EDB Manual

An Emacs Database

by Michael Ernst

This documentation describes version 1.18 of EDB, dated June 16, 1993. The documentation was last modified on Jun 19 1993.

Short Contents

Table of Contents

1 Introduction

EDB is a database program for GNU Emacs. It permits users to manipulate structured (or not-so-structured) data within Emacs and provides many of the usual database features, including:

- Flexible, customizable file layouts. Data may contain any character, including those used to delimit fields and records. Files read and written by the database may have arbitrary formats.
- Typed fields (e.g. integer, date, string); fields may also be subject to additional constraints (prime number, date before today, string that appears in some other record, etc.).
- Arbitrary data display formats for viewing records. Multiple display formats can be open on a database simultaneously, viewing the same or different records. The data display format can be automatically chosen based on the record's field values.
- Selective display of only those records of interest; others become temporarily user-invisible.
- Standard GNU Emacs editing commands, which work only within data fields and not on the surrounding text.
- Database summaries, which show in a single buffer one or more lines of information about each record.
- Sorting, with an easy-to-use a graphical interface for defining the sorting criteria; most sorting orders you would care about are easy to specify, but arbitrary ones are also permitted.
- Merging and reconciliation of databases.
- Reports generated from database information.
- Highly customizable via the underlying programming language, Emacs Lisp; many hooks and useful variables are provided to make this even easier.
- Documented by an 80-page manual.

EDB is more ambitious—and therefore more complex—than its forerunners (such as Forms Mode by Johan Vromans <jv@mh.nl>). While other packages don't provide as much functionality as EDB, they may be more appropriate for simple needs.

1.1 Organization of this manual

This manual contains two major parts. The first part describes how to use EDB to manipulate an existing database, and the second part describes how to design a new database.

The first part—which could be called the EDB User Manual—first presents basic commands such as starting up the database, adding, deleting, and modifying records, and searching; then it describes features for the more advanced user, such as sorting, displaying record summaries, marking or ignoring certain records, and producing reports.

The second part—which could be called the EDB Database Designer Manual—describes the three forms that database information can take: when being manipulated by EDB, when stored on disk, and when displayed on the screen. Separate chapters discuss specifying each of these representations. The manual then goes on to discuss customization hooks and explains some of the lower-level implementation details that an advanced designer may need to know.

1.2 Invoking EDB

You need three files to run EDB: a data file, a format file, and an auxiliary file. The data file (usual suffix '`.dat`') contains the information that makes up the database. The format file (usual suffix '`.fmt`') specifies how the fields of a particular record appear in the data display buffer, where the user may view or edit one record at a time. The auxiliary file (usual suffix '`.dba`') contains additional information about the database, such as the number of fields in each record, the layout of the data file (including what characters or strings serve as field and record separators), customizations, etc. A fourth type of file is the report format file, which is a format file used in generating reports printed on the screen, to a file, or to a printer; see Chapter 10 [Reports], page 37.

For examples of data, format, and auxiliary files, a description of how to create your own, and pointers to even more information, see Section 12.1 [Creating a new database], page 42, or see the examples provided with EDB (see Chapter 2 [Installation], page 6).

You can combine the format and auxiliary files (by placing the additional information at the end of the format file, in the "Local variables" section), but it is simpler to consider the two files separately. There may be many different ways to lay out a record on the screen, so a database could have many different format files; for the time being we will concentrate on the format file which is used first, which is called the primary format file, even though it might not be the one that is used most often.

When invoking the database, you typically only need to name the data file; the names of the others will be inferred from its name (see Section 16.1 [Auxiliary files], page 75) or may be mentioned explicitly by it. The `db-find-file` command starts up the database:

`M-x db-find-file`

Read a database from *database-file*; prompts when called interactively. If the database file doesn't specify a format and the format file can't be inferred from *database-file*, the user is prompted for it too. The user is always prompted for the format if prefix arg *prompt-for-format* is non-`nil`. If the database is already read in and *prompt-for-format* is `nil`, the existing database buffer is merely selected. When called non-interactively, argument *prompt-for-format* may be a string, the name of a format file to use.

You can also arrange that `find-file` automatically invokes EDB when called on a database file; see Chapter 2 [Installation], page 6 for details.

The usual save-file and write-file keystrokes are rebound in all database modes.

`C-x C-s` (`db-save-database`) Save the database to disk in the default save file. Any changes to the current record are processed first. The default save file is the file it was last saved to or read from. If optional arg *query* is specified, the user is asked first. Optional second arg *quietly* suppresses messages regarding the filename.

`C-x C-w` (`db-write-database-file`) Save the database to disk in file *filename*; it becomes the default save file. Any changes to the current record are processed first. If *filename* is not specified, the user is prompted for it. Optional second arg *quietly* suppresses messages regarding the filename.

1.3 Example EDB session

This section describes some of the most frequently used EDB commands. All of the commands used here are more fully documented elsewhere in the manual. For a very brief introduction to database mode, its submodes (view mode and edit mode), the data display buffer, EDB concepts, and more, see Chapter 3 [Database mode], page 16.

`M-x db-find-file` RET `forms-demo2.dat` RET

Load the database. After calling `db-find-file`, you may be asked for the name of the format file if EDB can't infer it from the name or contents of the data file. See Section 1.2 [Invoking EDB], page 2. When the database finishes loading, the data display buffer is visible and EDB is in database view mode (see Chapter 4 [Database view mode], page 18).

`n`
`p` Go to the next or previous record (see Section 4.1 [Moving around in the database], page 18). The data display buffer is in view mode, which is read-only and does not permit editing (see Chapter 4 [Database view mode], page 18).

`TAB`
`C-n`
: Go to the first field and switch to database edit mode. See Section 4.2 [Changing to edit mode], page 19, and Chapter 5 [Database edit mode], page 22. Once you are on a field, printing characters insert themselves and the other usual editing commands will work as well. `TAB` moves to the next field, and `C-n` moves to a field on the next line (or to the next line of this field, if it spans multiple lines). See Section 5.4 [Moving from field to field], page 23, and Section 5.5 [Movement within a field], page 23.

`M-TAB`
`C-p`
: Like `TAB` and `C-n`, but move backward by fields or lines.

`M-s`
: Search for a value in the current field. See Chapter 6 [Searching], page 25.

`C-c C-c`
: Return to view mode.

`h`
`D`
: See a summary of all the records of the database; `h` stands for "headers" and D stands for "Directory". See Chapter 8 [Summary mode], page 32. You can move around in the summary buffer using ordinary movement commands, and the record under point will be displayed in the data display buffer. Use `v` or `e` to return to the data display buffer; `v` puts you in view mode and `e` puts you in edit mode.

`M-x db-sort RET`
: Invoke the database sort interface, which permits easy specification of how records should be sorted, and then perform the sort. Type `C-h m` or `?` for help while in the database sort interface (or anywhere in EDB). See Chapter 7 [Sorting], page 28.

`C-x C-s`
: Save to disk any changes you have made to the database.

`C-h m`
: Display a list of commands. This command works in all Emacs modes (and in paricular, in all EDB modes).

`q`
`x`
: Quit EDB. `q` just buries the buffer; `x` also offers to save if changes have been made.

EDB provides many more commands than these; see this manual's table of contents or index (see [Concept Index], page 102) to find the topic that interests you.

1.4 Terminology

A database is a collection of records, each of which is comprised of fields. A record's fields are usually all related to some central object or concept; for instance, they might describe various information about a particular person such as name, address, and phone number. All records of a database have the same structure (they contain the same fields), though typically different records will have different information in the fields.

EDB permits database records to be viewed, edited, and manipulated in a structured way.

2 Installation

FTP the files from '`theory.lcs.mit.edu:/pub/emacs/edb/`'; the package is in a single compressed tar file '`edb.tar.Z`', and the individual files can also be obtained from the '`code`' subdirectory. Install the files in your Emacs load path (probably in a directory of their own). You can add a directory to your Emacs lisp load path by putting something similar to the following in your '`.emacs`' file:

```
(setq load-path (cons (expand-file-name "~/emacs/edb") load-path))
```

Finally, cause EDB to be autoloaded by putting the following in your '`.emacs`' file:

```
(autoload 'db-find-file "database" "EDB database package" t)
(autoload 'load-database "database" "EDB database package" t)
(autoload 'byte-compile-database "database" "EDB database package" t)
```

Now, when you start up Emacs, you will already be able to execute `db-find-file`; EDB will be loaded automatically. (You may also wish to autoload function `edb-update`; see Section 2.1 [EDB is in beta test], page 7.) See below if you wish to byte-compile the EDB sources.

Here is one way to arrange to automatically run EDB when you read a database file via the usual `find-file` command (ordinarily bound to `C-x C-f`), whether or not you choose to autoload EDB. This only works on databases which have been stored in EDB internal file layout (see Section 14.1 [Internal file layout], page 51).

```
(setq find-file-hooks (cons 'after-find-file-edb find-file-hooks))
(defun after-find-file-edb ()
  "If this is a database file in EDB internal file layout, run EDB."
  ;; When this is called, we are at the beginning of the buffer.
  (if (looking-at ";; Database file written by EDB")
      (progn
        (require 'database)
        (db-this-buffer)
        ;; db-this-buffer kills the current buffer; and an error results
        ;; when Emacs tries to switch back to it.  find-file-noselect
        ;; uses the buf variable to hold the new buffer.
        (setq buf (buffer-name (current-buffer))))))
```

Naturally, running EDB will do you little good without a database to manipulate; for information about creating a new database or using an existing one (EDB can handle nearly any file layout imaginable and many that aren't), see Chapter 12 [Designing a database], page 42. You may also want to use existing databases as guides, or to help familiarize yourself with EDB. A number of examples can be found in the compressed tar file '`examples.tar.Z`' or in the '`examples/`' subdirectory of '`theory.lcs.mit.edu:/pub/emacs/edb/`' (the two locations contain the same examples). You can test out function `after-find-file-edb` by performing `find-file` on '`forms-demo2-int.dat`'.

It is strongly recommended that you run EDB byte-compiled, as otherwise it is very sluggish. To byte-compile EDB, type `M-x byte-compile-database RET`. (You may need to load EDB first, by typing `M-x load-library RET database RET` or `M-x load-file RET database.el RET`, in order to define this function.) If you perform the byte-compilation yourself rather than using the `byte-compile-database` function, you **must** fully load the code before compiling it, by typing `C-u M-x load-database RET`. For more details, See Section 2.2.4.3 [Compiling EDB], page 11.

The texinfo documentation must be processed using release 2 of texinfo (which is available via anonymous ftp from '`prep.ai.mit.edu`', directory '`pub/gnu`'), but the resulting Info files can be read using any Info reader. If you don't have texinfo version 2, you can get the EDB documentation pre-processed in info format from '`theory.lcs.mit.edu:/pub/emacs/edb/`', files '`database.info`' and '`database.info-[123456]`' (seven files in all). Similarly, that directory contains ready-to-print versions of the manual (file '`database.dvi`' or '`database.ps`').

2.1 EDB is in beta test

EDB is a project I undertook because the existing tools for manipulating structured information in Emacs were lacking features I considered important. EDB now meets my needs—and those of a number of other users—but it still contains some important lacunae. I will fill these in as I can. I encourage users of EDB to mention which ones are most important to them, so that I can decide in what order to undertake the many projects on my "to do" list. I encourage you even more strongly to contribute code for features currently lacking—then you'll be sure of its inclusion in EDB, and you'll be helping others as well.

EDB has been in use since the summer of 1991, but it has not been exhaustively and systematically tested, so it may still contain bugs. Please send me bug reports and (if possible) bug fixes, and I'll correct the problems in the next release.

The mailing list '`edb-list@theory.lcs.mit.edu`' is intended for discussions relating to EDB: trading extensions, sharing experiences, asking questions, reporting bugs and bug fixes, and distributing updates to EDB. Send requests to be added to (or removed from) the list to '`edb-list-request@theory.lcs.mit.edu`'.

You can make it easy to install updates of EDB by putting something like the following in your '.emacs' file:

```
(setq edb-directory "~/emacs/edb")
(autoload 'edb-update "database" "EDB database package" t)
```

Then, when you receive a message containing diffs for a new version of EDB, you only have to type `M-x edb-update RET` in order to install them.

`M-x edb-update`
: Install the EDB update found in the current buffer after point. EDB is assumed to be in the directory specified by `edb-directory`. If you have trouble with this command, it is likely that your version of EDB is not exactly the same as the last release. You might have an old release, or you might have a pre-release. (When users request features or report bugs, I sometimes place a pre-release of the next version of EDB on theory.lcs.mit.edu so that their problems are corrected right away.)

`edb-directory`
: A string, the name of the directory containing the EDB source files.

  If `edb-directory` is not set, the user is prompted for the location of the files, which should all be in a single directory.

These diffs are also available by anonymous ftp from '`theory.lcs.mit.edu:/pub/emacs/edb/diffs/`'. The filenames are of the form '`edb-diff-`*oldversion*`-`*newversion*`.Z`', and the files are compressed patch files. To apply such a patch, uncompress it, connect to your EDB directory, and run the patch program with the diff file as input, like so: '`patch < edb-diff-1.14-1.15`'. Don't forget to load EDB and byte-recompile your EDB source directory\n" (use `M-X byte-compile-database RET`) if you run EDB compiled. If you have been running a prerelease of this version, you must get the entire distribution from '`theory.lcs.mit.edu`'.

2.2 In case of trouble

2.2.1 Data display buffer

In the data display buffer, if point is not where EDB expects it to be, or if other information gets out of synch, you may get an error message about a string not being found in the buffer where it was expected. (In order to prevent this sort of confusion, the user is prohibited from aborting when a record is being displayed in the database format buffer; this is done by setting `inhibit-quit` to `t`, if `db-debug-p` is non-`nil`.)

When it looks like point is not in the field it should be in or the text surrounding the fields has been illegally modified, EDB automatically calls the following function and displays the message "I was confused about where I was. Changes to the field might have been lost." (This error message is produced by function `db-parse-buffer-error`.) In the unlikely event that the data display buffer does get confused and is not automatically corrected, you can call the function yourself.

`M-x db-emergency-restore-format`
: Replace a format with a fresh one; use this if the format gets munged. Changes made to the current field since last moving onto it may be lost. If optional prefix arg *recompute* is non-`nil`, `display-record` recomputes the displayed text as well.

2.2.2 Variables

You may find that in some cases the documentation strings and/or default values of some variables are missing—as if the variables hadn't been defined yet. That's because they aren't defined yet; they are associated with part of EDB which hasn't been loaded because it hasn't been needed yet. The documentation and default values will appear when that part of EDB is loaded (if you set such variables, your values will not be replaced). Such variables are correctly declared buffer-local (if appropriate), so you can set them without fear of the changes affecting other buffers.

2.2.3 Exiting Emacs or saving files

If you find you are unable to exit Emacs or to execute `mde-save-some-buffers` (which replaced `save-some-buffers`, which is ordinarily bound to `C-x s`) because Emacs is trying to manipulate a database which doesn't exist or because an EDB bug is triggered by the attempt to save an existing database, you can set the variable `db-databases` to `nil`. This indicates to EDB that there are no databases read into memory and, therefore, no operations will be attempted on them as a part of saving all modified Emacs buffers.

`db-databases`
: Assoc list of database names and databases.

2.2.4 Compiling EDB

2.2.4.1 Expected compilation errors

When EDB is compiled with Jamie Zawinski's optimizing byte-compiler, (available from the GNU Emacs Lisp Code Archive at '`archive.cis.ohio-state.edu`'), several errors will be signalled. The following are expected errors which result from insufficiently fine control of the (otherwise outstanding) byte-compiler's error output.

- Functions `with-electric-help` and `x-flush-mouse-queue` are not known to be defined. EDB supports electric-help and X Windows when they are present, but they may not be in many environments. No run-time error will be raised by the absence of these features.
- Function `link-set-record` is defined as both af unction and a macro. This function is automatically defined when the link structure is defined, but EDB needs a different definition for the function than the one provided by the link structure creator.

2.2.4.2 Load EDB before compiling it

You must always fully load EDB before attempting to byte-compile it. The easiest way to ensure this is to compile EDB by using function `byte-compile-database`, which automatically fully loads EDB.

Otherwise, do `C-u M-x load-database RET` to load a full uncompiled version of EDB before you compile. You may need to load EDB before doing this (in order to define the `load-database` function), but do not omit this step even if you have already loaded EDB. It is not enough to simply do '`(require 'database)`' or call `db-find-file`, since neither of those actions loads all of EDB, only parts of it.

There are two reasons for loading the code before compiling it. The first is that, when the byte-compiler encounters an unknown symbol used in function position, it assumes that it is a function and attempts to funcall it. If the symbol is later defined to be a macro, this leads to a runtime error. The second reason is that, if the variable `db-disable-debugging-support` is non-`nil` (most users will want to use the default value, which is `t`) when compilation occurs, then code for assisting debugging will be compiled out and the code will be slightly smaller and faster, because it will not contain conditional code for printing status reports and intermediate results.

One symptom of compiling EDB without having loaded it is a message along the lines of '`Invalid function: (macro ...)`'. (EDB's implementation uses macros for efficiency, so problems result if, when compiling, an unknown symbol is assumed to be a function but is actually later defined as a macro.) Another symptom is that variables defined in autoloaded files will be reported as "not known to be defined." A problem with compiling EDB when an old version is loaded is that, if macro definitions have changed, the old definitions will be compiled into the new code.

`M-x load-database`
: Load all the files of EDB, the Emacs database. With prefix arg, load source, not compiled, code; EDB will run interpreted. This function is a good candidate for autoloading.

`M-x byte-compile-database`
: Compile source (.el) files in EDB, the Emacs database, which need it. If optional prefix argument *all* is non-`nil`, every source file is recompiled.

`M-x byte-compile-database-all`
: Compile all source (.el) files in EDB, the Emacs database, unconditionally. Calls `byte-compile-database`.

2.2.4.3 No insert-hook

EDB uses `db-insert` instead of `insert`. However, Emacs has a byte-code for `insert`, so EDB's redefinition may be ignored in compiled code. Emacs should really provide an `insert-hook` variable; version 19 does (two of them, in fact).

2.2.5 Using the mouse

EDB redefines various text deletion and insertion commands to ensure that inter-field text is not deleted, that indentation is correctly added and removed when appropriate, and so forth. When cutting or pasting is done with the mouse, however, these function redefined by EDB are bypassed and the data display buffer can be manipulated almost arbitrarily. This can cause various problems; `x-paste-text` often raises errors or produces incorrect results. There are no known problems with using the mouse to select fields and to move around within fields.

Correcting this bug does not have high priority, but users are encouraged to help find a solution or to provide a fix. One solution might be redefining the functions that are called when mouse deletion or insertion occurs; it might also be possible to use insertion-hooks or deletion-hooks, in versions of Emacs that provide them.

2.2.6 Long file names

EDB's files were named to accommodate systems which limit filenames to 14 characters or less; this is why the '`backtrace-fix`' package was renamed '`backtracef`' and other file names (like '`db-interfa.el`' appear to be truncated. There is one remaining problem, however. When the Texinfo documentation is converted into an Info tree, files with names like '`database.info-3`' are created, but the Info files are (all) saved as '`database.info-`'. Users with this problem can rename '`database.texi`' to '`edb.texi`' and modify the '`setfilename`' in its third line to refer to '`edb.info`'; then the resulting files '`edb.info-1`' through '`edb.info-6`' will be saved correctly. Some of the example files may also have names with more than fourteen characters.

2.2.7 Emacs 19

EDB is fully compatible with GNU Emacs 19 (and with Lucid GNU Emacs 19), though it does not yet take advantage of any of Emacs 19's special features (such as pull-down menus, modification hooks, and read-only regions).

GNU Emacs 19 (but not Lucid's version) may appear to hang while reading EDB format files. Emacs 19's regular expression routines are POSIX-compliant and, in some cases, much slower than the old version. (Some searches that used to take fracions of a second now take the better part of an hour!) This Emacs bug evidences itself when EDB is searching for display specifications in the format file, and in particular, if no display specifications contain optional parameters (following a comma), then EDB will read the format file quickly. Format files containing many complicated display specifications may take minutes or even hours to read. Emacs has not hung, but it will respond extremely slowly to user input.

Once the format file has been read, EDB performs as well under Emacs 19 as it does under Emacs 18.

2.2.8 Debugging EDB

2.2.8.1 Enabling debugging messages

Two useful sources of information for locating a problem in EDB are backtraces and the database log. If you encounter an EDB error, you should generate a backtrace and a database log; even if they do not provide you any information, they may help others who will see your bug report. Execute the following command and then repeat the commands that caused an error previously.

`M-x db-prepare-to-debug`

: Prepare to debug EDB. Set variables `debug-on-error`, `db-disable-debugging-support`, and db-debug-p. Also load uncompiled EDB source.

EDB's source code contains calls to debugging macros which print useful messages and save them in the '`*Database-Log*`' buffer for later examination. By default, the calls to these macros are removed at compile time; this results in slightly smaller, faster code. The following variables control this behavior.

`db-disable-debugging-support`

: If non-`nil`, then debugging calls will be compiled out of the source and the variable db-debug-p will have no effect. Setting this variable at run-time has no effect if you are running EDB compiled; you must set it when you compile EDB, or run EDB interpreted. Defaults to `t`.

`db-debug-p`

: T if database debugging is enabled. Defaults to `nil`. Has no effect on code compiled with `db-disable-debugging-support` set.

2.2.8.2 Printing circular structures

EDB's internal representation of the database structure is circular; if you try to print it, Emacs will signal an error. This is a particular problem when debugging functions that manipulate a database. EDB provides a partial fix by including the '`backtracef`' (originally '`backtrace-fix`'; the name was changed to accommodate systems with 14-character maximum file name lengths) package, which at least lets `backtrace` operate in the presence of circular structures; this can let you know where the specific problem lies.

A more complete fix is the custom-print package, which defines versions of the printing commands which support printing circular structures. You will also need to use a debugger which knows about these functions; Edebug version 2.7 or higher fits the bill and is well worth using in its own right. Both custom-print and Edebug are available by anonymous ftp from the GNU Emacs Lisp Code Directory at '`archive.cis.ohio-state.edu`', which is rooted at the directory '`/pub/gnu/emacs/elisp-archive`'. The files of interest are '`functions/custom-print.el.Z`' and '`packages/edebug.tar.Z`'.

After installing these packages (see their documentation for details), you can simply do something like

```
(setq print-level 4)
```

to make circular structures easily debugable. Be sure to reset `print-level` and `print-length` to `nil` before byte-compiling! (If you do not, the byte-compiled code may contain a '`#`' in place of some of the byte codes, resulting in the error message "Invalid read syntax: "#"".) One way to do this is to put the following code in your '`.emacs`' file:

```
(setq pre-byte-compile-file-hook
      (function (lambda ()
  (setq print-level nil
print-length nil))))
(insert-hooks 'byte-compile-file 'pre-byte-compile-file-hook)
```

You can get Noah Friedman's `insert-hooks.el` from the GNU Emacs Lisp Code Directory.

2.2.9 Reporting bugs

If you have problems with EDB or think you've found a bug, please report it to Michael Ernst; he doesn't promise to do anything but he might well want to fix it. Questions about EDB that are not answered in the manual are welcome as well; if the manual is unclear, that's a bug in the documentation. Suggestions for new features or modifications are always welcome (and implementations of such features, even more so); input from users determines which features are moved to the front of EDB's "to-do" list and which ones users apparently don't care about. The '`edb-list`' mailing list (see Section 2.1 [EDB is in beta test], page 7) is also a valuable resource for users of EDB.

Before reporting a bug or trying to fix it yourself, please perform the following steps. First, make sure you are using the most recent version of EDB, since the problem may have already been fixed. The most recent version of EDB can always be found in directory '`theory.lcs.mit.edu:/pub/emacs/edb/`'; the file '`code/database.el`' contains the version number, which is also apparent from the filenames of the contents of directory '`diffs/`'. You can find out which version of EDB you have by looking in your version of '`database.el`' or by typing `C-h v edb-version RET`.

Second, read the appropriate manual sections, so you understand how EDB ought to be behaving (and whether your bug is considered a feature).

Third, try to isolate the problem (for instance, by using the smallest possible data, format, and auxiliary files). If the problem causes an error (usually causing Emacs to beep and display a message in the echo area), then run command `db-prepare-to-debug` and then reproduce the error. This will produce a backtrace and a database log. For further debugging techniques, see Section 2.2.8.2 [Debugging EDB], page 13.

Send a bug report which includes all of the following information.

- A description of how to reproduce the bug and what you expected to occur.
- The version of EDB you are running (do `M-x edb-version RET`) and the version of Emacs (do `M-x emacs-version RET`).
- All files necessary to reproduce the bug, including the database file, the format file, and the auxiliary file, if any. I promise not to look at the content of your database file or distribute it; if you feel uncomfortable sending it nonetheless, construct another one that will result in the same error. Reproducing these files, when they are absent, usually takes longer than tracking down the bug, so if they are not included, I cannot promise to take any action on the bug.
- The backtrace and the database log (found in the '`*Backtrace*`' and '`*Database Log*`' buffers).

3 Database mode

A single database record (typically the "current record") is viewed in a data display buffer.[1] The layout and formatting of the data display buffer—where and how the fields of the current record are shown, and what fixed explanatory text surrounds them—is specified by a data display format. Only the database fields can be edited; the explanatory text is fixed. Creating a new data display format is described in Chapter 11 [Specifying the display format], page 38. Creating a new data display buffer (with the same or a different data display format) is described in Section 11.3 [Making additional data display buffers], page 40. Viewing summary information about all database records at once is described in Chapter 8 [Summary mode], page 32.

Database mode has two basic submodes, view mode (see Chapter 4 [Database view mode], page 18) and edit mode (see Chapter 5 [Database edit mode], page 22). These modes are used, respectively, when examining or manipulating records and when changing information in a particular record. Keystrokes have different meanings in these two modes. In view mode no editing may be done, and many printable characters are redefined to make manipulation of the database easy (for instance, `n` moves to the next record). In edit mode point is in a field of the current record which is being edited; most printable keys insert themselves, and other editing and movement commands work in the ordinary way. In the data display buffer, where database records are ordinarily viewed and edited, one of these two modes is always in effect. (You may be tempted to directly edit a raw database file in its on-disk layout. Do so only if you know what you are doing, and never change a database buffer out of database mode.)

The mode line indicates which mode you are in. It looks something like:

```
-***-Database: machine-dbase        (Edit Abbrev 42/431)---All------------
```

The mode line consists of three modification indicators, the word '`Database:`' (which reminds you that you are in database mode), the name of the database file being manipulated, minor mode information within parentheses, and the usual percentage-of-screen-visible indicator. The minor mode information consists of the database submode (such as view, edit, or summary), any other minor modes which are turned on (such as Abbrev Mode), the number of the current record, and the total number of records in the database.

---

[1] The data display buffer was previously called the format buffer; this is one reason that all of the variables and functions relating to it start with the `dbf-` prefix.

Ordinarily the Emacs mode line contains only one modification indicator consisting of two dashes (not modified), asterisks (modified), or percent signs (read-only). The EDB mode line contains three modification indicators, one each for the database, the displayed record, and the current field. The field indicator is '`*`' if the field under point has been modified, '`-`' if it has not, and '`%`' if it is read-only or if no field is under point (for instance, if the data display buffer is in database view mode rather than database edit mode).

The database is modified only when a changed record is written into it; changes to the displayed record (also called the current record) do not immediately affect the database proper. This permits such modifications to be conveniently undone. (See Section 4.3 [Undoing all changes to a record], page 19, and Section 4.4 [Making changes permanent], page 19.) The upshot of this is that the current record may be modified without the database being modified, since the database is considered modified only when the current record has been processed and the resulting value placed in the database. A similar situation exists for the current field and the displayed record. (See Section 5.2 [Undoing changes to a field], page 22.)

Do not attempt to directly change the major mode of a database buffer; if a database buffer is placed in another mode, the database functions will cease working (they refuse to operate on non-database buffers, since the consequences of such action could be severe); for instance, you may be unable to save any of your work due to errors raised in the execution of `save-some-buffers`. Furthermore, EDB makes assumptions about where point is located in view and edit modes; violating these can cause changes to the current record to be lost.

4 Database view mode

The data display buffer is in view mode whenever field information is not being edited. Most commands to move from record to record and to manipulate records (sorting, printing reports, showing summaries, etc.) are performed in view mode.

Basic operations are described here; more complicated ones, such as searching (see Chapter 6 [Searching], page 25), are given sections of their own.

4.1 Moving around in the database

`n` (`db-next-record`) Go to the *arg*th next record. In that record, go to the current field, if any.

`p` (`db-previous-record`) Go to the *arg*th previous record. In that record, go to the current field, if any.

`<`
`M-<` (`db-first-record`) Show the database's first record. With optional prefix argument, ignores omitting.

`>`
`M->` (`db-last-record`) Show the database's last record. With optional prefix argument, ignores omitting.

`j` (`db-jump-to-record`) Show the database's *arg*th record. Omitting is ignored unless optional argument *respect-omitting* is specified.

There are two special hybrid commands that show more of the current record if there's more to see and otherwise show the next (or previous) record.

`SPC` (`db-next-screen-or-record`) Go to the *arg*th next screenful of this display, or to the *arg*th next record, if this is the last screenful of this display. If point is in the summary buffer and the data display buffer is not visible, then move to the next record.

`DEL` (`db-previous-screen-or-record`) Go to the *arg*th previous screenful of this display, or to the *arg*th previous record, if this is the first screenful of this display. If point is in the summary buffer and the data display buffer is not visible, then move to the previous record.

4.2 Changing to edit mode

When in database view mode, you cannot edit the record being displayed without first changing to database edit mode; this is done by moving to the field you wish to edit. You can click the mouse on the field you wish to edit, or move to the first or last field (and from there to the desired field) via the following keystrokes:

`TAB`
`C-n`
`e` (`db-first-field`) Move to first field.

`C-p`
`M-TAB` (`db-last-field`) Move to last field.

4.3 Undoing all changes to a record

`C-x u` (`db-revert-record`) Set the record to be the same as the corresponding one in the database. In other words, undo any changes made since entering this record.

`C-x r` (`db-revert-database`) Replace the database with the data on disk. This undoes all changes since the database was last saved.

You can also undo changes to only a particular field; see Section 5.2 [Undoing changes to a field], page 22.

4.4 Making changes permanent

The user edits a copy of a database record; the database itself is not changed until the user commits the changes. This occurs automatically whenever any command causes a different record to be displayed, when the database is saved, when a report is generated, and so forth. It does *not* occur when the user switches from edit mode to view mode, though the field modification flag (in the mode line) will become a percent sign and the record modification flag, an asterisk. When the record is committed, the record modification flag will become a dash and the database modification flag will become an asterisk.

The user can manually install the current record, as modified, into the database. The following two functions are identical:

`RET` (`db-accept-record`) Install the current record in the database; make any changes permanent.

`M-x db-commit-record`
Install the current record in the database; make any changes permanent.

Committing a record makes the changes permanent only insofar as they become part of the in-memory representation of the database. The on-disk version is not affected unless the user overwrites it by using `db-save-database` or `db-write-database` (see Section 1.2 [Invoking EDB], page 2), or otherwise indicates that the database should be written to disk (say, by responding to a question about saving the database).

4.5 Adding and removing records

`a`
`i` (`db-add-record`) Add a new record to the database immediately before the current record.

`c` (`db-copy-record`) Insert a copy of the current record in the database immediately after it. The second of the two records is made the current record. With a prefix arg, inserts that many copies.

`o` (`db-output-record-to-db`) Copy the current record to *database*. *database* must be read in and compatible with the current database.

`d`
`k` (`db-delete-record`) Remove the current record from the database. With a prefix arg, doesn't verify.

By default, deleting a record marks the database as modified. Set the following variable to change this behavior.

`db-delete-record-modifies-database-p`
Non-`nil` if deleting a record should mark the database as modified.

4.6 Exiting database mode

`q` (`db-quit`) Quit editing the database for now; bury its buffers.

`x` (`db-exit`) Be done with the database; like `db-quit`, but offers to save any changes. With prefix arg, kills the data display buffer, and the database, if that was its only data display buffer.

You can also kill a database buffer in the usual way (for instance, by using `kill-buffer`); this causes `db-kill-buffers` to be called. If the database's last buffer is killed this way, the database itself is also killed. No offer is made to save changes; call `db-exit` in order to do that.

`db-kill-buffers`
: Kill this buffer, and the associated summary or data display buffer, if any. If its last data display buffer is killed, the database is killed too. Does not offer to save changes to the database or to this record; use `db-exit` with optional argument to do so.

5 Database edit mode

In database edit mode, point is always in the field currently being edited. The database is not modified as soon as changes are made in edit mode. A copy of the record in question is displayed and edited, and only when the user moves to a new record, initiates some other global action (not specific to the edited record), or explicitly commits the changes (see Section 4.4 [Making changes permanent], page 19). This permits easier undoing of incorrect modifications.

In order to perform most record-level operations, the user exits edit mode, switching to view mode, and then performs them there. Several commonly used commands, however, such as searching and moving from record to record, are accessible directly from edit mode.

Commands that move from field to field check the validity of the current field before moving off it; commands that move from record to record do this as well, then make any changes in the current record permanent (though the database file on disk is not changed).

Basic operations are described here; more complicated ones are given sections of their own.

5.1 Exiting edit mode

`C-c C-c`
: (`db-view-mode`) Switch to database view mode. With an argument, toggle between view and edit modes.

5.2 Undoing changes to a field

You can undo changes to the current field via Emacs' usual undo facility; use `C-x u` or `C-_` to undo changes made since entering the current field.

You can also revert the current field to its original value; this is useful if you made a change, moved off the field, and then moved back onto it.

`C-x U`
: (`db-revert-field`) Replace the onscreen text in this field with that of the underlying record. In other words, undo any changes made since entering this field.

From database view mode, you can simultaneously revert every modified field of a record to its original value; see Section 4.3 [Undoing all changes to a record], page 19.

5.3 Moving from record to record

These commands make any changes to the current record permanent.

`M-n` (`db-next-record`) Go to the *arg*th next record. In that record, go to the current field, if any.

`M-p` (`db-previous-record`) Go to the *arg*th previous record. In that record, go to the current field, if any.

5.4 Moving from field to field

`TAB` (`db-next-field`) Move to *arg*th next reachable field, wrapping if necessary. When called interactively, *arg* defaults to 1.

`M-TAB` (`db-previous-field`) Move to *arg*th previous reachable field, wrapping if necessary. When called interactively, *arg* defaults to 1.

`M-<` (`db-first-field`) Move to first field.

`M->` (`db-last-field`) Move to last field.

Also see the keystrokes `C-n` and `C-p`, described below.

5.5 Movement within a field

Many Emacs cursor motion commands retain their standard meanings, except that they do not move outside the field; among these are `C-f` (`forward-char`), `C-b` (`backward-char`), `M-f` (`forward-word`), `M-b` (`backward-word`), `C-a` (`beginning-of-line`), and `C-e` (`end-of-line`).

The line-movement commands have slightly changed meanings: if the motion would take the cursor out of the current field, then they move to the next field.

`C-n` (`db-next-line-or-field`) Move to *arg*th next line. If that would move out of the current field, move to the closest field to that, but not the current one, wrapping if necessary.

`C-p` (`db-previous-line-or-field`) Move to *arg*th previous line. If that would move out of the current field, move to the closest field to that, but not the current one, wrapping if necessary.

5.6 Editing a field

Many Emacs editing commands retain their standard meanings; for instance, printing characters insert themselves and deletion commands work as usual, except that they will not make changes outside the field; among these are `C-d` (`delete-char`), `DEL` (`backward-delete-char`), `M-d` (`kill-word`), `M-DEL` (`backward-kill-word`), and `C-k` (`kill-line`).

5.7 Getting help

You can get some information about the current field, such as what type of value it expects or what its contents signify, by using the following command.

`M-?` (`db-field-help`) Display help for current field using the recordfieldspec help-info field. If this is a string, display it. If it is a form, eval it and display the result.

6 Searching

A useful and commonly used database operation is searching for records that meet some criteria: for instance, finding a particular record or indicating that following operations should only apply to records that correspond to an address in greater Boston. EDB provides several functions to support such operations.

To perform a search pertaining to the contents of only one field, move to that field and use the following command:

`M-s` (`db-search-field`) Search for occurrences of *pattern* in the current field of any record. Finds the first match after the current record; wraps around automatically. With prefix arg, marks all matches in addition to going to the first one. If omitting is in effect, omitted records are ignored.

The same keystroke in view mode permits specification of patterns which depend upon the contents of several fields:

`s`
`M-s`
`M-S` (`db-search`) Please do not use `db-search`, which is unimplemented; use `db-search-field`. In a future version of EDB, `db-search` will permit searching on all fields of a record simultaneously.

For a description of marking, see Chapter 9 [Marking and omitting], page 34.

6.1 Search patterns

Search patterns can be as simple as a datum to match exactly or as complicated as the conjunction, disjunction, and negation of tests to be performed on field contents.

6.1.1 Basic patterns

A basic search pattern has the same form as the data that is kept in the field; for instance, to search in a string field for a particular string, use that string; to search for the date March 14, 1967, use '`3/14/67`' or '`14 March 1967`' or any other accepted date format. A basic pattern is treated somewhat more richly than a literal, however. In a string field, typing a string results in a match for any element which contains it as a substring; typing '`ail`', without the quotes of course, will match both "ailment" and "fail". In a date field, '`3/67`' will match any date in March of 67, not just those March 67 dates which specifically exclude a day of the month.

6.1.2 Comparisons

A search pattern may also be a comparison prefix (`<`, `>`, or =) plus a datum which is treated exactly like any other element of that field. In a string field, '`<ail`' will match all elements lexicographically less than "ail"; '`=ail`' will match only fields containing exactly "ail", but not "ailment" or "fail". Warning: '`= ail`' looks for an entry containing " ail", that is, four-character sequence starting with a space.

In a date field, searches work slightly differently; for instance, '`>3/14`' will match dates after March 14 in any year, and '`=3/67`' will match only dates in March 67 whose day of month is not specified. For more information about the interpretation of patterns, see the documentation for the particular types.

6.1.3 Logical connectives

More complicated patterns can be built up out of simpler ones via the logical connectives AND, OR, and NOT. These work in the obvious way. One pattern which finds any date between the ides of March and Christmas, inclusive, is '`> 3/14 AND < 12/26`'; two patterns which find dates except March 14 are '`NOT 3/14`' and '`< 3/14 OR > 3/14`'. To find strings that either contain the substring "ail" or start with a, b, or c, use '`ail OR < d`'.

The precedence of these connectives is: NOT, which is most tightly bound to its test, then OR, then AND. There are no provisions for grouping or otherwise overriding this ordering. Connectives (and REGEXP, described below) consume all surrounding spaces and tabs.

6.1.4 Other pattern operations

One other pattern operation of interest is the regexp operator for string fields. This is invoked by either using REGEXP with surrounding spaces, or / without a trailing space. For instance, '`/^[ace]`' matches any string field starting with a, c, or e; '`/.`' matches any nonblank string field;

and '`NOT REGEXP a.*b.*c`' matches any string which does not contain the letters a, b, and c in order. This last example shows that EDB's search commands are more powerful than general regular expression searching.

7 Sorting

`M-x db-sort`

Sort the database. With a prefix arg, don't confirm the sort order.

Ordinarily, calling this function invokes a graphical field ordering tool which permits easy specification of which fields are significant and what their order of importance as sort keys is. In a database whose records had fields foo, bar, baz, bum, and bee, and in which records should first be sorted on baz in increasing order, then on foo in decreasing order, and finally on bum in increasing order, ignoring bar and bee entirely for the purposes of the sort, the display would look like

```
==== Significant fields:
  baz                   increasing
  foo                   decreasing
  bum                   increasing
==== Nonsignificant fields:
  bar                   increasing
  bee                   increasing
==== Omitted records to end:  No
```

In addition to which fields should be sorted by, the sort interface permits specification of how omitted records should be treated.

`t` (`dbsi-toggle-omitted-to-end`) Toggle the boolean value of dbsi-omitted-to-end-p. This controls whether omitted records should all be placed at the end of the sorted order or should be sorted according to the same criteria as non-omitted records.

To change the relative order of fields, and whether they're significant or not, use the following commands.

`C-k` (`dbsi-kill-line`) Kill field on current line, placing it in the sort interface kill stack.

`C-y` (`dbsi-yank-line`) Yank most recently killed (lifo ordering) field, inserting it before point. This removes the field from the sort interface kill stack.

To specify how a particular field should be ordered, use the following commands.

`i` (`dbsi-increasing`) Specify that the field at point should use an increasing ordering.

`d` (`dbsi-decreasing`) Specify that the field at point should use a decreasing ordering.

`o` (`dbsi-ordering-function`) Specify an ordering function for the field at point. An "ordering function" returns -1, 0, or 1 depending on whether its first argument is less than, equivalent to, or greater than its second argument.

`s` (`dbsi-sorting-function`) Specify a sorting function for the field at point. A "sorting function" returns `t` if its first argument is less than its second argument and `nil` otherwise.

Each database has a default sort order in its `field-priorities` slot (see Section 17.1 [The database structure], page 85) which is used when setting up the sort interface (and is used for sorting when no other ordering information is specified). When the user exits the sort interface, that slot can be set to the ordering depicted on the screen (see below).

Each data display buffer also has a default sort order: the database's `field-priorities` slot is ignored if the variable `dbf-field-priorities` is `nil`. The sort interface permits the setting and clearing of this value as well; when it is cleared, the default sort order is once again taken from the database's `field-priorities` slot. Setting the database's sort order automatically clears the buffer-local sort order.

`dbf-field-priorities`
: The list of field priorities for this database in this data display buffer. If non-`nil`, overrides the database's field-priorities slot.

`dbf-omitted-to-end-p`
: The default, local to this data display buffer, for the omitted-to-end-p database slot. Only used if dbf-field-priorities is non-`nil`.

The following commands are used to exit the sort interface; most of them also cause the database to be sorted. Some of them set the default ordering for the database or the data display buffer.

`RET`
`C-c C-c` (`dbsi-use-ordering-make-database-default`) Use the current ordering to sort, and make it the default for future sorts of this database. The user is warned if there are killed, non-yanked fields.

`A`
`U` (`dbsi-use-ordering-make-buffer-default`) Use the current ordering to sort, and make it the default for future sorts in this data display buffer only. The user is warned if there are killed, non-yanked fields.

`a`
`u` (`dbsi-use-ordering`) Use the current ordering for this sort only.

`!` (`dbsi-this-field-only`) Sort according to only the field at point. All editing of other fields is ignored.

`q` (`dbsi-quit`) Abort the sort and exit the sort interface.

`c` (`dbsi-quit-clear-buffer-default`) Clear the default sort order for this buffer and exit the sort interface without sorting. In the future, the default sort order will come from the database.

The sort interface returns a field priorities list to be used when sorting; when the sort interface is entered, either the value of the `dbf-field-priorities` variable, or the database's `field-priorities` slot, is being used. For information about the format of the field priorities list, see Section 17.1 [The database structure], page 85.

Sorting does not ordinarily mark the database as modified, because not the data itself, but only the way it is arranged, has been changed. If you can set `db-sort-modifies-p` to non-`nil`, then whenever a database is sorted (even if the resulting order is the same as the original one), the database will be marked as modified.

`db-sort-modifies-p`

If non-`nil`, then sorting a database marks it as modified too.

7.1 Sorting and ordering functions

In order to specify the relative order of two field values (for the same field, but from different records), the database designer provides a sorting function, an ordering function, or both. If only one is provided, the other is automatically generated from it. In any event, only one of them is used for a given field on any particular sort.

A sorting function takes two field values as arguments and returns `t` if its first argument is less than its second argument (that is, the first argument appears previous to the second in the sorted order). The sorting function returns `nil` if the arguments are equal or if the first argument is greater than the second (appears later in the sorted order).

An ordering function, on the other hand, returns complete information about the relative order of its two arguments: it returns -1, 0, or 1 depending on whether its first argument is less than, equivalent to, or greater than its second argument.

Use of an ordering function can result in fewer comparisons in some cases, because it returns more information. This is worthwhile if a significant amount of processing is required before the comparison is done. For example, suppose that two addresses are being carefully checked for equality and some of the steps leading up to that are expansion of abbreviations, standardization of spelling, capitalization, and spacing, etc.; then it is better to return the exact relative ordering than to possibly require another time-consuming operation to determine it.

If it is possible to canonicalize the values beforehand, that may be even more efficient, but that is not always possible; consider the case of a small (but tedious to extract) part of the information in each field being compared. Therefor, the user is permitted to specify an ordering function.

8 Summary mode

A summary is a listing containing abbreviated information about every record; it permits many records to be viewed at once.

This is available in view mode via the following command:

`D`
`H`
`h` (`db-summary`) Display a summary (or directory) of all database records according to the variable `dbf-summary-function`, which is set by `dbf-set-summary-format`. The summary appears in a separate buffer. When called from the summary buffer, this updates the summary.

When a summary is created, the summary display format appears in the summary buffer once for each record, with appropriate values substituted for its display specifications. Omitted records are included in the summary only if the data display buffer variable `dbf-summary-show-omitted-records-p` is non-`nil`.

`dbf-summary-show-omitted-records-p`

Nil if omitted records should be omitted from the summary, `t` otherwise.

The entire format is indented by two characters; the first and second columns contain '`+`' and '`[`', respectively, if the record is marked or omitted. For information about marking and omitting records, see Chapter 9 [Marking and omitting], page 34.

The summary buffer is not updated whenever a record value changes; in the interest of efficiency, it remains as is until the next `db-summary` command is issued, at which time the summaries are redisplayed after all, some, or none of them have been recomputed. For the same reason, when a single mark or omit bit changes, the summary is updated; when many change, it is usually not. [[[Describe exceptions. This may change soon in any event.]]]

When point is in the summary buffer, the associated data display buffer (nearly) always displays the record under point. (Some Emacs commands can move point without EDB noticing.) Movement in the summary buffer is by any of the ordinary Emacs commands, including searching. Most view mode commands also work in the summary buffer.

The summary display format defaults to the first non-literal line in the database display format—that is, the first line which contains a display specification.

Summary display formats can display information any way that an ordinary display format can, including showing more than one field of the record in question or spanning several lines. The only restriction is that the summary display format must cover a specific number of lines: each display specification must have its `min-height` and `max-height` slots set to equal values. For more information about display formats, See Chapter 15 [How information is displayed], page 65.

The following command may appear in a format file or an auxiliary file, or it can be invoked directly by the user. When it appears as an Emacs Lisp form, remember the special meaning of the backslash character and double it where necessary.

`M-x dbf-set-summary-format`

Specify the format used in the Database Summary buffer. Argument *summary-format* is a string containing display specifications. Call this in the data display buffer, or in a format file or auxiliary file.

9 Marking and omitting

The marking facility permits operations to be performed on only certain records of a database. For instance, to create a report which describes only some of the database records, you would first mark the records of interest. Then you would call `db-report` with a prefix argument (do so by pressing `C-u` first), or would make optional second argument *MARKEDP* non-`nil`. See the documentation of the individual operations to see whether they support operation on only the marked records.

The omitting facility is similar to the marking facility in that it restricts attention to a subset of the current database; however, omitted records are ignored by most operations. By default omitted records are skipped by the record-motion commands, excluded from searches, reports, and other functions, and so forth. Omitting is useful when the user wishes to concentrate on a subset of the database without being distracted by other records that may be present; while the records are still present in the database, they are not seen by the user.

Mark and omit bits are associated only with the version of a database being operated upon. They are never saved in the database file, and when a database is first read in, all of its mark and omit bits are unset; that is, they are boolean false. (When a database is written to disk, although the disk version of the data will not contain the mark and omit information, that information is not lost from the working copy of the database.)

This behavior is a feature, not a bug. Marking and omitting are intended to help the user temporarily group database records for operations upon it; if there is information that cannot be recreated from a record's fields, then the user should consider adding another field for that information. On the other hand, mark or omit criteria may be complicated. If such a pattern is used often, then the user may wish to write a function to set the bits appropriately, which function could be bound to a keystroke or automatically executed when the database is read in.

9.1 Setting the mark and omit bits

Every record may be thought of as having a pair of bits or boolean values indicating whether it is marked and whether it is omitted. The most straightforward way to set these bits is to use an operation to mark, unmark, omit, or unomit a particular record; these are bound to keystrokes in database view mode.

`m` (`db-mark-record`) Toggle whether the current record is marked. With a nonzero prefix argument, set it to be marked. With a zero prefix argument, set it to be unmarked.

`O` (`db-omit-record`) Change whether the current record is omitted. With a nonzero prefix argument, set it to be omitted. With a zero prefix argument, set it to be unomitted.

The searching commands, when called with a prefix argument, mark each matching record; Chapter 6 [Searching], page 25.

Once all records of interest have been marked, through one or more marking and/or searching commands, unmarked records can be omitted from consideration. This is useful if you want to work on only a small number of records, or if specifying the records of interest is easier than specifying those not of interest: instead of omitting all the uninteresting records, simply mark the interesting ones, then use the following command to cause the unmarked ones to become omitted.

`M-x db-omit-unmarked-records`
: Omit all unmarked records. Also clears all mark bits and sets `dbc-omit-p`.

The converse operation transfers information from the omit bits to the mark bits.

`M-x db-mark-unomitted-records`
: Mark all unomitted records. Also clears all omit bits.

It is also possible to clear all the mark or omit bits.

`M-x db-unmark-all`
: Clear the mark bit of every record.

`M-x db-unomit-all`
: Clear the omit bit of every record.

9.2 Movement among marked and omitted records

Ordinarily, record movement commands (those which move from one record to another) ignore omitted records, so that the user never lands on an omitted record. Marked records, on the other hand, are not treated specially by the record movement commands. The following database view mode keystrokes permit you to move to omitted records or to move directly to marked records.

`M-n` (`db-next-record-ignore-omitting`) Go to the *arg*th next record, ignoring omissions. That is, all records, even those which are omitted, are counted.

`M-p` (`db-previous-record-ignore-omitting`) Go to the *arg*th previous record, ignoring omissions. That is, all records, even those which are omitted, are counted.

`M-C-n` (`db-next-marked-record`) Go to the *arg*th next marked record. Omitted records are treated according to db-omit-p.

`M-C-p` (`db-previous-marked-record`) Go to the *arg*th previous marked record. Omitted records are treated according to db-omit-p.

9.3 Details of omitting

The "omitted" bit of each record has no effect unless the following variable is set:

`dbc-omit-p`

Non-`nil` if omitting is in effect, `nil` otherwise. Use function `dbc-set-omit-p`, which works in either a data display buffer or a summary buffer and sets the variable's value in both, instead of setting this directly. Setting this to `nil` is cheaper than changing the omit function to the empty one, since no omit bits are recomputed. This variable is automatically set by the omitting functions.

When `dbc-omit-p` is `nil`, the values of records' omit bits are remembered, and they may still be set and unset, but they have no effect on any operations until `dbc-omit-p` is once again set to non-`nil`. When `dbc-omit-p` is non-`nil`, "Omit" will appear in the mode line of the database buffer.

The following operations work in either database view mode or database summary mode.

`M-o` (`db-omitting-toggle`) Change whether omitting is in effect. With a nonzero prefix argument, turn omitting on. With a zero prefix argument, turn omitting off.

This does not change the current omit-function, and an omit bit is always computed for each record, but omit bits have no effect on any operations if omitting is not in effect.

`M-C-o` (`db-toggle-show-omitted-records`) Toggle whether omitted records are shown in the summary. With a nonzero prefix argument, show omitted records in the summary. With a zero prefix argument, don't show omitted records in the summary.

`M-O` (`db-omitting-set`) Set the criteria for automatically determining whether to omit a record. This isn't implemented yet.

10 Reports

Reports can be generated from a database by using the following command in view mode:

`r` (`db-report`) Create a report according to *report-filename*. Prefix argument *markedp*, if non-`nil`, means report on only marked records. If omitting is in effect, omitted records are not reported upon. When called interactively, prompts for *report-filename*.

The way a report looks is specified in precisely the same as are display formats and summary formats (see Chapter 15 [How information is displayed], page 65). This information must be placed in a file; the user cannot type it directly when creating a report. This restriction makes errors in the report format easier to correct.

The report is placed in the '`*Database Report*`' buffer, which is in Text mode. The information may then be edited, saved to disk, or otherwise manipulated. The buffer is in Text mode and is not yet a file; you must save it to make it a file.

To create a report which mentions only marked records (see Chapter 9 [Marking and omitting], page 34), supply a prefix argument to the report command by typing `C-u` first.

10.1 Bugs in report generation

There are currently a few unfortunate bugs in EDB's handling of reports. The most noticable of these are errors in the handling of format information: primarily, indentation is not respected. These problems will be corrected in a future release of EDB.

11 Specifying the display format

Different layouts and on-screen arrangements of the values stored in database records are appropriate when the user is concentrating on different aspects of the data. Sometimes the user would prefer to see just a few of the fields; at other times he may want to see the records in full detail. It may also be appropriate for the display format of a record to depend on the record's field values. This section describes how to choose a different display format for the record being displayed, either manually or automatically.

11.1 Changing display formats

EDB permits the creation and use of a variety of display formats with a single database; the user can also conveniently change the way that a particular record is displayed by using `db-alternate-format` and specifying the filename of the new display format, or a nickname for the format that has been specified by the user or the database designer. Choosing a different format does not create a new data display buffer; it changes the way that records are displayed in the current one.

`M-x db-alternate-format`

: Select and use an alternate display format to view the database. If neither *format-name* nor *filename* is specified (as is the case when this is called interactively), the user is prompted for them. In Emacs Lisp code, if `dbf-alternate-format-names` has been been set, usually only one of the arguments is specified. If both are specified, then *format-name* becomes a name for the format *filename* specifies; if *format-name* is already associated with a different format file, an error is signalled.

: If the current format is unnamed, the user is prompted for a name to give it, so that it can be conveniently restored if need be. This behavior is suppressed, and the record is not displayed, if the function is not being called interactively.

: Selecting the current format does not cause any work to be done.

: Some databases automatically set the format of the record being displayed, usually by setting `dbf-before-display-record-function` to a function that overrides the format in effect when a record is about to be displayed. This may cause this function to appear not to be doing any work. In actuality the format is being set, then reset.

`dbf-alternate-format-names`

: Association list of format names and format specifiers. Each format name is an arbitrary string. A format specifier is a filename or a list of values for format variables. The user sets the format specifier to a filename, and after that format file has been read, EDB replaces the filename with a list of values for format variables, so that the file need not be read again.

It is convenient for a database designer to set this, pre-assigning format names to files so that the user only needs to remember the format names, not the filenames.

`dbf-format-name`
: The string representing the format currently in use.

`dbf-format-file`
: The format file from which this format was built.

These values can be set in the auxiliary or format files so that the user can choose a format name (with completion) instead of having to remember a filename. If the selected format's specifier is a filename, then after the file is read in, the format-spec is modified by replacing the filename with information about the format such as the displayspecs, the invariant text between them, and so forth. Subsequent selections of that format will not cause disk accesses. For an example of the use of `db-alternate-format`, see Section 16.2.3 [Record display hooks], page 78.

While it is not currently possible to selectively omit certain fields from a data display buffer, judicious use of alternate formats can result in nearly the same effect.

11.2 Execution of format file eval expressions

Often the "local variables" section of a format file contains code that should only be executed once, or should only be executed before the database is read in, because of either efficiency or correctness constraints. Because of this, the "local variables" section of a format file is executed only when it is read in from disk (which is usually only once). In order to cause an expression to be evaluated every time that a particular display format is selected, use the `dbf-always` macro:

`dbf-always`
: Execute *body*, and place its forms in `dbf-always-forms`. They will be executed each time that this format replaces another.

`dbf-always-forms`
: Forms executed every time that the format is selected. These forms are only executed when a different format is replaced, not every time that a record is displayed (or even every time that `db-alternate-format` is called). See also `dbf-before-display-record-function`.

Of course, it is often valuable to overwrite a value when the display format changes; this is the purpose of `dbf-always`. It is always safe to set variables whose name begins with `dbf` in such forms, though changes to some such values—none of which the user should be changing anyway—will not take when a display format is being returned to (though they will work when it is first chosen). This will affect the user only in that a call to `dbf-set-summary-format` will have an effect only the first time that a format file is read in, not every time that it replaces another, even if it is enclosed in a `dbf-always` form.

The forms in `dbf-always-forms` are not executed every time that a record is displayed, or even every time that `db-alternate-format` is called, but only when a format replaces another one (that is, `db-alternate-format` is called and its first argument is not equal to `dbf-format-name`).

Here is an example of a common problem with an expression which causes an error if evaluated every time that the format is selected. The primary format file (the one that is used when the database is read in) is permitted to set the `fieldnames` slot of the database structure to a list; other parts of the database initialization code propagate that information into other slots of the database structure and change the list into a vector, which is its proper representation. If the user switches to another display format and back to the primary one, and the `database-set-fieldnames` expression was evaluated, then the next attempt to access the `fieldnames` slot of the database as a vector would cause an error. In this case, the proper solution is to use `database-set-fieldnames-to-list` (see Section 17.1 [The database structure], page 85) instead, but such functions are not provided for every slot that it would be dangerous to set. If several format files all set a value which is dangerous to change, then another possibility is to check the value before setting it: if it is already set, then don't do anything. Another possibility is to move all assignments to database slot values from the format file to the auxiliary file.

11.3 Making additional data display buffers

In addition to changing the display format of an existing data display buffer, it is sometimes useful to have two different data display buffers both examining the same database, either so that two different records can be viewed or edited simultaneously or so that two different formats can be used at the same time—or both.

Use the following function to create a second (or additional) data display buffer for the current database.

`M-x db-additional-data-display-buffer`

Create another data display buffer in which to view this database.

If you edit the same record in more than one data display buffer, only the last one committed (by calling `db-commit-record`, moving to another record, saving the database, etc.) will have an effect. (Simply switching from database edit mode to database view mode does *not* commit the changes; failing to commit changes will make it appear that changes in one data display buffer are not being communicated to the other ones associated with the same database. For more on committing, see Section 4.4 [Making changes permanent], page 19.) It is perfectly safe, however, to edit different records of the same database in different data display buffers, or to perform any other database manipulations.

This section does not describe how to specify a new format, only how to use multiple data display buffers. To learn about formatting directives and specification of display format files, see Chapter 15 [How information is displayed], page 65.

12 Designing a database

Preceding chapters have discussed the use of an already-existing database. However, before a user can manipulate a database using EDB, a database designer (who will possibly also be a user later on) must have specified pertinent information about the database, including

- the number of fields per record and the type of each one (see Chapter 13 [Record field types], page 45);
- the layout of the file containing the database (see Chapter 14 [Database file layout], page 51);
- how each record should be displayed on the screen when it is being viewed or edited (see Chapter 15 [How information is displayed], page 65); and
- special actions to be performed, such as updating the last-edit-date field of a record whenever any other field is modified (see Chapter 16 [Customization], page 75).

Detailed information about the database's internal representation is supplied in Chapter 17 [Database representation], page 85.

12.1 Creating a new database

This section tells you how to quickly create a trivial database; please don't be satisfied with this, but follow the references to learn how to access EDB's more sophisticated features. Another good way to learn about EDB is to look at the example databases provided with EDB (see Chapter 2 [Installation], page 6).

First, you must decide how many fields the database shall contain, and what their names are. The names are Lisp symbols, and they are primarily used internally—most users will never be aware of how EDB refers to the fields. Let us make a name database with three string fields called "first", "middle", and "last" and one integer field called "age".

As mentioned in Section 1.2 [Invoking EDB], page 2, you need three files to run EDB: a data file, a format file, and an auxiliary file. The auxiliary file is optional; its functionality can be placed in the format file.

The data file may have any one of a number of layouts. Perhaps the simplest is tab-delimited text, which is also the default: fields are separated by tab characters and records are separated by newlines. EDB doesn't deal well with empty databases (because it always tries to display a record in the data display buffer), so create a file with at least one record. For instance, we could create a file '`names.dat`' containing

```
Harry	S	Truman	88
```

where a single tab character separates the words. It doesn't matter whether the file contains a final newline. The easiest way to create a database file laid out in EDB's internal representation, which permits faster reading and writing of the data file, is the following. Create a database in some simpler format (such as tab-delimited text), read it in the usual way, set database slot `internal-file-layout-p` to `t` (for instance, via `C-u M-x db-toggle-internal-file-layout RET`), and finally save the database. It will be saved in EDB's internal file layout; for details, see Section 14.1 [Internal file layout], page 51. For more information about different data file layouts, see Chapter 14 [Database file layout], page 51.

The format file specifies how a record is displayed on the screen; a backslash followed by a field name indicates that the field's contents should be inserted there. For instance, here is a format for our names database:

```
Family: \last
Given:  \first
Middle: \middle

Age:    \age
```

For more information about specifying display formats, see Chapter 15 [How information is displayed], page 65.

The auxiliary file is used to customize EDB for a particular database; the only information that it absolutely must contain is the names (and possibly types) of the database fields. They are set by function `database-set-fieldnames-to-list` (see Section 17.1 [The database structure], page 85); by providing additional information to this call, you can specify types other than string (which is the default) for your record fields. Any auxiliary file customization may instead appear in the "Local variables" section of the format file (see Section 14.5 [Reading from disk], page 63). So our format file '`names.fmt`' would look like this:

```
Family: \last
Given:  \first
Middle: \middle

Age:    \age
```

```
Local Variables:
eval: (database-set-fieldnames-to-list database
                                       '(first middle last (age . integer)))
End:
```

For more information about customizing EDB for your application, see Chapter 16 [Customization], page 75.

Now that we have the two files 'names.dat' and 'names.fmt', we are ready to invoke the database. Simply call `db-find-file` on 'names.dat'; 'names.fmt' will be automatically used as the format file, and you can begin editing the database and adding new records.

12.2 Manipulating database fields

EDB will have a graphical method to manipulate databses by adding, removing, and rearrange fields, among other other manipulations; a rudimentary version of this functionality exists in the 'db-convert.el' file but lacks a good user interface or documentation. Here is a way to perform those actions by directly manipulating the database file.

Make sure the database is stored in a regular layout (the rest of this section assumes that the standard tab-separated text layout is used; see Section 14.2 [Regular file layout], page 52). If the database file is in EDB's internal file layout, convert it to a regular layout by reading in the database, setting database slot `internal-file-layout-p` to `nil` (for instance, via `C-u 0 M-x db-toggle-internal-file-layout RET`), and saving the database. (See Section 14.1 [Internal file layout], page 51.)

In the tab-separated database format, you can add new fields by adding new tabs in the right places in each record; simply edit the database file to add the next fields as desired. Similarly, you can delete or rearrange fields. You may wish to use keyboard macros, or write an Emacs Lisp function, to help youwith this chore.

When you perform this edit, be sure that auto-fill-mode is turned off, lest spurious newlines be added to the file.

You will also need to modify any parts of your database that depend on the number or order of fields; for instance, a call to `database-set-fieldnames-to-list` will need to be changed, and the format file(s) should be edited if you wish to be able to view or edit the contents of the new fields.

13 Record field types

Each of a database's records consists of similarly-typed fields: the fifth field might always contain an invoice number, for instance, but the invoice number would vary from record to record. This chapter describes how to specify and use different field types.

The most important information about a record field is

- display representation onscreen and in reports
- EDB's internal representation
- storage representation in data files
- how to convert among these representations
- how to sort items of that type

This information is separated into a display specification and a record field specification. The display specification determines how a field's contents are displayed and parsed onscreen (say, in a data display buffer). The record field specification controls everything else about the record field; its information does not depend on the onscreen (or in-report) visual appearance of the field. The database designer specifies a displaytype for each display specification in the format file (that is, for each location in the data display buffer that will contain a representation of some record field). The database designer specifies a record field type for each field in a database record, whether or not the field is ever displayed to the user. (A particular field's contents may appear zero, one, or more times in a data display buffer; one displayspec structure is created for each occurrence.)

Displaytypes and record field types are distinct; they supply complementary information. There is not even a one-to-one relation between them. A particular record field type may be displayed in any of a number of ways by using different displaytypes—dates are such an example. On the other hand, record field types which are interpreted, sorted, and stored on disk differently, but which have the same internal representation—say, as a string—could all be displayed and edited using the same displaytype.

This chapter discusses record field types, record field specifications, and the recordfieldspec structure, the internal structure which holds the information specified by the former two items. For more information about displaytypes, display specifications, and the displayspec structure, see Chapter 15 [How information is displayed], page 65.

13.1 Specifying a record field type

The function `database-set-fieldnames-to-list` (see Section 17.1 [The database structure], page 85) is used to specify the types (and names) of record fields.

A record field type gives information about one field of the database's records: the type of the contents, what sorting function to use, how to write it to disk and read it back, constraints on its value, etc. The database designer must provide one for each record field. Most of the time one of the predefined types (see Section 13.2 [Predefined record field types], page 47) suffices. The remainder of this section describes how to define a new record field type, when that is necessary.

The first step in creating a new record field type is to make a recordfieldspec structure (when a predefined record field type is used, EDB looks up an existing recordfieldspec). A recordfieldspec can be created from scratch by calling `make-recordfieldspec`, but it is often easier to copy an existing one with `copy-recordfieldspec` (use `recordfieldtype->recordfieldspec` to look up a predefined recordfieldspec; for a list of such, see Section 13.2 [Predefined record field types], page 47) and then modify the copy as appropriate.

`recordfieldtype->recordfieldspec`

Return the recordfieldspec associated with symbol *recordfieldtype*.

Next, set the recordfieldspec's slots to appropriate values by using `recordfieldspec-set-`*slotname*; for a list of the slots, see Section 13.3 [The recordfieldspec structure], page 48. Finally, install the record field type by calling `define-recordfieldtype-from-recordfieldspec`

`define-recordfieldtype-from-recordfieldspec`

Define a recordfieldtype named *typename* (a symbol) with the default *recordfieldspec*. DISPLAYSPEC may also be a typename symbol itself. After this call, `recordfieldtype->recordfieldspec` called with argument *typename* returns the proper record field specification.

Examples of record field type creation can be found in the file '`db-types.el`', which contains a number of record field type (and displaytype) definitions that can be studied or copied.

Record field types should not be confused with display types; a display type is used to specify how a particular value is shown on the screen, but a record field type constrains the information actually contained in the record field.

13.2 Predefined record field types

The following record field types are predefined by EDB; their definitions can be found in the file '`db-types.el`'. The recordfieldspec structure contains a record field type's information; see Section 13.3 [The recordfieldspec structure], page 48. Users can define record field types in the same way as '`db-types.el`' does; these record field types are not privileged in any way and are provided only for convenience. For more information about recordfieldspec creation, see Section 13.1 [Specifying a record field type], page 46.

integer
: Ordinary integers.

integer-or-nil
: Integers or `nil`, the empty value; by default, `nil` is treated as larger than any integer, so it comes last in an increasing-order sort.

number
: Ordinary numbers. A number is either an integer or a floating-point number.

number-or-nil
: Numbers or `nil`, the empty value; by default, `nil` is treated as larger than any number, so it comes last in an increasing-order sort.

boolean
: This displayspec corresponds to the yes-no displaytype. For the purposes of sorting, `t` is considered less than `nil`, so it appears first in a sort in increasing order.

string
: Ordinary strings.

one-line-string
: Strings which may not contain newlines.

string-or-nil
: Either a string or the value `nil`, which is converted to the empty string. Sorting treats `nil` identically to the empty string.

nil-or-string
: Identical to the string-or-nil recordfieldspec (except for the name). This exists so that display fields of type nil-or-string can conveniently default to this recordfieldspec.

one-line-string-or-nil
: The obvious combination of the one-line-string and string-or-nil recordfieldspecs.

date
: A date which specifies zero or more of the year, month, and day. By default the date is sorted by year, then month, then day; an unspecified component is larger than any specified component ("March 14, 1967" would appear before "January 1" if dates were sorted in increasing order). Dates are read from database files using the function value of `storage-string->date`, which is set by default to `date-stored->actual`, which can parse nearly any string representation of a date and returns a date if it is passed one. Dates are written using the function value of `storage-string->date`, which defaults

to `format-date-full`, which produces a string of the form "March 14, 1967". If the speed of reading and writing database files is very important to you, consider using `fset` to set `date->storage-string` and `storage-string->date` to more efficient functions, such as `date->storage-string-mmddyyyy` and `storage-string-mmddyyyy->date`, or `date->storage-string-lisp` and `storage-string-lisp->date`.

date-or-nil
: A date, or nil.

date-efficient-storage
: When the dates in a database file are known to have a particular format, using `parse-date-string` is unnecessarily inefficient. The date-efficient-storage recordfieldspec specifies that `storage-string->date`, which can efficently read dates written by `simple-format-date`, be used instead. The time savings is noticable on large databases.

time
: A time.

If you find any of these typenames cumbersome, you can create your own aliases for them using `define-displaytype-from-displayspec`, `define-recordfieldtype-from-recordfieldspec`, or `define-type-alias`.

`define-type-alias`
: Make symbol *alias* refer to the same displaytype and recordfieldtype as *typename*.

13.3 The recordfieldspec structure

The recordfieldspec structure contains information regarding the content of a record field, but nothing concerning how it is displayed on the screen or read from user input.

The slots of a recordfieldspec are listed below; a slot may be accessed by using the macro `recordfieldspec-`*slotname* and set using the macro `recordfieldspec-set-`*slotname*, whose second argument is the value to be stored in the slot.

Most of the slots may be left `nil`, and reasonable default actions will occur.

`type`
: A symbol such as `string` or `integer`, the type of the data described by this recordfieldspec.

  When no displaytype is explicitly specified in a display specification, then a displaytype with the same name as `type` slot is used by default; this is the only use for this slot.

`default-value`
: The default value for fields described by this recordfieldspec; used when creating new records.

`common-form-function`
: A function which, called on the contents of a record field, returns them in canonical form. This can be used for determining non-trivial equality, when two nonidentical values should be considered equivalent.

`merge-function`
: A function which, called on the contents of two record fields, returns a combination of the two. Often it queries the user for help.

`order-fn`
`sort-fn`
: The record field's ordering and sorting functions (see Section 7.1 [Sorting functions], page 30). Both ordering and sorting are possible if either slot is filled.

If both slots are empty, then a dummy ordering or sorting function is used, so sorting on this field has no effect. Since the function is called and its result examined, this is more expensive than not sorting on the field in the first place. If it does not make sense to sort on a particular field, then it is best to keep that field out of the field priorities used for sorting (which is the `field-priorities` database slot, or is interactively specified through the database sort interface (see Chapter 7 [Sorting], page 28), or is specified as an argument to `database-sort`).

Users may set the `order-fn` and `sort-fn` slots directly, but should use the following functions to access them:

`recordfieldspec-order-function`
: Return an order function for records described by *recordfieldspec*. If optional argument *reversep* is non-`nil`, then the order function goes in the opposite order. If the order-fn slot of the appropriate recordfieldspec of database doesn't contain one, one is made up on the fly from the sort-fn slot; `equal` is used to determine whether two records are equal. If the sort-fn slot is also empty, the resulting function always returns 0, indicating equality.

`recordfieldspec-sort-function`
: Return a sort function for records described by *recordfieldspec*. If optional argument *reversep* is non-`nil`, then the sort function goes in the opposite order. If the sort-fn slot of the appropriate recordfieldspec of database doesn't contain one, one is made up on the fly from the order-fn slot. If the order-fn slot is also empty, the resulting function always returns `nil`, indicating that it is not the case that the first argument is less than the second.

`match-function`

A function which takes a pattern and a field value and returns non-`nil` if they match. The function should also be able to take as its first argument a field value rather than a pattern.

`help-info`

A string which is displayed by `db-field-help` when there is no field-specific help available. Field-specific help is usually preferable to this help, which only describes the type of the field's contents.

`actual->stored`

A function which converts a field value into its on-disk representation (a string).

`stored->actual`

A function which recovers a field value from its on-disk representation (a string). If this function returns a string, it should return something reasonable if supplied the empty string as its argument. (That can happen when an empty database is read.)

`change-hook`

A function called when the value of this field is changed. This is not currently implemented. For more change hooks, see Section 16.2.5 [Display format change hooks], page 80, and see Section 16.2.6 [Recordfieldspec change hooks], page 82.

`constraint-function`

A function which the value of this field must satisfy; that is, the function must return non-`nil` on it. The function may reject the value either by returning `nil` or by signalling an error; the latter permits the function to provide an informative message about the problem.

Four arguments are supplied to `constraint-function`: the field value, the record, the record fieldnumber, and the database. This permits cross-field and cross-record constraints. The record argument may be `nil`, in which case the function should return `t` if the value is acceptable for some conceivable record. This occurs, for instance, when values are read in a call to `db-field-query-replace`.

The constraint function may interact with the user; for instance, it may give the use the opportunity to override the constraint.

14 Database file layout

This chapter discusses specifying how a database is read from a file (or saved back to it).

Broadly stated, there are three possible file representations for a database: EDB's internal file layout, a regular layout, or a nonregular layout. EDB's internal file layout is designed for fast reading and writing, but is not very human-readable. A regular layout is one in which records (and fields within a record) are separated from one another in a predictable (though not necessarily invariant) way. A nonregular layout is any other kind of layout; the user may specify arbitrary Emacs Lisp code to read and write such files. Support for tagged file layout (a special case of nonregular file layout) is included with EDB.

If the database is to be stored in EDB internal file layout, a lot of this information is not needed except when the database is first created.

The sections of this chapter each describe a file layout, except the last, which describes in detail the process of reading a database from disk.

14.1 Internal file layout

The first line of a database file in EDB's internal file layout looks something like

```
;; Database file written by EDB; format 0.3
```

followed by the printed representations of two Emacs Lisp forms, a record (the database structure) and a list of records (the records of the database). Databases stored in this layout can be loaded and saved very quickly (sometimes orders of magnitude faster than databases which EDB must parse when reading), and they never suffer from ambiguities between data and delimiting text, but they are not easy for people to read and understand. A human- or program-readable version of the database can be generated when it is needed, either by creating a report or by saving in some other file layout. This is a good option when all manipulation of a database will be done via EDB.

Since this file layout is rather complicated, databases are often created in some other file layout and then converted to this one. To convert from another file layout to EDB's internal file layout, read in the database, set database slot `internal-file-layout-p` to `t` (for instance, via `C-u M-x db-toggle-internal-file-layout RET`), and then write or save the database (via `C-x C-w` or `C-x`

`C-s`). Convert a database from EDB file layout to some other representation is similar, but slot `internal-file-layout-p` is set to `nil` (say, via `C-u 0 M-x db-toggle-internal-file-layout RET`), and certain variables and database values may need to be set (see the documentation for the layout you desire, elsewhere in this chapter). Making a report can also convert a database to a different file layout, with even more flexibility than the techniques described here.

A database file in EDB internal file layout is basically the printed representation of the Lisp database structure used by EDB when the database is read in. As such, it contains all the information in the database slots described in Section 17.1 [The database structure], page 85, except that the `data-display-buffers` and `first-link` slots are set to `nil`; in the file, the records follow the database structure. After a database has been saved in internal file layout, then any forms in the auxiliary or format files that set these slots can be removed if desired; this is not necessary, however.

14.2 Regular file layout

EDB can conveniently read and write database files in which records are separated from one another by a record delimiter and, within each record, fields are separated by a field delimiter. When the delimiters are the newline and tab characters, respectively, the result is the standard "tab-separated text" layout, which is often used for transferring information from one program to another.

The record and field delimiters need not be single characters; they can be arbitrary strings or can even be specified by a regular expression instead of a particular string. The latter is useful if the exact delimiter is not known ahead of time (for example, if records may be separated by one or more carriage returns). This regular expression mechanism can only be used when reading the database: when writing a database, all the record and field delimiters will be identical. (Exception: you may specify an arbitrary record-writing function (see Section 14.4 [Nonregular file layout], page 60) and arbitrary functions for either reading records or for separating records or fields in regular layout, but should use the simpler reading mechanisms whenever possible, for your own sake.)

Regular file layout has two disadvantages. First, it is somewhat slower to read and write than EDB's internal representation. Second, the strings used as delimiters may not appear in the database fields, lest those occurrences be misinterpreted as delimiters rather than as data. EDB provides two solutions to the latter problem: substitution and quoting (see Section 14.2.2 [Resolving ambiguities], page 56).

14.2.1 How to specify regular file layouts

In a database stored in regular file layout, records and fields can be separated by particular strings, by regular expressions, by context-sensitive regular expressions, or by arbitrary functions. The sepinfo structure holds this information for use when reading the database from disk (and writing it back). The sepinfos used when reading a database are stored in its `record-sepinfo` and `field-sepinfo` slots (for more details about the database structure, see Section 17.1 [The database structure], page 85).

When reading, if a separation function is specified, it is used; otherwise, if a regular expression is specified, it is used; otherwise, a string must be specified. It is converted into a regular expression, the regular expression slots of the sepinfo are filled in, and reading proceeds as if the user had specified a regular expression. (A user who wishes to have a regular expression recomputed when it is next needed should set it to `nil` when setting the corresponding string value.)

When a sepinfo is used for writing, it must specify literal string separators. (The sepinfo may have its separation function or regular expressions set as well, for reading, but those slots are ignored when writing.)

14.2.1.1 The sepinfo structure

The sepinfo structre contains the information required to decide where records or fields start and end (actually, to determine where the record or field separators start and end; "sepinfo" is short for "separator information"). The slots of this structure may be accessed by using the macro `sepinfo-`*slotname*. The slots may be set using the macro `sepinfo-set-`*slotname*, whose second argument is the value to be stored in the slot.

The `pre-first-` slots describe text that precedes the first item of interest. In a record sepinfo they describe the file header, which precedes the first record. In a field sepinfo they describe any information that preceded the first field of every record, after the record separator.

The `post-last-` slots are similar, but are used to inform EDB of text following the last information-carrying text. In a record sepinfo, they describe the file trailer, which follows the last record in the file. A field sepinfo's `post-last-` slots tell about information following the last field of a record but preceding the record separator.

The `-submatch` integers describe which submatch of a regexp match is the actual separator, as opposed to surrounding text used to help make the match unambiguous. This specification of the submatch permits context-sensitive matching that you might otherwise expect could not be done with regular expressions alone. For instance, suppose a database has records with a variable number of fields separated by newlines, that records are also separated by newlines, and that the first field of each record has some special form different from all other fields (say, it is a number with a decimal part). The following code would permit separation of the records without writing a special function to do so and without including the decimal number in the separating text:

```
(sepinfo-set-sep-regexp (database-record-sepinfo database)
                        "\\(\n\\)[0-9]+\\.[0-9]+")
(sepinfo-set-sep-regexp-submatch (database-record-sepinfo database) 1)
```

When you set the slots of the sepinfo, be careful to use a correct value. For instance, if your record separator is a form feed on a line by itself, you probably want to set the `sep-string` slot of the database's `record-sepinfo` to ‘`"\f\n"`’, or possibly ‘`"\n\f\n"`’, rather than just ‘`"\f"`’, lest the newlines be considered to be part of the records rather than part of the separator.

The slots of the sepinfo are listed below but are not described in detail; see the preceding description for details of their use.

`pre-first-string`
: Setting the slot to `nil` (or not setting it) is equivalent to setting it to the empty string.

`pre-first-regexp`

`pre-first-regexp-submatch`

`sep-string`

`sep-regexp`

`sep-regexp-submatch`

`sep-function`
: A function that takes a buffer position, the end of the previous separator (that is, the start of the current record or field), as its argument and returns a pair of two buffer positions bracketing the next separator. That is, the returned values are the end of the current record of field and the beginning of the next one (or nil if there are no more). When the function is called, point is at the beginning of an item and the buffer is narrowed to the list being currently processed.

: The use of a separation function is useful when the separation criteria cannot be expressed as a combination of regexp expressions. The `pre-first-` and `post-last-` slots are still used even if a function is specified.

`post-last-string`
: Setting the slot to `nil` (or not setting it) is equivalent to setting it to the empty string.

`post-last-regexp`

`post-last-regexp-submatch`

14.2.1.2 Examples of setting record and field separators

[[[Put examples here.]]]

[[[For instance, to parse "[Mary, John,Jack, and Jill]" and to write it back out as "[Mary, John, Jack, Jill]", the following specification would suffice: pre-first-string "[" sep-string ", " sep-regexp ", +\\(and +\\)?" sep-regexp-submatch 0 post-last-string "]"]]]

[[[The -string slots are used for writing; but what if you only have a regexp for the leading or trailing junk, but you want that restored exactly? You can set pre-first-string *after* the database file has been found. For instance, in db-before-read-hooks, use a function such as

```
(defun btxdb:read-comments ()
  (save-excursion
    (set-buffer db-buffer)
    (goto-char (point-min))
    (if (search-forward "@" nil t)
(sepinfo-set-pre-first-string
 (database-record-sepinfo database)
 (buffer-substring (point-min) (point))))))
```

or even put

```
(sepinfo-set-post-last-string
 (database-record-sepinfo database)
 (save-excursion
   (set-buffer db-buffer)
   (goto-char (point-min))
   (re-search-forward "\n\C-l\n")
   (buffer-substring (match-beginning 0) (point-max))))
```

as is in your auxiliary file.]]]

[[[If all records have the same number of lines on disk, use the following function to return an appropriate sep-function. This is useful when, for instance, both the field separator and the record separator are the newline character.

`make-n-line-sep-function`

Return a sep-function useful when all records have exactly *n* lines on disk.

]]]

14.2.2 Resolving ambiguities

Substitution and quoting are two mechanisms for dealing with the problem of distinguishing field and record separators from the contents of database records. For instance, if the newline character (actually, a string consisting of only the newline character) is used as a record separator, and records may contain multiline text fields (or other fields whose storage representation contains a newline), then how would EDB know, when reading the database back in, which newlines are record separators and which are part of fields?

There are several ways to avoid this ambiguity.

- Disallow the use in record fields of the character or string causing the ambiguity. For instance, in the example above, you might change the record field type of all of the string fields to one-line-string.
- Change the separator(s) to strings that do not appear in the storage representation of any field. For instance, when reading a Unix password file, colons should not appear in the field text, so a colon can be used as the field separator, like so:

  ```
  (sepinfo-set-sep-string (database-field-sepinfo database) ":")
  ```

  Strings containing non-printing characters are another good bet, but this method relies on luck and the hope that the chosen separators will never appear in data.
- Change the representation of the ambiguous string, when it appears in data; this guarantees that whenever the string does appear in a database file, it stands for a separator. This scheme is called substitution, because another string is substituted for the ambiguious one when it appears in data. This is similar to the previous workaround, which changed the separators rather than the data-bearing instances of the string. Ambiguities are still possible, if the substituted text happens to appear elsewhere in data. Specifying a substitution is described below.
- When all of the previous methods are insufficient, the more powerful quoting mechanism can be used. It works similarly to the quoting mechanisms of programming languages that permit specification of strings which contain the character usually used to delimit string constants. It permits any strings to be used in separators and also to appear in data, but it slightly increases the size of the data and slows down reading a writing. Specifying that quoting be performed is described below.

- The simplest solution is to use EDB's internal file layout (see Section 14.1 [Internal file layout], page 51). Ambiguities can only occur when the field data and the separators are both text to be interpreted by EDB. EDB's internal file layout uses Emacs Lisp's mechanisms (a form of built-in quoting) to ensure that what is read in is identical to what was written out. The database designer need not worry further about the problem.

[[[Describe substitution in detail.]]]

Substitution is the replacement of potentially ambiguous strings by other ones. For instance, when writing tab-separated text, each occurrence of the newline character in a field could be replaced by control-k when the database is written. Then, when the file is read in, every newline can be safely assumed to be a record separator. The final step is converting the control-k characters back into newlines. This approach is taken by some marketed databases; for instance, I believe that FileMaker does just this. The problem with this approach is that if there were any control-k characters in the text, then when the database is read back in, they will be (incorrectly) converted to newlines. EDB warns when the database is being written out if this problem could occur; the user is given the option of choosing a different substitution or of aborting the database write operation. It is usually possible to find a substitution—a character or sequence of characters that doesn't appear in the data.

[[[for instance, put

eval: (database-set-substitutions database '(("\n" . "\C-k")))

in the "Local Variables" section of your format file. Or probably just put a short blurb here and put a longer one after the list.]]]

[[[Describe quoting in detail.]]]

Quoting is...

14.2.3 Problems with end-of-file newlines

Here is a subtle problem which can come up if you use '`"\n\n"`' as a record separator and exactly one newline appears at end of your database file. For convenience, EDB adds a record separator at the end of its working copy of the file, if there's not one already there. In this example, two newlines will be added, but then the file will still not end in a record separator, since after finding the first pair of newlines after the last record, EDB won't yet be at the end of the file because there will still be another character (namely, `\n`) there.

The moral is that if there are any extra characters after the last record, even a newline, they should be specified. Either of the following forms will do the trick:

```
(sepinfo-set-post-last-string (database-record-sepinfo database) "\n")

(sepinfo-set-post-last-regexp (database-record-sepinfo database) "\n\\'")
(sepinfo-set-post-last-regexp-submatch (database-record-sepinfo database) 0)
```

The '`\\'`' is not strictly necessary in this example.

[[[Maybe I should special-case this; i.e., replace end-of-file test with end-of-file-or-only-newline-remaining test? It comes up pretty frequently.]]]

Here is an even more subtle problem: suppose that you want to get rid of every newline at the end of the database file, but you don't know how many there are. Using '`"\n*\\'"`' in place of '`"\n\\'"`' above will not work, because the post-last-record regexp is searched for backward from the end of the buffer, and (because of the way that `regexp-search-backward` is implemented) the backwards regexp match for '`\n*`' is always the empty string! The proper way to write this would be

```
(sepinfo-set-post-last-regexp (database-record-sepinfo database)
      "[^\n]\\(\n*\\'\\)")
(sepinfo-set-post-last-regexp-submatch (database-record-sepinfo database) 1)
```

14.3 Tagged file layout

Another popular file layout supported by EDB is that of field values preceded by the fieldname. For instance, a record might be represented in the file by

```
Where:Here
When:  Now
What: This!
```

which indicates a record in which the 'where', 'when', and 'here' fields have the specified values.

Tagged files are a special case of files in nonregular layout; support for them is implemented through the mechanisms described in Section 14.4 [Nonregular file layout], page 60.

To read a database file in tagged format, call the function `db-tagged-setup` in the database's format or auxiliary file. Its argument specifies the names of the fields and the tags that precede them in the database file.

`db-tagged-setup`

: Ready the database to read files in tagged format. Creates database local variables and sets database slots. Argument *tagged-field-specs* is a list of tagged-field specifications, one for each field in a database record. Each tagged-field specification is a three-element list of the field name (a symbol), the tag used to identify it in the file (a string), and a brief help string. Instead of a symbol, the tagged-field name may be a cons of the field name and its type. To indicate that a field is never found in the input file (typically because it is computed on the fly), use `nil` for its tag.

  This function should be called first in an auxiliary or format file, so that the defaults it chooses can be overridden. `database-set-fieldnames-to-list` should not be called if this function is.

Calling this function sets the database's field names and installs appropriate functions for reading and writing the database. It also creates some database local variables (see Section 16.3 [Local variables], page 82) which can be modified (by use of the `database-set-local` function) in order to customize the behavior of the parsing and output functions with respect to what characters can appear in a tag, what the separator between tag and value looks like, and how continuation lines are handled. By default, records are separated by blank lines, tags are separated from field values by '`:`', white space around the separator is not significant on input, the separator is followed by one tab on output, and continuation lines start with whitespace.

`db-tagged-tag-chars`

: The characters that are allowed in field tags, in a form suitable for placing inside [] in a regular expression.

`db-tagged-separator`

: The string that separates field names from values. Used only if db-tagged-separator-regexp or db-tagged-separator-output is `nil` (depending on whether the record is being read or written).

`db-tagged-separator-regexp`

: A regexp for the separator between field names and values when parsing.

`db-tagged-separator-output`

The separator between field names and values on output.

`db-tagged-continuation`

The string that marks (the beginning of) a continuation line. Used only if db-tagged-continuation-regexp or db-tagged-continuation-output is `nil` (depending on whether the record is being read or written).

`db-tagged-continuation-regexp`

A regexp for a continuation line in a value when parsing.

`db-tagged-continuation-output`

The fixed string to use (before) continuing values on output.

Other hooks permit arbitrary manipulations of records; for instance, if a database nearly conforms to the tagged file model, these can be used to customize the behavior of the existing tagged code. One way to do this is to have a function in `db-tagged-rrfr-hooks` remove the field from the file representation before the record is parsed, then have `db-tagged-wrfr-after-hooks` modify the automatically generated tagged file representation for that field. These functions can also be used for simpler tasks, of course.

`db-tagged-rrfr-hooks`

Hooks run on each database record before tagged parse.

`db-tagged-wrfr-before-hooks`

Hooks run before each tagged write of a database record. The record is bound to the dynamic variable record, and point is where the record will be inserted in the buffer.

`db-tagged-wrfr-after-hooks`

Hooks run after each tagged write of a database record. The record is bound to the dynamic variable record, and point is immediately after the file representation of the record.

14.4 Nonregular file layout

Unlike many databases, EDB can work with data stored in any file layout whatever—so long as you specify how the information is to be extracted. If the file layout is too complicated to be described by regular expressions describing the record and field separators and their context (see Section 14.2 [Regular file layout], page 52), then you may write Emacs Lisp code which extracts the information from the database file.

The great advantage of this mechanism is that it permits you to maintain your current files, in exactly their current file layouts, and to keep the same tools and habits you've accumulated with respect to them, but also to manipulate them in a structured way with EDB when necessary. For instance, you might wish to maintain the database file in a file format easy for people to read all the time, rather than having to create a report for that purpose.

Three pieces of information must be provided: how to find the extent of a file record, how to read a file record, and how to write a file record. The third may be omitted if the database is only being read in the custom file layout (and will be saved in some more tractable file format). If the second is provided (that is, the `read-record-from-region` database slot is set), then the file will be assumed to be in a nonregular file layout and the value of that slot will be used to read the database, no matter what other information is provided.

Information about how to separate one record from another within the file is found in the `record-sepinfo` slot of the database, as usual. In many cases, even if the file layout of the data is nonregular, it will be easiest to describe the record separator with a string or a regexp. For more details, see Section 14.2 [Regular file layout], page 52. You may also set the sepinfo's `sep-function` slot to a function. The function should take one argument, the end of the previous record (`nil` the first time it's called), and return a pair whose car is the end of the current record and whose cdr is the start of the next record (`nil` if there is no next record in the file).

The `read-record-from-region` slot of the database contains a function of no arguments which, when called with the current buffer narrowed to a single file record (that is, narrowed to the representation of a single database record), returns a record in the database's internal file layout. The variable `database` is dynamically bound to the current database, and so the right way to create the record to be returned is via `(make-record database)`. Its fields can then be set with `record-set-field`.

The `write-region-from-record` slot of the database optionally contains a function which takes a database record as its argument and inserts the file representation of that record in the current buffer; the variable `database` is dynamically bound to the current database. If this slot is not specified (and slot `internal-file-layout-p` is `nil`), then the fieldsep and recordsep information, if present, will be used to write the record (see Section 14.2 [Regular file layout], page 52). This permits the use of a simple, regular output file layout with a more flexible input file layout.

Tagged format is a special case of nonregular file layout for which EDB provides support; see the implementation of support for tagged database files in '`db-tagged.el`' and Section 14.3 [Tagged file layout], page 58. Another example is given below.

14.4.1 Example of database in nonregular file layout

Here is a simple example of a database in a nonregular file layout; this does not mean that the file representation of each record is vastly different from the others (it may be, but is not in this instance), but that there is no regular rule for extracting field values from the record.

Suppose we had a database of the following form:

```
Place:   Dentist's Office
Time:    Never!
Purpose: Root canal

Place:   Home
Time:    Midnight
Purpose: Sleep

Place:   Other places
Time:    Other times
Purpose: Other things
```

Then in order to read and write this database, we place the following code in the auxiliary file (see Section 14.5 [Reading from disk], page 63):

```
(sepinfo-set-sep-string (database-record-sepinfo database) "\n\n")
(database-set-read-record-from-region database 'arb-demo-rrfr)
(database-set-write-region-from-record database 'arb-demo-wrfr)

(defun arb-demo-rrfr ()
  (goto-char (point-min))
  (if (re-search-forward
       "Place:[ \t]*\\(.*\\)\nTime:[ \t]*\\(.*\\)\nPurpose:[ \t]*\\(.*\\)")
      (let ((result-record (make-record database)))
        (record-set-field result-record 'place (match-string 1) database)
        (record-set-field result-record 'time (match-string 2) database)
        (record-set-field result-record 'purpose (match-string 3) database)
        result-record)
    (error "This didn't look right to me.")))

(defun arb-demo-wrfr (record)
  (insert "Place:   " (record-field record 'place database)
          "\nTime:    " (record-field record 'time database)
          "\nPurpose: " (record-field record 'purpose database)))
```

The auxiliary file would also specify the database's fieldnames:

```
(database-set-fieldnames-to-list database '(place time purpose))
```

as well as possibly other information such as the summary format or the name of the default format file. See the example database auxiliary file '`arb-demo.dba`' for a concrete example of this.

All this Emacs Lisp code may be placed in "Local Variables" section of the format file instead of in the auxiliary file, if desired. For more information about the "Local Variables" section of a file, section "Variables" in *The GNU Emacs Manual*.

This particular example is simple enough that a special function for reading isn't strictly necessary. Reading can be done under the control of regular expressions; for instance, each field separator would be `"\n[^:]*:[ \t]*"`. See the example database auxiliary file '`arb-demo-regexp.dba`' for a concrete example of this. Or, you could just use EDB's support for database files in tagged file layout, which is exactly what this is; see Section 14.3 [Tagged file layout], page 58.

14.5 What happens when a database is read in from disk

In brief, the following happens after you execute `db-find-file`:

1. If the database is already read in and its buffer has not been killed, the buffer is simply selected. No other work is done.
2. Otherwise, the database file is inserted in a special buffer of its own. If the database is in EDB internal file layout (that is, if an identifying header is found), it is read in immediately. Otherwise, a new, empty database is created. In either case the dynamic variable `database` is bound; this makes it possible to refer to the database in the auxiliary and format files (even before it has been read in, if it is not in EDB internal file layout).
3. The format file is found (see Section 16.1 [Auxiliary files], page 75), and the data display buffer is created.
4. The function `db-setup-format` is called; it performs the rest of the work necessary for setting up the data display buffer (everything up to the running of `db-before-read-hooks`). Its first action is to insert the format file's contents into the data display buffer.
5. The auxiliary file, if any, is loaded. This happens in the data display buffer, and the dynamic variable `database` and the buffer-local variable `dbc-database` are bound to the current database. For more information about how the auxiliary file is found and what it can do, see Section 16.1 [Auxiliary files], page 75.

The auxiliary file is not read every time that `db-setup-format` is called, only when a database's primary display format is read. (The primary display format is the one initially selected when a database is first read in.)

6. The "Local Variables" section, if any, of the format file is executed; this may set variables and execute Emacs Lisp code, exactly analogously to the auxiliary file. EDB ignores the value of `inhibit-local-variables` when evaluating this code. This section is then deleted from the working copy of the file, so that it does not appear in the data display buffer when a user is viewing database records. For more information about the "Local Variables" section of a file, see section "Variables" in *The GNU Emacs Manual*.
7. Database information is propagated; for instance, the names of the database fields are known by now, and various other database slots are filled in depending on this information, if they haven't been set yet.
8. The format file is parsed, and literal text and formatting directives are distinguished from one another. This work is done by the `db-setup-format-parse-displayspecs` function. When that function is done, `db-setup-format` returns the database data display buffer as its result.
9. The hooks in `db-before-read-hooks` are run in the data display buffer.
10. If the database had already been read because it was stored in internal file layout, it is massaged a bit to get it into its final form; for instance, the backward links are added between adjacent records.

    Otherwise, the database is finally read; the values of the `recordsep` and `fieldsep` slots of the database determine whether the layout is regular or nonregular and direct the parsing. The `substitutions` slot and quotation slots (`quotation-char`, `quotation-char-regexp`, `quoted-regexp`, and `quoted-strings`) direct replacement of characters that could not be written into the file, and the `stored->actual` slot of each recordfieldspec completes the translation to the data's internal format from its file layout.
11. The hooks in `db-after-read-hooks` are run in the data display buffer.
12. The database has now been read and is in its final form. The first record of the database is displayed in the data display buffer, which is then placed in view mode and selected (made visible to the user).

15 How information is displayed

The display of information, both on the screen (whether in the data display buffer, the summary buffer, or elsewhere) and in other output (such as reports), is controlled by formatting commands. We will discuss a data display buffer by way of example; the formatting specifications are the same for summary buffers and reports as well.

Display types should not be confused with record field types; a display type is used to specify how a particular value is shown on the screen, but a record field type constrains the information actually contained in the record field. This chapter does not discuss record field specifications, which specify everything about a record field type except how it is displayed and parsed in output intended for humans to read. For more information about that, and about the distinction between record field types and displaytypes (the latter of which is described in this chapter), see Chapter 13 [Record field types], page 45.

A *display format* gives all of the information necessary to create a data display buffer; it consists of literal text that is displayed as is (and may not be edited by a user of the database) and of *display specification*s that instruct EDB how to display a particular field's contents. The display specifications do not appear in the data display buffer; they are replaced by fields' values, which may or may not be editable. An example of a display specification is '`\name,width=16`', which indicates that the 'name' field of the database should be displayed (after being padded or truncated to exactly 16 characters).

When a format is first specified, it is parsed and the formatting information specified in the display specification strings is used to create a displayspec structure. Users should never have to manipulate displayspecs directly.

15.1 Display specifications

Here is a (quite complicated) example display format:

```
\name,one-line-string,actual->display=upcase\ , \occupation'
Pay:     \\\salary,min-width=4:  too much!
Address: \addr,indent is home sweet home
```

This display format is valid if the database contains fields called "name", "occupation", "salary", and "addr"; any other fields are ignored. Some typical records would be displayed like this:

```
JOHN DOE, butcher
Pay:     \  22:  too much!
Address: 123 Main St.
         Anyplace, USA is home sweet home


JANE ROE, baker
Pay:     \4444444:  too much!
Address: 675 Massachusetts Avenue is home sweet home
```

A display specification consists of a backslash; a fieldname indicating which field of the database record is to be inserted; optional comma-separated type and formatting information; and optionally a backslash followed by a space. A display specification abbreviation, which consists of a backslash and the abbreviation name (and so looks like a simple display specification in which no optional information is specified), can be used instead of a standard display specification; Section 15.6 [Display specification abbreviations], page 74.

A display specification starts with a backslash to distinguish it from the surrounding text. To specify the backslash character in literal text, type it twice; when backslashes occur in pairs they do not indicate the start of a display specification. Display specifications continue as long as the text can be parsed as one. Almost any non-whitespace character may be used as part of a display specification. Whitespace is a good way of indicating the end of a field specification since it never appears in a display specification. A display specification can be terminated without indicating any literal text by placing a backslash and a space at its end; both of these characters will be ignored. This is useful when the literal text that follows the display specification happens to conform to the display specification syntax (is a letter, number, or almost any type of puntuation).

The optional information includes the type of this display field and formatting directives for it; if the type is present, then it must come first among the displayspec's optional specifications. Each optional parameter is preceded by a comma to separate it from the preceding one (or from the fieldname, for the first optional parameter). The optional information is typically of the form "*slotname*=*value*", which sets the specified slot to the given value, or "*slotsetter*", which sets some slot to a particular value. Explicitly specified formatting information overrides any defaults.

The display type can be specified by writing a typename (such as '`string`' or '`integer`') as the first optional parameter. A type abbreviation may be used instead of a typename; the defined type abbreviations are '`#`' for integer, '`$`' for number, '`"`' for string, and '`'`' for one-line-string, and they need not be preceded by a comma, though they may be.

The display type specifies default values for the display specification (actually for the displayspec structure, which is derived from the display specification). It is rarely necessary even to specify the displaytype—most display specifications consist of simply a backslash and a fieldname—since if the displaytype is omitted then a displaytype with the same name as the record field type (actually the `type` slot of the recordfieldspec) will be used. This works because typically displaytypes and recordfieldtypes with the same names and complementary definitions are declared at the same time; displaytypes and recordfieldtypes are conceptually distinct, however. In particular, you must specify a displaytype that is compatible with the record field type; if you specify a displaytype of '`integer`' when the data is actually a string, an error will result. You can use function `database-set-fieldnames-to-list` to specify recordfieldtypes; see Section 17.1 [The database structure], page 85.

15.2 Predefined displaytypes

The file '`db-types.el`' defines the following displaytypes, corresponding recordfieldtypes, and some useful associated functions. Users can also define displaytypes; see Section 15.4 [Defining new displaytypes], page 70.

integer
: Ordinary integers.

integer-or-nil
: Integers or `nil`, the empty value; `nil` is formatted as the empty string.

number
: Ordinary numbers. A number is an integer or a floating-point number.

number-or-nil
: Numbers or `nil`, the empty value; `nil` is formatted as the empty string.

yes-no
: This displayspec corresponds to the boolean recordfieldtype. The field is three characters long and contains "Yes" or "No ".

string
: Ordinary strings. By default there is no maximum or minimum width or height, and subsequent lines are indented relative to the first character of the first line.

one-line-string
: Strings which may not contain newlines.

string-or-nil
: Either a string or the value `nil`, which is displayed as the empty string.

nil-or-string
: Either a string or the value `nil`. When the user enters the empty string as the field value, or when a new record is created, the value `nil` is used in preference to the empty string.

one-line-string-or-nil
: Both a one-line-string and a string-or-nil.

date
: A date which specifies zero or more of the year, month, and day. The date is formatted by `format-date` and parsed by `parse-date-string`; see below for details.

time
: A time which specifies zero or more of the hour, minute, and second. The time is formatted by `format-time` and parsed by `parse-time-string`; see below for details.

15.2.1 Date displaytype

EDB defines a date abstraction and a variety of useful operations upon it; the best way to learn about these features is to read '`db-time.el`'

This section provides more detail about the date displaytype. A date specifies a year, month, and day (all integers); any or all of these components may be omitted. Dates are created by the constructor `make-date` and a date's components are retrieved using the selectors `date-year`, `date-month`, and `date-day`.

`make-date`
: Make an EDB date object with arguments *year month day*.

[[[Document parse-date-string, format-date, simple-format-date, def-xxx-type.]]]

[[[I added several useful (to me, anyway) displayspecs for various date types. These are meant to be used in a display spec, like: \datefield,date-mmddyy The displayspecs are implemented with similarly named formatting functions, which I also implemented. All of the new formatting functions are named format-date-XXX, where XXX are the various styles.]]]

15.2.2 Time displaytype

[[[Similarly to the above, for times.]]]

15.3 Enumeration types

An enumeration displaytype is used for fields whose values are one of a fixed set of alternatives. Each alternative may be a single character (say, M or F for gender) or specifyable by a single character (for example, if the first letters of the alternatives are unique); the user need only type a single character in order to select one of the alternatives. Another possiblity is that each alternative consists of an entire string entered with completion. (The string may consist of only a single character if desired, but the user must still type `RET` after entering the string.) The internal representation of the data—its recordtype—need have nothing to do with the way that the alternatives are specified. The next two sections describe the two types of enumeration displaytypes, which are nicknamed one-char-enum and (for the multicharacter alternative type) enum.

15.3.1 One-character enumeration displaytypes

One-character enumeration displaytypes are not yet implemented.

`define-one-char-enum-displaytype`

Not documented.

15.3.2 Multi-character enumeration displaytypes

Multi-character enumeration displaytypes require a user to enter an entire string in order to specify one of the alternatives. This typing may be done with completion in the minibuffer, meaning keys such as `TAB` and `?` will complete a partly-entered choice or list the remaining possibilities. (For more about completion, see section "Completion" in *The GNU Emacs Manual*.) The internal, input, display, and file storage represenations of the value may all be different. Multi-character enumeration displaytypes (also known as enum displaytypes) are created by calling the following function, which also creates a corresponding recordfieldtype.

`define-enum-type`

Make *typename* (a symbol or string) an enumerated type. Both a displaytype and a recordfieldtype are created.

alternatives is a list. Each alternative is a list of up to four components: the internal representation, any constant Lisp object, often a string; the input representation typed by the user to specify this alternative, a string or list of strings (for multiple input representations); the display representation, a string; and the file storage representation, a string.

If the input representation is omitted and the internal representation is a string, that string is used. If the display representation is omitted, it defaults to the first input representation. The display representation is automatically also a valid input representation. If the file storage representation is omitted, it defaults to the display representation. If all the other components are omitted, the internal representation string may be used in place of a one-element list containing just it.

Optional argument *optstring* is a displayspec option string.

When a record field's type is an enum type, both EDB and code written by the database designer may assume that the value in the record field is one of the valid representations. (Similarly, when a field's type is string, EDB can assume that the field content is actually a string.) This means that the empty string, `nil`, and other special values must be specifically mentioned when the enumeration type is defined. Here is a way to define an enumeration type which is either a day of the week or the empty string:

```
(define-enum-type 'workday
  '("Monday" "Tuesday" "Wednesday" "Thursday" "Friday" ""))
```

If it is possible for the field value to be `nil` (but not the empty string) after reading the database, and `nil` should be displayed as "Unknown" (and that string parsed into a value of `nil`), the following definition suffices:

```
(define-enum-type 'workday
  '("Monday" "Tuesday" "Wednesday" "Thursday" "Friday" (nil "Unknown")))
```

15.4 Defining new displaytypes

When you are about to type a complicated display specification—or a simple one more than once—consider defining and using a displaytype instead. Displaytypes are more concise (and so less cumbersome and less error-prone), easier to change (since a change to the displaytype can affect every display specification that uses it), and clearer (since a descriptive typename makes immediately clear what the intention is). Furthermore, displaytypes can be built up incrementally, with each one making a few changes to those from which it inherits defaults.

There are two ways to define a new displaytype; each requires specifying the name of the displaytype and some formatting information to be associated with that displaytype.

The first method permits a displaytype to be specified by the optional part of a display specification, which is a string consisting of comma-delimited optional parameters. The first optional parameter may be a type, in which case the defaults for values not explicitly set in the other parameters are taken from that type.

`define-displaytype-from-optstring`

Define a displaytype named *typename* according to *optstring*. *typename* is a symbol or string and *optstring* is the optional parameters part of a display specification string.

The second method is more useful for complicated displaytypes; it is also somewhat more efficient, which is why the file '`db-types.el`' uses it to define the standard predefined types. This method is to create a displayspec directly, modify it as desired using the structure slot modifiers (i.e., `displayspec-set-`*slotname*; see Section 15.5 [Display specification optional parameters], page 71, for a list of slotnames), and then associate a typename with the displayspec. In fact, this is precisely what `define-displaytype-from-optstring` does.

`define-displaytype-from-displayspec`

Define a displaytype named *typename* (a symbol) with the default *displayspec*. *displayspec* may also be a typename symbol itself.

`make-displayspec`

Create and return a new displayspec.

`displaytype->displayspec`

Return a copy of the displayspec corresponding to string or symbol *displaytype*. Return `nil` if there's no corresponding displayspec.

15.5 Display specification optional parameters

This section describes the display specification optional parameters, which correspond exactly to slots of the displayspec, EDB's internal representation of the display specification.

Optional display specification parameters are separated only by commas; display specifications never contain whitespace. These parameters are of two forms: slotsetters, which are a single word and set a slot to a particular value; and slot assigners, which are of the form "*slotname*=*value*" and set the slot to the value. Unless otherwise specified, each slot can be set by a slot assigner whose name is the same as that of the slot. An example of a display specification containing two optional parameters, one a slot assigner and one a slotsetter, is '`\name,width=16,unreachable`'.

Display specification fields are processed in order, so only the last instance of a particular parameter has any effect. Any explicitly specified parameter overrides defaults, values inferred from the type, or previously specified parameters.

If you find yourself repeatedly writing similar display specifications, or large, bulky display specifications, consider defining a new type to do some or all of the work for you; see Section 15.4 [Defining new displaytypes], page 70.

`record-index`
: This integer is the field index in a database record of the value formatted by this displayspec. This is set by looking up the fieldname part of the displayspec.

`indent`
: This a boolean value determines whether the second and subsequent lines should align with the beginning of the first one or should be flush left, in column 0. It is set and unset with the `indent` and `noindent` slotsetters. The first of the following displays has indent set, and the second does not:

```
Name:    John Doe
Address: 123 Main St.
         Anyplace, USA
```

```
Name:    John Doe
Address: 123 Main St.
Anyplace, USA
```

This causes alignment of the first character of subsequent lines with the first character of the first line; it does not do anything clever with whitespace in the field value, nor does it align different lines differently.

`min-width`
`max-width`
: These integers are the minimum and maximum widths which the display may occupy. If the formatted value is too short, the function in the `padding-action` slot is called to lengthen and/or justify it. If the formatted value is too long, the function in the `truncation-display-action` slot is called to shorten it; if that slot is empty, the field is simply truncated. The `width` slot assigner sets both the `min-width` and `max-width` displayspec slots. The `min-length`, `max-length`, and `length` slot specifiers are synonyms for the `min-width`, `max-width`, and `width` slot specifiers.

`min-height`
`max-height`
: These are analogous to `min-width` and `max-width`, but for the number of lines occupied by the formatted value (actually, the number of newlines in the string, plus one). There is a `height` slot assigner which sets them both.

`truncation-display-action`
: This function helps reduce the size of the formatted value when it is too large to fit in the specified displayspec size. It defaults to simply truncating the formatted field to the maximum permissible size. It may also be set with the `trunc-display` slot assigner. At present, this function is only called if the formatted value is too wide; there is no analogous function called when it is too tall.

`padding-action`

This function determines how a field that is too small for the displayspec (that is, the printed representation contains fewer characters than specified in the `min-width` slot) should be expanded to fit. The padding function takes three arguments: the minimum length, the unpadded display representation, and the length of that representation.

The `padding-action` slot may also be set to a cons of a padding character and a padding direction: `nil` for left-justification (padding on the right), and non-`nil` for right-justification. (You cannot set the `padding-action` displayspec slot to a cons by using a display specification, since display specifications may not contain whitespace, so the easiest way to right-justify a single field is to use the `right-justify` slotsetter.) The default, which can also be obtained just by setting the slot to `nil`, is to pad on the right with space characters.

`actual->display`
`display->actual`

These functions actually do the work of converting between the data's internal representation and its displayed representation (a string). Other functions (such as those in the `truncation-display-action` and `padding-action` slots) may then be called on the result returned by the `actual->display` function. These slots may be set with the `a->d` and `d->a` slot assigners.

The `display->actual` function takes either one argument or four arguments: either just the field text or the field text, the previous field value, the record being operated upon, and the record fieldnumber of the field in question. EDB ascertains at runtime how many arguments the function should be applied to. The old field value is passed in case it contains hidden (undisplayed) attributes that need to be preserved across changes. The other two arguments permit a particular `display->actual` function to be used for more than one field of a record, allow the field text parse to depend on other record field values, and provide for other complicated needs. Most `display->actual` functions can be specified to take a single argument.

The `actual->display` function takes either one argument or three arguments: either just the field value or the field value, the record, and the record fieldnumber. EDB ascertains at runtime how many arguments the function should be applied to. The reasons the additional arguments may be specified are similar to those outlined above (for instance, to permit the displayed representation of a field to depend on other information in the record); most `actual->display` functions will just take one argument—for intance, `upcase` is a valid `actual->display` function.

`match-actual->display`
`match-display->actual`

These functions are like `actual->display` and `display->actual`, but are only invoked when reading a displaying a search specification. If they are not specified (as will usually be the case), then the ordinary (`match-`-less) versions are used for search specifications too. This is used, for instance, for the string type, so that `dbm-string-prefix-regexp` can be used to specify a regular expression search rather than a substring search.

These slots should be set to symbols, not to functions proper; that is, to specify that function `foo` should be use, set the slot to `'foo`, not to `(function foo)`.

I don't know that these belong in the displayspec, but I don't quite know where they do belong.

`truncation-editing-action`

This function specifies what to do when a field being edited is too large for the specified displayspec size; this action may be different from that taken when simply displaying the offending value. It may also be set with the `trunc-edit` slot assigner.

`reachablep`

A Boolean value determining whether movement commands should skip this display field. The `reachable` and `unreachable` slotsetters are used to assign a value to this slot.

15.6 Display specification abbreviations

Complicated display specifications—those which specify more than a few optional parameters—can clutter the display format, keeping it from looking like the data display buffer will when a database record is being displayed. The user may tolerate the complicated display specification, define a new displaytype (which would permit the display specification to consist of just the field-name and displaytype), or use a field abbreviation which is defined elsewhere in the format file. The display specification abbreviation is a symbol which expands to a full display specification; when a field specification consisting of only the abbreviation is encountered, the expansion is substituted and processing continues. (Actually, a displayspec corresponding to the expansion is used, but since displayspecs are immutable, this doesn't make a difference.) Display specification abbreviations can be much more concise than ordinary display specifications, which contain at least a fieldname and often other information to boot.

[[[How to define display specification abbreviations.]]] [[[Poorly-named variable dbf-fieldabbrevs controls this; it isn't getting set anywhere, though.]]]

16 Customization

16.1 Auxiliary and format files

A database designer can customize a database by providing code to be executed when the database is read in (see Section 14.5 [Reading from disk], page 63). The optional auxiliary file usually contains the code specific to a particular database, but the format file, which specifies the on-screen arrangement of fields of a record, can also contain such code.

Since the auxiliary file is read after the format file has been found but before it has been parsed, neither file can specify the other. The format file can, however, load arbitrary files, which is nearly as good as being able to specify an auxiliary file.

The auxiliary file can be specified in the `aux-file` database slot; if it isn't, EDB looks for a file with the same name as the database file, but ending with one of the suffixes in `db-aux-file-suffixes`.

`db-aux-file-suffixes`
: List of auxiliary file suffixes; the basename is that of the database file. The suffixes are tried in order; the default is (".dba" ".aux" "a"). The . that may precede the extension must be specified explicitly.

`db-aux-file-path`
: List of directories (strings) to search, in order, for auxiliary files not found in the directory with their associated databases.

The auxiliary file is evaluated in the data display buffer and so can set variables local to that buffer, such as hooks (see Section 16.2 [Hooks and customization functions], page 76). The database itself can be manipulated via the dynamic variable `database` or the buffer-local variable `dbc-database`. For instance, auxiliary files often set the `print-name` slot of their associated databases.

Code in an auxiliary file should be specific to the particular database; more general code is best placed in a separate file which is `load`ed (or, better, `require`d) by the auxiliary file. For instance, if you want to permit EDB to manipulate files of type Foo, you should put all Emacs Lisp code that applies to every Foo file in one file ('`db-foo.el`', say), and then put `(require 'db-foo)` in the auxiliary file associated with a particular Foo file. (Alternately, you may autoload a function that will be called in the auxiliary file; function `db-tagged-setup` is autoloaded from '`db-tagged.el`' in this manner.) Either technique keeps auxiliary files simple and small and makes Foo-specific code easier to debug, byte-compile, and load only once per session. These advantages easily outweigh the introduction of an extra file.

Since the format file has not yet been interpreted, the auxiliary file could even change the contents of the buffer (and so the apparent contents of the format file); such extreme trickiness is only called for in special circumstances.

The format file can contain Emacs Lisp code in its "Local Variables" section; that code can do anything that the code in the auxiliary file can do. If the format file is not named explicitly in the database (in the `default-format-file` slot), then function `db-file->format-file` tries to find one based on the database file name and the suffixes in `db-format-file-suffixes`; if that doesn't work either, the user is prompted for a display format to use.

`db-format-file-suffixes`
: List of format file suffixes; the basename is that of the database file. The suffixes are tried in order; the default is (".dbf" ".fmt" "f"). The . that may precede the extension must be specified explicitly.

`db-format-file-path`
: List of directories (strings) to search, in order, for format files not found in the directory with their associated databases.

Code in the format file is useful for customizations specific to a particular format (such as setting variables which are local to the data display buffer); they can also be used for database-specific customizations if the database designer is sure that the file will always be the primary (first-selected) format for the database.

16.2 Hooks and customization functions

Hooks are variables whose values are "hook functions" (or lists of hook functions) which are called at particular times, such as when EDB has finished loading (to permit the user to load customization code) or when a value has just been changed. Since hook functions can contain arbitrary code, they permit very powerful customizations. Customization functions, like hooks, can call arbitrary code, but are single functions, never lists. Sometimes "hook" is used to mean either a hook or a customization function.

The following sections describe EDB's hooks and customization functions. Many of these hooks are change hooks, which permit a function (or functions) to be run whenever a value changes.

These change hooks may be divided into two basic types: format change hooks and recordfieldspec change hooks. The former are associated with a particular display format and are invoked when the value in a particular field, or in any field, changes. The latter (which are not yet implemented) are associated with a recordfieldspec and are invoked whenever a database record slot of a particular type is changed.

16.2.1 Load and read hooks

After EDB has finished loading, the following hook is run. This permits user customizations to be loaded automatically when EDB is (rather than being loaded unconditionally in the '`.emacs`' file, for instance), and permits users to change the definitions of functions defined by EDB, if desired.

`db-load-hooks`

Function or list of functions run after loading EDB. You can use this to customize key bindings or load extensions.

The following two hooks are useful for causing database values seen by EDB to be different than those in the database file. The first can be used to modify the database file before it is read in; the second can be used to modify the database after it has been read in but before the user has had a chance to see it.

`db-before-read-hooks`

Function or list of functions run immediately before a database is first read but after all local variables are set. The hooks are run in the data display buffer with variable database bound. Variable db-buffer is bound to a buffer containing the database file.

This is a global variable. If you set it to be specific to a particular database (for instance, in the format or auxiliary file), then consider having its last action be to reset the variable to `nil`.

`db-after-read-hooks`

Function or list of functions run after a database is completely read. The hooks are run in the data display buffer with variable database bound. For databases with nonregular formats, you might put a call to `database-stored->actual` here, for instance.

This is a global variable. If you set it to be specific to a particular database (for instance, in the format or auxiliary file), then consider having its last action be to reset the variable to `nil`.

16.2.2 Database minor mode hooks

EDB provides hooks that are run whenever the data display buffer is switched between view mode and edit mode and which are run when a summary buffer is created.

`db-view-mode-hooks`
: Function or list of functions called when database view mode is entered.

`db-edit-mode-hooks`
: Function or list of functions called when database edit mode is entered.

`database-summary-mode-hooks`
: Function or list of functions run when switching to database summary mode.

16.2.3 Record display hooks

The following function is run by the `display-record` function, which places a record's values in a data display buffer, each time a record is about to be displayed.

`dbf-before-display-record-function`
: A function called before a record is displayed by `display-record`. The function should take one argument, the record.

  This is a good place to put calls to `db-alternate-format`. Depending on your function's implementation, however, you may silently override any user calls to that function.

Here is an example of how you might use this:

```
(defun set-format-from-data (record)
  (if (< 0 (record-field record 'net-profit dbc-database))
      (db-alternate-format "loss format" "~/acct/db/loss.fmt")
    (db-alternate-format "profit format" "~/acct/db/profit.fmt")))

(setq dbf-before-display-record-function 'set-format-from-data)
```

This uses two different display formats, depending on the value of one field of a record. As the user moves from record to record in the database, each one is shown using the appropriate display format. A preferable implementation omits the filenames from the calls to `db-alternate-format` and instead uses, in the format or auxiliary file,

```
(setq dbf-alternate-format-names
  '(("loss format" . "~/acct/db/loss.fmt")
    ("profit format" . "~/acct/db/profit.fmt")))
```

See the example file 'arb-demo.dba' for an example of this. `dbf-alternate-format-names` need not specify the full pathnames if the format files are located in the same directory as the database or if '`"~/acct/db"`' is placed in `db-format-file-path`.

It would also be profitable to set `dbf-format-name` to whichever the first format was (this could be done in the format file's "Local Variables" section so that the first-selected buffer wouldn't get read in twice (once when the database was read in and once when `display-record` was first called).

Finally, the database designer would probably arrange for there to be a change hook on the net-profit field so that when its value changed, the record could be redisplayed in the appropriate format automatically.

16.2.4 Edit mode hooks

This function is called whenever the user enters a field to edit it, which provides an easy way to customize the behavior of particular format fields.

`dbf-enter-field-function`

: A function called whenever a display field is entered. The function takes the displayspec index as an argument, which is guaranteed to be `dbf-this-field-index`.

It is sometimes advantageous to have a particular action happen only once per edit of a record. For instance, when a record's address, city, state, or zip-code fields are edited, we might like to copy all the values to the old-address, old-city, old-state, and old-zip-code fields. We only want this to happen once, however: if the user edits first the address, then the city, we don't want to repeat the process, because then the old-address field would get written over by the new value of the address field.

One way to prevent this from happening more than once is to set a variable when the copying is done, and then don't do the copying if that variable is set. The variable would be reset whenever a new record was edited. The following variable, which contains a list of other variables to reset each time database edit mode is entered, can accomplish just what is desired, when combined with a judicious use of change hooks.

`dbf-reset-on-edit-list`

An alist of (variable-name . default-value) pairs. Every time Edit Mode is entered, these buffer-local variables are reset to their default values. This is good for making sure that something only happens once each time a record is edited.

16.2.5 Display format change hooks

The following hook is run whenever a new record is created.

`db-new-record-function`

Function called on empty records before they're inserted in the database. Takes two arguments, the record and the database.

A typical use is to set default information or add a timestamp. For instance:

```
(defun set-update-date (record database)
  "Provide defaults for new records in the database."
  (record-set-field new-rec 'updatedate
                    (parse-date-string (current-date)) database))
(setq db-new-record-function 'set-update-date)
```

The display format change hooks are called when a user changes a record field value. There are separate change hooks that run the first time any field is modified, whenever any field is modified, and whenever a particular field is modified. They run in the order `dbf-first-change-function`, `dbf-every-change-function`, and finally one of the elements of `dbf-change-functions`. Each change hook is either `nil` or a function of three variables: the fieldname of the just-modified field (a symbol) and the pre- and post-modification field values. The function should return a boolean which determines whether the entire record should be redisplayed; it is useful to return `t` if the change hook modifies fields other than that named by its first argument, and `nil` otherwise. (This result is `or`ed into `dbf-redisplay-entire-record-p`, which controls whether the record should be completely redisplayed after a field modification is done, and which may be set directly by the adventurous.)

`dbf-first-change-function`

A function called the first time a record field is modified, or `nil`. The function takes the fieldname and the old and new values as arguments, and returns `t` if the record should be redisplayed.

Here is an example of code to update the last modification field of a record, assuming its type is date:

```
(defun update-last-modified-date (fieldname oldvalue newvalue)
```

```
    "Put the current date in this record's `last modified' field."
    (dbf-this-record-set-field 'last-modified
                               (parse-date-string (current-date))))

  (setq dbf-first-change-function 'update-last-modified-date)
```

`dbf-every-change-function`
: A function called whenever a record field is modified, or `nil`. The function takes the fieldname and the old and new values as arguments, and returns `t` if the record should be redisplayed.

`dbf-change-functions`
: A vector of one function (or `nil`) per record field (not display field). The functions take the fieldname and the old and new values as arguments, and return `t` if the record should be redisplayed. Use `dbf-set-change-function` to set the fields of this vector.

`dbf-set-change-function`
: Set the change function for *fieldname* to *function* in the current database. *function* should take the fieldname and the old and new values as arguments, and return `t` if the record should be redisplayed.

It is easy to make a field's value dependent on that of another field. For instance, suppose a salesman's commission should be 10% of the selling price of an item, and both fields are of type number. The database designer might choose to make the commission field unreachable (see Section 15.5 [Display specification optional parameters], page 71) and compute it whenever the selling price field varies. The latter operation could be done as follows:

```
(defun set-commission-from-selling-price (fieldname oldvalue newvalue)
  ;; If we used (dbf-displayed-record-field 'selling-price) for newvalue, this
  ;; would work even if not called as a change function for selling-price.
  (dbf-displayed-record-set-field 'commission (/ newvalue 10)))

(dbf-set-change-function 'selling-price 'set-commission-from-selling-price)
```

The user may modify records explicitly by calling `dbf-displayed-record-set-field` (see Section 17.3.1 [Manipulating records], page 93); when that is done, the following hook is invoked. It is different from the above functions in that they are called when the user edits a field, while it is called when Emacs Lisp code modifies a field (usually as a result of some user action).

`dbf-set-this-record-modified-function`
: A function called every time the working copy `dbf-this-record` is created by `dbf-set-this-record-modified-p`. The function takes no arguments and its return value is ignored. It is called after `dbf-this-record-original` is copied to `dbf-this-record` and after `dbf-this-record-modified-p` is set to `t`.

Another function is invoked when changes to a record are committed—that is, when changes to the record which is being displayed are copied back into its original in the database.

`dbf-after-record-change-function`

Function called whenever changes to a record are recorded semi-permanently by `dbf-process-current-record-maybe`. For convenience, the function takes the record as an argument, which is guaranteed to be `dbf-this-record`. Its return value is ignored.

16.2.6 Recordfieldspec change hooks

Recordfieldspec change hooks are not yet implemented.

16.3 Local variables

Variables may be specified to be local to a particular data display buffer or to a database; that is, when the variable's value is changed in one data display buffer or in one database, its value elsewhere is unaffected.

16.3.1 Per-data-display-buffer variables

Per-data-display-buffer variables permit different data display buffers to have different values for variables. This feature is heavily used by the EDB implementation; for instance, the per-buffer variable `dbc-database` records which database the data display buffer is displaying. Per-data-display-buffer variables are also useful when several data display buffers are all displaying the same database. The built-in Emacs function `make-variable-buffer-local` makes an ordinary variable local to every buffer.

`db-default-field-type`

The type to use for record fields whose type is not explicitly specified.

16.3.2 Per-database variables

Per-database variables permit every data display buffer viewing a particular database to share information without making it global or interfering with other databases and other data display buffers. When the database is saved in internal EDB file layout, per-database variables are also saved, so their values persist from one invocation of the database to the next. Use the following functions to create and manipulate per-database variables.

`database-make-local`
: Declare a database-local variable named by *symbol* for *database*. Each such variable should only be declared once. If optional argument *value* is specified, the variable is set to it.

  Database designers who are very concerned about speed should arrange to call this function in increasing order of frequency of variable reference; that is, add the least-used variables first.

`database-local-p`
: Return non-`nil` if *symbol* is a database-local variable for *database*.

`database-set-local`
: Set the value of database-local variable *symbol*, in *database*, to *value*. *symbol* must have been declared by a previous call to `database-make-local` unless optional argument *no-error* is supplied, in which case the function does that automatically.

`database-get-local`
: Return the value of database-local variable *symbol* for *database*. *symbol* must have been declared by a previous call to `database-make-local` unless optional argument *no-error* is supplied, in which case `nil` is returned.

To learn how to set local variables automatically whenever a record is edited, See Section 16.2.4 [Edit mode hooks], page 79.

(This section of the manual does not refer to the "Local Variables" section of the database format file, which may be used to set variables and execute arbitrary Emacs Lisp code when a data display buffer is being set up; that is described in Section 14.5 [Reading from disk], page 63.)

16.4 Global variables

This section describes a potpourri of customization variables which you can use to control EDB's behavior.

When a potentially slow computation is underway, EDB displays a message in the echo area reporting how many records have been processed. Use the following variable to control how often this message is updated.

`db-inform-interval`

When doing a lengthy computation, inform the user of progress every this many records. If `nil`, don't inform.

To make EDB use `with-electric-help` where appropriate instead of `with-output-to-temp-buffer`, set the following variables, which default to `nil`. You must have '`ehelp.el`' on your load path or have already loaded it.

`use-electric-help-p`

Non-`nil` if Emacs programs should use electric help where possible. Don't set this to a non-`nil` value unless the ehelp package is available.

EDB does not simply test for `(featurep 'ehelp)` because some packages load '`ehelp`' without determining whether the user desires that behavior. Even if that has happened, users of EDB have a way to retain Emacs's traditional behavior.

`with-electric-help-maybe`

Similar to 'with-electric-help' if `use-electric-help-p` is non-`nil`; otherwise like `with-output-to-temp-buffer` with the "*Help*" buffer. Ehelp is loaded if necessary. *body* is not a thunk (a function of no arguments) but simply a set of forms.

17 Database representation

Perhaps the most important information about a database—besides the records it contains—is the number of fields in each record, and the type of each field. As explained in Section 1.4 [Terminology], page 4, a database consists of records with identical numbers of fields; each field has an associated type such as string or integer. Each field also has a name which is used when extracting its value from the record.

A database is basically just a doubly-linked circular list, where each link contains a single record. The database contains some additional supporting information, and so does each link in the list of records. The database is represented as a structure of type database whose `first-link` slot points to the circular list of links. Internally, database records are represented by vectors; however, the programmer should never manipulate those vectors directly, only through the functions described in this chapter.

Given a record, it is not possible to determine which link (if any) points to it; similarly, you cannot go from a link to its containing database. The database-to-link and link-to-record connections are one-way.

17.1 The database structure

The internal representation of a database is as a structure of type database. The slots of this structure may be accessed by using the macro `database-`*slotname*. The slots may be set using the macro `database-set-`*slotname*, whose second argument is the value to be stored in the slot.

The slots of the database are as follows:

`print-name`

: A string which briefly describes the database. It appears, among other places, in prompts for questions regarding the database. It defaults to "Unnamed databse *n*", where the positive integers are assigned to *n* in order.

`first-link`

: The first link in the database. The links are arranged as a doubly-linked circular list, and each link contains a record, among other information. See Section 17.1.1 [The link structure], page 90.

`no-of-records`
: An integer, the number of records (and links) in the database. The first link is numbered 1 and the last link is numbered `no-of-records`.

`file`
: A string, the name of the file from which this database was read.

`file-local-variables`
: A string, the text of the "Local Variables" section of the file from which this database was read, if any.

`aux-file`
: A string, the name of this database's auxiliary file. If it is `nil`, then a number of default filenames are tried, based on `db-aux-file-suffixes` (see Section 16.1 [Auxiliary files], page 75).

`data-display-buffers`
: A list of data display buffers which are displaying this database. Since every summary buffer is associated with (and subordinate to) a data display buffer, summary buffers are not listed in the database structure.

`default-format-file`
: A string, the name of the default format file for this database. If it is `nil`, then a number of default filenames are tried by function `db-file->format-file` (see Section 16.1 [Auxiliary files], page 75).

`omit-functions`
: This does not appear to be used at present.

`no-of-fields`
: An integer, the number of fields in each record.

`fieldnames`
: A vector of symbols, the names of the record fields. Function `fieldnumber->fieldname` (see Section 17.3.2 [Accessing record fields], page 94) uses this to determine the name of a field, given its index.

  The user may set this slot to be a list, and EDB will automatically convert it to a vector, as well as setting other database slots that can be determined from it. (When doing so, use `database-set-fieldnames-to-list`, which can safely be placed in any format file, instead of `database-set-fieldnames`; for details, see Section 11.1 [Changing display formats], page 38.) This information is duplicated in the recordfieldspecs.

  `database-set-fieldnames-to-list`
  : Set *database*'s fieldnames and record field types according to *fieldnames-list*. Users should never call `database-set-fieldnames` directly. *fieldnames-list* is a list of fieldnames (symbols); each list element may instead be a cons of fieldname and type to specify the field's recordfieldtype as well. If no type is specified for a field, the value of `db-default-field-type` is used.

This function sets several database slots besides the fieldnames slot, but has no effect if the fieldnames slot of the database is already set.

The following call specifies three fields of types one-line-string, integer, and string, presuming that variable `db-default-field-type` (see Section 16.3.1 [Per-data-display-buffer variables], page 82) has not been changed from its default of `string`:

```
(database-set-fieldnames-to-list database '((name . one-line-string)
                                            (age . integer)
                                            address))
```

`fieldname-alist`

An alist of fieldnames and indices. Function `fieldname->fieldnumber` (see Section 17.3.2 [Accessing record fields], page 94) uses this to determine the index of a field, given its name.

`recordfieldspecs`

A vector of symbols or recordfieldspecs which specify the type of each record field. If the value is a symbol, it is a record field type name which is converted to a recordfieldspec via function `recordfieldtype->recordfieldspec`, which performs a lookup in `db-recordfieldtypes`. To access or change a particular recordfieldspec, use the following functions:

`database-recordfieldspec`

Return the recordfieldspec of *database* corresponding to *record-index*. Dereferences via `recordfieldtype->recordfieldspec` any symbol found in the recordfieldspecs slot of *database*.

`database-recordfieldspec-type`

Return the type of the recordfieldspec of *database* corresponding to *record-index*.

`database-set-recordfieldspec`

Set the recordfieldspec of *database* corresponding to *record-index* to *rs*. Use this to redefine, on a per-field basis, subfields of the recordfieldspec.

`field-priorities`

Determines in which order fields are compared when sorting records, and which fields are ignored entirely.

This slot's vlaue is a cons of two lists: the first list contains fields that will be used for sorting, and the second list is the ignored fields. Each list consists of pairs of fieldnumber and order-info. The user may use `nil` for the second list when setting this slot. EDB always maintains the list of ignored fields, however, as its order might be worthwhile—for instance, for reminding the user of what the order used to be.

The order-info specifies how the field should be sorted: in increasing order, in decreasing order, or according to an arbitrary function. To choose the default ordering, or its inverse, use the symbol `increasing` or `decreasing`. Otherwise, order-info is a cons of *type* and *value*, where *type* is a symbol (either `order-function` or `sort-function`) and *value* specifies the function.

In the sorting section's example (see Chapter 7 [Sorting], page 28), the record fields were foo, bar, baz, bum, and bee, and records were to first be sorted on baz in increasing order, then on foo in decreasing order, and finally on bum in increasing order, ignoring bar and bee entirely for the purposes of the sort. The corresponding field priorities list would be

```
(((2 . increasing) (0 . decreasing) (3 . increasing)) .
 ((1 . increasing) (4 . increasing)))
```

`omitted-to-end-p`

A boolean which determines whether, when sorting, omitted records should be sorted in the usual way or placed at the end of the sorted order.

`internal-file-layout-p`

A boolean which determines whether the database will be saved in internal file layout. This has no effect when the database is read, but it is set at read time so that, by default, the database will be written out as it was read in.

Setting this slot, then saving the database to disk, is a good way to convert the database to or from internal file layout. It can be set in the usual way, or interactively via use of the following function:

`M-x db-toggle-internal-file-layout`

Toggle whether the database will be saved in EDB's internal file layout. With a nonzero prefix argument, set it to use internal file layout. With a zero prefix argument, set it not to use internal file layout.

`record-sepinfo`
`field-sepinfo`
`alternative-sepinfo`

These sepinfos are used when reading databases with regular file layouts. A sepinfo contains a particular string, a regular expression, or a function that specifies how pieces of information are separated in the disk file (for more about the sepinfo struture, see Section 14.2.1 [How to specify regular file layouts], page 53). These sepinfos describe how to separate records, fields within a record, and alternatives within a field. (The latter is not yet fully implemented.)

`read-record-from-region`

Nil or a function of no arguments which returns a record read from the current region of the current buffer. For details, See Section 14.4 [Nonregular file layout], page 60.

`write-region-from-record`

Nil or a function which takes a record as its argument and inserts the file representation of that record in the current buffer. For details, See Section 14.4 [Nonregular file layout], page 60.

`sub-fieldsep-string`
`sub-recordsep-string`

When delimiter substitution is required in reading a database, these strings are temporarily used to delimit fields and records, respectively. (These strings replace the actual field and/or record separators before substitution occurs.) Their values are chosen automatically if these slots aren't set.

`quotation-char`

A character which is used to quote delimiters which appear in a database field. This character is prepended to ambiguous strings; strings preceded by it are treated verbatim rather than as delimiters. For more about quotation in reading databases, see Section 14.2 [Regular file layout], page 52.

`quotation-char-regexp`

A regexp used to recognize the quotation character.

`quoted-regexp`

A regexp which matches all strings that should be quoted. If it is `nil`, it is set from the `quoted-strings` slot.

`quoted-strings`

A list of strings that should be quoted. If it is `nil`, then the value returned by function `quoted-strings-default`—basically all the strings mentioned in the database's `record-sepinfo`, `field-sepinfo`, and `alternative-sepinfo` slots, plus the `quotation-char` slot—is used instead.

`actual-quoted-regexp`

The regexp that is actually used for finding quoted strings. The user should never set this slot.

`substitutions`

An alist of actual and stored strings which permits translations from how the data appears in the data file to how it should really look; for instance, in data files with the tab-separated text layout, fields may not contain newlines, so any newlines in the data can be converted to some other character (such as '`^K`') when the database is written and then converted back when it is read in again. For more about substituion in reading databases, see Section 14.2 [Regular file layout], page 52.

`modified-p`
: Non-`nil` if this database has been modified since it was last read or written.

`modifiable-p`
: Non-`nil` if this database may be modified. It is set to `nil` if the database file is not writable, and occasionally for other reasons. This does not prevent the user from editing edit mode, only from making changes while in edit mode.

  The slot may be set directly, the following function is bound in view, edit, and summary modes to permit the slot to be changed interactively.

  `C-x C-q` (`db-toggle-modifiable-p`) Toggle whether the database may be modified by the user. With a nonzero prefix argument, set it modifiable. With a zero prefix argument, set it non-modifiable.

`locals`
: An alist of symbols and values for per-database variables. (For the number of local variables I expect each database to have, an alist is faster than a hashtable, and it's easier to save to disk besides.) Such variables should be created with `database-make-local`, set using `database-set-local` (note the singular form) and dereferenced with `database-get-local`; for more information about these functions, see Section 16.3 [Local variables], page 82.

17.1.1 The link structure

The records of the database—the information that the user cares most about—are kept in a doubly-linked list, one record per link. The link structure also contains some other information about the record which doesn't belong in the record proper. The slots of a link are listed below; a slot may be accessed by using the macro `link-`*slotname* and set using the macro `link-set-`*slotname*, whose second argument is the value to be stored in the slot.

`prev`
: The previous link in the circular list.

`next`
: The next link in the circular list.

`omittedp`
`markedp`
: These booleans are non-`nil` if this record is marked or omitted, respectively. For more information about marking and omitting, see Chapter 9 [Marking and omitting], page 34.

`summary`
: A string which is used to represent this record in the summary buffer, or `nil` if the record's value has changed since the last summary buffer was made (or if no summary buffer has been made).

`record`
: The database record proper, a vector with as many elements as the record has fields. Setting this slot with the `link-set-record` function also has the effect of setting the

`summary` slot to `nil`, which is usually what is desired; to set only the `record` slot, use the `link-set-record-slot` macro instead.

17.2 Mapping over the database

Mapping refers to applying a function to each link or record in the database, or executing a piece of code for each link or record. Four functions provided this capability. The first two, more complicated, ones, provide access to each link of the database in turn.

`maplinks`
: Apply FUNC to every link in *database*. If optional third arg *omit* is non-`nil`, apply FUNC only to unomitted links. If optional fourth arg *message* is non-`nil`, it should be a format string containing one numeric (%d) specifier. That message will be issued every `db-inform-interval` links. If optional fifth arg *accumulate* is non-`nil`, return a list of the results; otherwise return `nil`.

  In the body, variable maplinks-index is bound to the index of the link being operated upon, and maplinks-link is the argument to FUNC. The loop may be short-circuited (aborted) by calling `maplinks-break`. To avoid the per-link function call overhead, use `maplinks-macro` instead.

`maplinks-macro`
: Execute BODY for each link in *database*, and return `nil`. If optional third arg *omit* is non-`nil`, execute BODY only for unomitted links. If optional fourth arg *message* is non-`nil`, it should be a format string containing one numeric (%d) specifier. That message will be issued every `db-inform-interval` links.

  In the body, variable maplinks-link is bound to the link being operated upon, and maplinks-index is bound to its index. The loop may be short-circuited (aborted) by calling `maplinks-break`. Speed demons should call this instead of `maplinks` to avoid a function call overhead per link.

`maplinks-break`
: Cause the maplinks loop to quit after executing the current iteration. This is not a nonlocal exit! It sets a flag which prevents future iterations. Actually, it sets maplinks-link.

Two other functions provide a slightly different interface which simplifies access to each record. Links and the information contained in them are not accessible from database records, but when that information is not of interest, these functions provide direct access to records.

`maprecords`

> Apply FUNC to every record in *database*. Return `nil`. If optional third arg *omit* is non-`nil`, apply FUNC only to unomitted records. If optional fourth arg *message* is non-`nil`, it should be a format string containing one numeric (%d) specifier. That message will be issued every `db-inform-interval` records. If optional fifth arg *accumulate* is non-`nil`, return a list of the results; otherwise return `nil`.
>
> This is syntactic sugar for a call to `maplinks`, which see. See also `maprecords-macro`.

`maprecords-macro`

> Execute BODY for each record in *database*, and return `nil`. If optional third arg *omit* is non-`nil`, execute BODY only for unomitted records. If optional fourth arg *message* is non-`nil`, it should be a format string containing one numeric (%d) specifier. That message will be issued every `db-inform-interval` links.
>
> In the body, variable maprecords-record is bound to the record being operated upon. The loop may be short-circuited (aborted) by calling `maprecords-break`.
>
> This is syntactic sugar for a call to `maplinks-macro`, which see. See also `maprecords`.

`maprecords-break`

> Cause the maplinks loop to quit after executing the current iteration. This is not a nonlocal exit! It sets a flag which prevents future iterations. Actually, it sets maplinks-link.

For instance, to sum, for all records, the values contained in field summand (of type number), you could use any of the following forms, presuming that variable `database` was set to the database in question:

```
(let ((result 0))
  (maplinks-macro
    (setq result (+ result (record-field (link-record maplinks-link)
                                         'summand database)))
    database)
  result)

(let ((result 0))
  (maprecords-macro
    (setq result (+ result (record-field maprecords-record 'summand database)))
    database)
  result)

(let ((result 0))
  (maprecords
    (function (lambda (record)
      (setq result (+ result (record-field record 'summand database)))))
    database)
```

```
  result)

(apply (function +)
       (maprecords
 (function (lambda (record) (record-field record 'summand database)))
 database nil nil t))
```

17.3 Manipulating records

A database consists of records, each of which has the same makeup: corresponding fields in a database's records contain data of the same type. For instance, the fifth field of each record might contain an address, and the seventh field, a date. The particular addresses and dates would would vary from record to record. (Different databases will contain records with different numbers and types of fields.) Each field has a name and a type, which specifies what sort of information can be stored in the field; for more details about record field types, see Chapter 13 [Record field types], page 45.

Records are represented internally as vectors, but should never be operated on as such; use the abstractions described in this section.

17.3.1 Creating and copying records

`make-record`

Return a record with number of fields specified by argument *database*.

When the user creates a new record by using `db-add-record` (see Section 4.5 [Adding and removing records], page 20), `db-new-record-function` is invoked (see Section 16.2.5 [Display format change hooks], page 80), the number of records in the database is modified, and so forth. `make-record`, on the other hand, performs none of these housekeeping tasks.

`copy-record`

Return a copy of *record*.

`copy-record-to-record`

Copy the field values of the *source* record to the *target* record.

17.3.2 Accessing record fields

Ordinarily, record fields are accessed by specifying the name of the desired field; the database must also be specified so that the fieldname-to-fieldnumber correspondence can be determined.

`record-field`
: Return from *record* the field with name *fieldname*. Third argument is *database*.

`record-set-field`
: Set, in *record*, field *fieldname* to *value*. Fourth argument *database*. Check constraints first unless optional fifth argument *nocheck* is non-`nil`. This version correctly deals with reversed *value* and *database* arguments.

There are also special commands for manipulating the current record—that is, the one that appears in the data display buffer. They are better because they require fewer arguments, flag that a redisplay of the record is necessary, and automatically call `dbf-set-this-record-modified-p`, which is essential if the changes are to be copied back into the original record in the database from the one that is being displayed. (A copy is always displayed so that changes can be gracefully undone.)

`dbf-displayed-record`
: Return the record currently displayed in this data display buffer. This is `dbf-this-record` if `dbf-this-record-modified-p` is non-`nil` and `dbf-this-record-original` otherwise.

`dbf-displayed-record-field`
: Return the value of the field named *fieldname* from the displayed record.

`dbf-displayed-record-set-field`
: Set field with name *fieldname* in displayed record to *value*. Cause the entire record to be redisplayed pretty soon.

`dbf-displayed-record-set-field-and-redisplay`
: Set field with name *fieldname* in displayed record to *value*. Cause the entire record to be redisplayed immediately.

`dbf-set-this-record-modified-p`
: Set the value of `dbf-this-record-modified-p` to *arg*. If *arg* is non-`nil` and `dbf-this-record-modified-p` is `nil`, also do the necessary record-copying and call `dbf-set-this-record-modified-function`.

It is also possible—and more efficient—to use the fieldnumbers directly. The database does this internally, remembering fields by their numbers and only converting to fieldnames when interacting with the user. Adopting such a strategy for all field accesses would be cumbersome, error-prone, and make reading code difficult, but in some situations—particularly when `record-field` or `record-set-field` is being called with a constant second argument—it is worthwhile. The code can be sped up by allocating a variable for the fieldnumber, looking it up after the database has been loaded (for instance, by calling `fieldname->fieldnumber` after `database-set-fieldnames-to-list` or in `db-after-read-hooks`), and then using that variable along with `record-field-from-index` or `record-set-field-from-index`.

Do not confuse the record fieldnumber, which describes in what order fields happen to occur in the database's internal representation of a record, with the format fieldnumber, which describes in what order fields are displayed in the data display buffer.

`fieldname->fieldnumber`

Given a *fieldname* and *database*, return a record fieldnumber. Do not be fooled into thinking this is a format fieldnumber.

`fieldnumber->fieldname`

Given a record *fieldnumber* and *database*, return a record fieldname. The first argument is not a format fieldnumber.

`record-field-from-index`

Return from *record* the value of the *fieldno*th field.

`record-set-field-from-index`

Set, in *record*, the *fieldno*th field to *value*. Checks field constraints first if *database* is non-`nil`.

17.3.3 Mapping over record fields

To perform an action on every field of a record, use the following function or macro.

`mapfields`

Apply *func* to each field in *record*, with variable mapfields-index bound. Third argument is *database*.

`mapfields-macro`

Execute *body* for each field of *record*, a record of *database*, with variables mapfields-field and mapfields-index bound.

18 Naming conventions

18.1 Function and variable naming conventions

The names of EDB's functions and variables contain one of the following prefixes:

`edb-` These variables contain information about EDB such as the version number, last modification date, or names of the files comprising EDB. They do not relate to general database functionality, only to this particular implementation.

`db-` In a variable, indicates that the variable is global and affects all databases. In a function, indicates that the function is user-visible and may be called interactively. It is also used in some situations for internal database functionality which is not connected with any particular buffer.

`database-`
These functions operate on the (internal representation of) the database structure itself.

`dbc-` Indicates a variable local to the data display buffer which refers to the current database (the database being manipulated by that data display buffer), or a non-user-visible function which manipulates such variables. The 'c' stands for "current."

`dbf-` Indicates a variable local to the data display buffer which controls some aspect of formatting, or a non-user-visible function which manipulates such variables. The 'f' stands for "format"; many such variables are intimately related to the format, and the data display buffer used to be called the format buffer.

`dbs-` Indicates a variable local to the summary buffer, or a summary buffer function. Since the summary buffer may disappear at any time, the summary buffer gets most of its information from the associated data display buffer's local variables.

`dbfs-` Indicates a variable which is too important to be kept only in the summary buffer, which may disappear at any time, but is so often used by the summary buffer that it would be inefficient to keep it only in the data display buffer. Such variables are kept in both the data display and summary buffers.

`dbsi-` Indicates a variable local to a sort interface buffer, or a sort interface function.

18.2 File naming conventions

The names of EDB's files contain one of the following suffixes:

`.dat` These are database files proper; they contain the information that makes up the fields and records of the database. Database filenames may also contain no extension at all.

`.fmt` Format files control the structure of the data display buffer, which displays one record at a time.

`.dba` Auxiliary files contain arbitrary Emacs Lisp code; they can be used to define functions, set variables, or operate directly on the database.

For more information, see Section 1.2 [Invoking EDB], page 2.

Function Index

A

B

C

D

Q

R

S

W

X

Variable Index

Concept Index

E

F

S

T

U

V

W

Y