCII data, such as, e.g. SMTP. Due to additional metadata they allow for restoring an exact copy of the database, including file ownership and privileges, which is especially important if the database in question contained some security-related data. This is the preferred format.

Another flat file format is the **binary** format. It stores only key/data pairs and does not keep information about the database file itself. It cannot be used to copy databases between different architectures. The binary format was introduced in **GDBM** version 1.9.1 and is retained mainly for backward compatibility.

The following functions are used to export or import **GDBM** database files.

int gdbm_dump (GDBM_FILE *dbf***, const char ****filename***,**
int *format***, int** *open_flag***, int** *mode***)** Dumps the database file *dbf* to the file *filename* in requested *format*. Allowed values for *format* are: **GDBM_DUMP_FMT_ASCII**, to create an ASCII dump file, and **GDBM_DUMP_FMT_BINARY**, to create a binary dump.

The value of *open_flag* tells **gdbm_dump** what to do if *filename* already exists. If it is **GDBM_NEWDB**, the function will create a new output file, replacing it if it already exists. If its value is **GDBM_WRCREAT**, the file will be created if it does not exist. If it does exist, **gdbm_dump** will return error.

The file mode to use when creating the output file is defined by the *mode* parameter. Its meaning is the same as for **open**(2).

int gdbm_load (GDBM_FILE ****pdbf*, const char ****filename***,**
int *flag***, int** *meta_mask***, unsigned long ****errline***)** Loads data from the dump file *filename* into the database pointed to by *pdbf*. If *pdbf* is **NULL**, the function will try to create a new database. On success, the new **GDBM_FILE** object will be stored in the memory location pointed to by *pdbf*. If the dump file carries no information about the original database file name, the function will set **gdbm_errno** to **GDBM_NO_DBNAME** and return -1, indicating failure.

Otherwise, if *pdbf* points to an already open **GDBM_FILE**, the function will load data from *filename* into that database.

The *flag* parameter controls the function behavior if a key from the dump file already exists in the database. See the **gdbm_store** function for its possible values.

The *meta_mask* parameter can be used to disable restoring certain bits of file's meta-data from the information in the input dump file. It is a binary OR of zero or more of the following:

GDBM_META_MASK_MODE
Do not restore file mode.

GDBM_META_MASK_OWNER
Do not restore file owner.

Other functions

int gdbm_reorganize (GDBM_FILE *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.

int gdbm_sync (GDBM_FILE *dbf***);**
Synchronizes the changes in *dbf* with its disk file.

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

int gdbm_setopt (GDBM_FILE *dbf***, int** *option***, void ****value***, int** *size***);**
Query or change some parameter of an already opened database. The *option* argument defines what parameter to set or retrieve. If the *set* operation is requested, *value* points to the new value. Its actual data type depends on *option*. If the *get* operation is requested, *value* points to a memory region where to store the return value. In both cases, *size* contains the actual size of the memory pointed to by *value*.

Possible values of *option* are:

GDBM_SETCACHESIZE
GDBM_CACHESIZE

Set the size of the internal bucket cache. The *value* should point to a **size_t** holding the desired cache size, or the constant **GDBM_CACHE_AUTO**, to select the best cache size automatically.

By default, a newly open database is configured to adapt the cache size to the number of index buckets in the database file. This provides for the best performance.

Use this option if you wish to limit the memory usage at the expense of performance. If you chose to do so, please bear in mind that cache becomes effective when its size is greater then 2/3 of the number of index bucket counts in the database. The best performance results are achieved when cache size equals the number of buckets.

GDBM_GETCACHESIZE

Return the size of the internal bucket cache. The *value* should point to a **size_t** variable, where the size will be stored.

GDBM_GETFLAGS

Return the flags describing current state of the database. The *value* should point to an **int** variable where to store the flags. On success, its value will be similar to the flags used when opening the database, except that it will reflect the current state (which may have been altered by another calls to **gdbm_setopt**).

GDBM_FASTMODE

Enable or disable the *fast writes mode*, similar to the **GDBM_FAST** option to **gdbm_open**.

This option is retained for compatibility with previous versions of **GDBM**.

GDBM_SETSYNCMODE
GDBM_SYNCMODE

Turn on or off immediate disk synchronization after updates. The *value* should point to an integer: 1 to turn synchronization on, and 0 to turn it off.

NOTE: setting this option entails severe performance degradation and does not necessarily ensure that the resulting database state is consistent, therefore we discourage its use. For a discussion of how to ensure database consistency with minimal performance overhead, see **CRASH TOLERANCE** below.

GDBM_GETSYNCMODE

Return the current synchronization status. The *value* should point to an **int** where the status will be stored.

GDBM_SETCENTFREE
GDBM_CENTFREE

Enable or disable central free block pool. 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 theory) more file space to be reused more quickly. The *value* should point to an integer: **TRUE** to turn central block pool on, and **FALSE** to turn it off.

The **GDBM_CENTFREE** alias is provided for compatibility with earlier versions.

GDBM_SETCOALESCEBLKS
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**. The *value* should point to an integer: **TRUE** to turn free block merging on, and **FALSE** to turn it off.

GDBM_GETCOALESCEBLKS

Return the current status of free block merging. The *value* should point to an **int** where the status will be stored.

GDBM_SETMAXMAPSIZE

Sets maximum size of a memory mapped region. The *value* should point to a value of type **size_t**, **unsigned long** or **unsigned**. The actual value is rounded to the nearest page boundary (the page size is obtained from **sysconf(_SC_PAGESIZE)**).

GDBM_GETMAXMAPSIZE

Return the maximum size of a memory mapped region. The *value* should point to a value of type **size_t** where to return the data.

GDBM_SETMMAP

Enable or disable memory mapping mode. The *value* should point to an integer: **TRUE** to enable memory mapping or **FALSE** to disable it.

GDBM_GETMMAP

Check whether memory mapping is enabled. The *value* should point to an integer where to return the status.

GDBM_GETDBNAME

Return the name of the database disk file. The *value* should point to a variable of type **char****. A pointer to the newly allocated copy of the file name will be placed there. The caller is responsible for freeing this memory when no longer needed.

GDBM_GETBLOCKSIZE

Return the block size in bytes. The *value* should point to **int**.

int gdbm_fdesc (GDBM_FILE *dbf***);**

Returns the file descriptor of the database *dbf*.

CRASH TOLERANCE

By default **GNU dbm** does not protect the integrity of its databases from corruption or destruction due to failures such as power outages, operating system kernel panics, or application process crashes. Such failures could damage or destroy the underlying database.

Starting with release 1.21 **GNU dbm** includes a mechanism that, if used correctly, enables post-crash recovery to a consistent state of the underlying database. This mechanism requires OS and filesystem support and must be requested when **gdbm** is compiled. The crash-tolerance mechanism is a "pure opt-in" feature, in the sense that it has no effects whatsoever except on those applications that explicitly request it. For details, see the chapter **Crash Tolerance** in the **GDBM manual**.

GLOBAL VARIABLES

gdbm_error gdbm_errno

This variable contains code of the most recent error that occurred. Note, that it is not C variable in the proper sense: you can use its value, assign any value to it, but taking its address will result in syntax error. It is a per-thread memory location.

const char *gdbm_version

A string containing the library version number and build date.

int const gdbm_version_number[3]

This variable contains library version numbers: major, minor, and patchlevel.

VERSIONING

The version information is kept in two places. The version of the library is kept in the **gdbm_version_number** variable, described above. Additionally, the header file **gdbm.h** defines the following macros:

GDBM_VERSION_MAJOR
Major version number.

GDBM_VERSION_MINOR
Minor version number.

GDBM_VERSION_PATCH
Patchlevel number. **0** means no patchlevel.

You can use this to compare whether your header file corresponds to the library the program is linked with.

The following function can be used to compare two version numbers:

int gdbm_version_cmp (int const *a***[3], int const** *b***[3])**
Compare two version numbers formatted as **gdbm_version_number**. Return negative number if **a** is older than **b**, positive number if **a** is newer than **b**, and 0 if they are equal.

ERROR CODES

GDBM_NO_ERROR
No error occurred.

GDBM_MALLOC_ERROR
Memory allocation failed.

GDBM_BLOCK_SIZE_ERROR
This error is set by the **gdbm_open** function, if the value of its *block_size* argument is incorrect and the **GDBM_BSEXACT** flag is set.

GDBM_FILE_OPEN_ERROR
The library was not able to open a disk file. This can be set by **gdbm_open**, **gdbm_fd_open**, **gdbm_dump** and **gdbm_load** functions.

Inspect the value of the system **errno** variable to get more detailed diagnostics.

GDBM_FILE_WRITE_ERROR
Writing to a disk file failed. This can be set by **gdbm_open**, **gdbm_fd_open**, **gdbm_dump** and **gdbm_load** functions.

Inspect the value of the system **errno** variable to get more detailed diagnostics.

GDBM_FILE_SEEK_ERROR
Positioning in a disk file failed. This can be set by **gdbm_open** function.

Inspect the value of the system **errno** variable to get a more detailed diagnostics.

GDBM_FILE_READ_ERROR
Reading from a disk file failed. This can be set by **gdbm_open**, **gdbm_dump** and **gdbm_load** functions.

Inspect the value of the system **errno** variable to get a more detailed diagnostics.

GDBM_BAD_MAGIC_NUMBER
The file given as argument to **gdbm_open** function is not a valid **gdbm** file: it has a wrong magic number.

GDBM_EMPTY_DATABASE
The file given as argument to **gdbm_open** function is not a valid **gdbm** file: it has zero length. This error is returned unless the *flags* argument has **GDBM_NEWDB** bit set.

GDBM_CANT_BE_READER
This error code is set by the **gdbm_open** function if it is not able to lock file when called in **GDBM_READER** mode.

GDBM_CANT_BE_WRITER
This error code is set by the **gdbm_open** function if it is not able to lock file when called in writer mode.

GDBM_READER_CANT_DELETE
Set by the **gdbm_delete**, if it attempted to operate on a database that is open in read-only mode.

GDBM_READER_CANT_STORE
Set by the **gdbm_store** if it attempted to operate on a database that is open in read-only mode.

GDBM_READER_CANT_REORGANIZE
Set by the **gdbm_reorganize** if it attempted to operate on a database that is open in read-only mode.

GDBM_ITEM_NOT_FOUND
Requested item was not found. This error is set by **gdbm_delete** and **gdbm_fetch** when the

requested key value is not found in the database.

GDBM_REORGANIZE_FAILED
The **gdbm_reorganize** function is not able to create a temporary database.

GDBM_CANNOT_REPLACE
Cannot replace existing item. This error is set by the **gdbm_store** if the requested key value is found in the database and the *flag* parameter is not **GDBM_REPLACE**.

GDBM_MALFORMED_DATA
Input data was malformed in some way. When returned by **gdbm_load**, this means that the input file was not a valid **gdbm** dump file. When returned by **gdbm_store**, this means that either *key* or *content* parameter had its **dptr** field set to **NULL**.

The **GDBM_ILLEGAL_DATA** is an alias for this error code, maintained for backward compatibility.

GDBM_OPT_ALREADY_SET
Requested option can be set only once and was already set. As of version 1.21, this error code is no longer used. In prior versions it could have been returned by the **gdbm_setopt** function when setting the **GDBM_CACHESIZE** value.

GDBM_OPT_BADVAL
The *option* argument is not valid or the *value* argument points to an invalid value in a call to **gdbm_setopt** function.

GDBM_OPT_ILLEGAL is an alias for this error code, maintained for backward compatibility. Modern applications should not use it.

GDBM_BYTE_SWAPPED
The **gdbm_open** function attempts to open a database which is created on a machine with different byte ordering.

GDBM_BAD_FILE_OFFSET
The **gdbm_open** function sets this error code if the file it tries to open has a wrong magic number.

GDBM_BAD_OPEN_FLAGS
Set by the **gdbm_dump** function if supplied an invalid *flags* argument.

GDBM_FILE_STAT_ERROR

Getting information about a disk file failed. The system **errno** will give more details about the error.

This error can be set by the following functions: **gdbm_open**, **gdbm_reorganize**.

GDBM_FILE_EOF

End of file was encountered where more data was expected to be present. This error can occur when fetching data from the database and usually means that the database is truncated or otherwise corrupted.

This error can be set by any GDBM function that does I/O. Some of these functions are: **gdbm_delete**, **gdbm_exists**, **gdbm_fetch**, **gdbm_export**, **gdbm_import**, **gdbm_reorganize**, **gdbm_firstkey**, **gdbm_nextkey**, **gdbm_store**.

GDBM_NO_DBNAME

Output database name is not specified. This error code is set by **gdbm_load** if the first argument points to **NULL** and the input file does not specify the database name.

GDBM_ERR_FILE_OWNER

This error code is set by **gdbm_load** if it is unable to restore the database file owner. It is a mild error condition, meaning that the data have been restored successfully, only changing the target file owner failed. Inspect the system **errno** variable to get a more detailed diagnostics.

GDBM_ERR_FILE_MODE

This error code is set by **gdbm_load** if it is unable to restore database file mode. It is a mild error condition, meaning that the data have been restored successfully, only changing the target file owner failed. Inspect the system **errno** variable to get a more detailed diagnostics.

GDBM_NEED_RECOVERY

Database is in inconsistent state and needs recovery. Call **gdbm_recover** if you get this error.

GDBM_BACKUP_FAILED

The GDBM engine is unable to create backup copy of the file.

GDBM_DIR_OVERFLOW

Bucket directory would overflow the size limit during an attempt to split hash bucket. This error can occur while storing a new key.

GDBM_BAD_BUCKET

Invalid index bucket is encountered in the database. Database recovery is needed.

GDBM_BAD_HEADER

This error is set by **gdbm_open** and **gdbm_fd_open**, if the first block read from the database file does not contain a valid GDBM header.

GDBM_BAD_AVAIL

The available space stack is invalid. This error can be set by **gdbm_open** and **gdbm_fd_open**, if the extended database verification was requested (**GDBM_XVERIFY**). It is also set by the **gdbm_avail_verify** function.

The database needs recovery.

GDBM_BAD_HASH_TABLE

Hash table in a bucket is invalid. This error can be set by the following functions: **gdbm_delete**, **gdbm_exists**, **gdbm_fetch**, **gdbm_firstkey**, **gdbm_nextkey**, and **gdbm_store**.

The database needs recovery.

GDBM_BAD_DIR_ENTRY

Bad directory entry found in the bucket. The database recovery is needed.

GDBM_FILE_CLOSE_ERROR

The **gdbm_close** function was unable to close the database file descriptor. The system **errno** variable contains the corresponding error code.

GDBM_FILE_SYNC_ERROR

Cached content couldn't be synchronized to disk. Examine the **errno** variable to get more info,

Database recovery is needed.

GDBM_FILE_TRUNCATE_ERROR

File cannot be truncated. Examine the **errno** variable to get more info.

This error is set by **gdbm_open** and **gdbm_fd_open** when called with the **GDBM_NEWDB** flag.

GDBM_BUCKET_CACHE_CORRUPTED

The bucket cache structure is corrupted. Database recovery is needed.

GDBM_BAD_HASH_ENTRY

This error is set during sequential access (@pxref{Sequential}), if the next hash table entry does not contain the expected key. This means that the bucket is malformed or corrupted and the database needs recovery.

GDBM_ERR_SNAPSHOT_CLONE
Set by the **gdbm_failure_atomic** function if it was unable to clone the database file into a snapshot. Inspect the system **errno** variable for the underlying cause of the error. If **errno** is **EINVAL** or **ENOSYS**, crash tolerance settings will be removed from the database.

GDBM_ERR_REALPATH
Set by the **gdbm_failure_atomic** function if the call to **realpath** function failed. **realpath** is used to determine actual path names of the snapshot files. Examine the system **errno** variable for details.

GDBM_ERR_USAGE
Function usage error. That includes invalid argument values, and the like.

DBM COMPATIBILITY ROUTINES

GDBM includes a compatibility library **libgdbm_compat**, for use with programs that expect traditional UNIX **dbm** or **ndbm** interfaces, such as, e.g. **Sendmail**. The library is optional and thus may be absent in some binary distributions.

As the detailed discussion of the compatibility API is beyond the scope of this document, below we provide only a short reference. For details, see the **GDBM Manual**, chapter **Compatibility with standard dbm and ndbm**.

DBM compatibility routines

In **dbm** compatibility mode only one file may be opened at a time. All users are assumed to be writers. If the database file is read only, it will fail as a writer, but will be opened as a reader. All returned pointers in datum structures point to data that the compatibility library **will free**. They should be treated as static pointers (as standard UNIX **dbm** does).

The following interfaces are provided:

#include <dbm.h>

int dbminit (const char ***_name_);**
int store (datum _key_**, datum** _content_**);**
datum fetch (datum _key_**);**
int delete (datum _key_**);**
datum firstkey (void);
datum nextkey (datum _key_**);**
int dbmclose (void);

NDBM Compatibility routines:

In this mode, multiple databases can be opened. Each database is identified by a handle of type **DBM ***. As in the original **NDBM**, all returned pointers in datum structures point to data that will be freed by the compatibility library. They should be treated as static pointers.

The following interfaces are provided:

#include <ndbm.h>

DBM *dbm_open (const char ****name*, **int *flags*, **int** *mode***);**
void dbm_close (DBM ****file*);**
datum dbm_fetch (DBM ****file*, **datum *key***);**
int dbm_store (DBM ****file*, **datum *key*, **datum** *content*, **int** *flags***);**
int dbm_delete (DBM ****file*, **datum *key***);**
datum dbm_firstkey (DBM ****file*);**
datum dbm_nextkey (DBM ****file*, **datum *key***);**
int dbm_error (DBM ****file*);**
int dbm_clearerr (DBM ****file*);**
int dbm_pagfno (DBM ****file*);**
int dbm_dirfno (DBM ****file*);**
int dbm_rdonly (DBM ****file*);**

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
```

BUG REPORTS

Send bug reports to <bug-gdbm@gnu.org>.

SEE ALSO

gdbm_dump(1), **gdbm_load**(1), **gdbmtool**(1).

AUTHORS

by Philip A. Nelson, Jason Downs and Sergey Poznyakoff; crash tolerance by Terence Kelly.

COPYRIGHT

CONTACTS

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 maintainers by:
  e-mail: downsj@downsj.com
and
  e-mail: gray@gnu.org

For questions and feedback regarding crash tolerance, you may contact Terence Kelly at:
  e-mail: tpkelly @ { acm.org, cs.princeton.edu, eecs.umich.edu }