BIBTOOL

A Tool to Manipulate BibTeX Files

C Programmer's Manual

Gerd Neugebauer

Abstract

BibTool provides a library of useful C functions to manipulate BibTeX files. This library has been used to implement the BibTool program. This document describes This library and allows you to write C programs dealing with BibTeX files.

— This documentation is still in a rudimentary form and needs additional efforts. —

This file is part of BibTool Version 2.68

Gerd Neugebauer
Im Lerchelsböhl 5
64521 Groß-Gerau (Germany)

WWW: `http://www.gerd-neugebauer.de/`

Net: `gene@gerd-neugebauer.de`

Contents

1 Introduction

The BibTool C library provides functions to deal with BibTeX files. These functions are described in this document. Thus it should be fairly easy to write new C program which handle BibTeX files. The reader is assumed to be familiar with BibTeX files. this documentation will not repeat an introduction into BibTeX.

This documentation can not only be used to write new C programs dealing with BibTeX files but also to understand BibTool—The Program which serves as one example for using the BibToolC library. In any case it is essential to understand some of the underlying concepts. Thus it is vital to read some sections very carefully.

The BibTool program uses the BibTool C library. Well, in fact it is the other way round. Historically the BibTool program was first and then the library has been extracted from it. Nevertheless the BibTool program can serve as an example how the BibTool C libary can be used.

1.1 The Module `main.c`

This is the BibTool main module. It contains the `main()` function which evaluates the command line arguments and proceeds accordingly. This means that usually resource files and BibTeX files are read and one or more BibTeX files are written.

This file makes use of the BibTool C library but is not part of it. For this purpose it has to provide certain functions which are expected by the library. These functions are:

```
save_input_file()
save_macro_file()
save_output_file()
```

The arguments and the expected behavior of these functions is described below.

If you are trying to understand the implementation of BibTool the file `resource.h` plays a central rôle. Consult the description of this file for further details.

If you are trying to write your own program to manipulate BibTeX files then this file can serve as a starting point. But you should keep in mind that this file has grown over several years and it contains the full complexity of the BibTool program logic. Thus you can reduce this file drastically if you start playing around with the BibTool C library.

`int main()` Function

`int argc;`	*Number of arguments*
`char *argv[];`	*Array of arguments*

This is the main function which is automatically called when the program is started. This function contains the overall program logic. It has to perform the appropriate initializations, evaluate command line arguments, and run the main loop.

Returns: 0 upon success. Usually a failure raises an exception which leads to an `exit()`. Thus this function does not need to signal an error to the calling environment.

2 The BibTool C Library

2.1 The Header File `bibtool/bibtool.h`

This header file contains includes for all other header files belonging to the BibTool C library. It is here for the convenience of the C programmer who does not have to include two dozen header files but can use this single file. Thus any C program which utilizes the BibTool C library can start as follows:

```
#include <bibtool/bibtool.h>
```

Note that this include file also contains includes to system specific header files. They are determined during configuration.

2.2 The Header File `bibtool/check.h`

This header file makes available the function defined in `check.c`.

2.3 The Module `check.c`

`void` **`add_unique_field()`** Function

`Symbol` **`key;`** *the key to check*

A unique constraint for a field.

Returns: nothing

`void` **`apply_checks()`** Function

`DB` **`db;`**

Returns: nothing

2.4 The Header File `bibtool/crossref.h`

This header file makes available the function defined in `crossref.c`. This file includes the header files `database.h` and `record.h`.

2.5 The Module `crossref.c`

This module contains functions to expand crossref entries.

`void` **`clear_map()`** Function

Reset the map to it's initial state where no elements are contained.

Returns: nothing

`void` **`crossref_map()`** Function

`String` **`spec;`**	*the argument*

Returns: nothing

`bool` **`expand_crossref()`** Function

`DB` **`db;`**	*Database containing the entries.*
`Record` **`rec;`**	*The record to expand*

Expand all items inherited via a crossref.

Returns: `false` iff an error has occured

`void` **`map_add()`** Function

`rec_type` **`s_rec;`**	*the index of the source record type*
`Symbol` **`s_fld;`**	*the source field name*
`rec_type` **`d_rec;`**	*the index of the destination record type*
`Symbol` **`d_fld;`**	*the destination field name*

Add or overwrite a filed name mapping.

Returns: nothing

`Symbol` **`map_get()`** Function

`rec_type` **`s_rec;`**	*the index of the source entry type*
`Symbol` **`s_fld;`**	*the name of the source field*
`rec_type` **`d_rec;`**	*the index of the destination entry type*

Getter for a map element.

Returns: the new field name or `NO_SYMBOL`

2.6 The Header File `bibtool/database.h`

This header file contains functions which deal with databases.

This header file provides also access to the functions and variables defined in `database.c`. Consult the documentation of this file for details.

This header file automatically includes `<stdio.h>` and `record.h` aswell.

DB Type

This is a pointer type referencing a BibTeX database. It contains all information which characterizes a database.

The members of this record should not be used explicitly. Instead the macros should be used which are provided to accss this data type.

`typedef struct {`

`Record` **`db_normal`**`;`	*List of normal records.*
`Record` **`db_string`**`;`	*List of local macros.*
`Record` **`db_preamble`**`;`	*List of additional TeX code.*
`Record` **`db_comment`**`;`	*List of trailing comments which are not attached to records.*
`Record` **`db_modify`**`;`	*List of modification rules.*
`Record` **`db_include`**`;`	*List of included files.*
`Record` **`db_alias`**`;`	*List of aliases.*

`}` **sDB**, ***DB**;

`DB NoDB` Macro

This is an invalid database. In fact it is `NULL` of the type `DB`.

`Record DBnormal()` Macro

`DB` *The database to consider.*

This is the functional representation of the normal component of a database. It can be used to extract this information. It can also be used as a lvalue.

`Record DBstring()` Macro

`DB` *The database to consider.*

This is the functional representation of the string component of a database. It can be used to extract this information. It can also be used as a lvalue.

`Record DBpreamble()` Macro

`DB` *The database to consider.*

This is the functional representation of the preamble component of a database. It can be used to extract this information. It can also be used as a lvalue.

`Record DBcomment()` Macro

`DB` *The database to consider.*

This is the functional representation of the comment component of a database. It can be used to extract this information. It can also be used as a lvalue.

`Record DBalias()` Macro

`DB` *The database to consider.*

This is the functional representation of the alias component of a database. It can be used to extract this information. It can also be used as a lvalue.

`Record DBmodify()` Macro

`DB` *The database to consider.*

This is the functional representation of the modify component of a database. It can be used to extract this information. It can also be used as a lvalue.

`Record DBinclude()` Macro

`DB`	*The database to consider.*

This is the functional representation of the include component of a database. It can be used to extract this information. It can also be used as a lvalue.

2.7 The Module `database.c`

This module contains functions which deal with databases. Databases are stored in an abstract datatype `DB` which is defined in `database.h`. Methods are provided to query and modify a database.

`int apply_modify()` Function

`DB db;`	*the database*
`Symbol key;`	*the key*
`Record rec;`	*the record*

Returns:

`int * db_count()` Function

`DB db;`	*Database to count.*
`int *lp;`	*pointer to an integer for the length.*

Count all entries in a database. This includes normal as well as special records. The result is stored in a static array which is reused by `db_count()`. A pointer to this array is returned. The indices correspond to the entry types defined with `add_entry_type()` or declared as symbolic constants in `entry.h`.

The end of the array is marked by an element containing a negative number. In addition the argument `lp` can point to an integer where the number of valid elements is stored. If `lp` is `NULL` this step is omitted.

Returns: Static array containing the statistics.

`Record db_find()` Function

`DB db;`	*Database to search in.*
`Symbol key;`	*the key to search for*

Search the database for a record with a given key. If `RecordOldKey` is set for the record then use this value. Otherwise use `*Heap`. `*Heap` contains the reference key of normal records.

Deleted records are ignored. An arbitrary matching record is returned. Thus if more than one record have the same key then the behavior is nondeterministic.

Returns: the record found or `NULL` if none is found `void` **`db_forall()`** Function

`DB` **`db;`**	*Database containing* `rec`*.*
`bool (`**`*fct`**`)(DB,Record);`	*Boolean valued function determining the end of the processing. It takes two arguments a* `DB` *and a* `Record`*.*

Visit all normal records in the data base and apply the given function `fct` to each. If this function returns `true` then no more records need to be visited. No special order can be assumed in which the records are seen.

Returns: nothing

`void` **`db_insert()`** Function

`DB` **`db;`**	*Database to insert the record into.*
`Record` **`rec;`**	*Record to add to the database.*
`bool` **`verbose;`**	*Boolean to determine whether progress should be reported.*

Add a record to a database. The record can be any kind of record. It is added to the appropriate category.

Returns: nothing

`void` **`db_mac_sort()`** Function

`DB` **`db;`**	*Database to sort.*

Sort the macros of a database. The sorting uses increasing lexicographic order according to the character codes of the macro names. Note that this might lead to different results on machines with different character encodings, e.g. ASCII vs. EBCDIC.

Returns: nothing

`Symbol` **`db_new_key()`** Function

`DB` **`db;`**	*Database to search in.*
`Symbol` **`key;`**	*Key to find.*

Search the database for a record with a given old key and return the new one.

Returns: nothing

`void` **`db_rewind()`** Function

`DB` **`db;`**	*Database to rewind.*

Rewind the normal records of a database to point to the first record if at least one records exists. Otherwise nothing is done.

Returns: nothing

`Record` **`db_search()`** Function

`DB` **`db;`**	*Database to search in.*
`Symbol` **`key;`**	*the key to search for*

Search the database for a record with a given key. If `RecordOldKey` is set for the record then use this value. Otherwise use `*Heap`. `*Heap` contains the reference key of normal records.

Deleted records are not ignored! An arbitrary matching record is returned. Thus if more than one record have the same key then the behavior is nondeterministic.

Returns: the record found or `NULL` if none is found

`void` **`db_sort()`** Function

`DB` **`db;`**	*Database to sort.*
`int` **`(*less)(Record,Record);`**	*Comparison function to use. This boolean function takes two records and returns -1 if the first one is less than the second one.*

Sort the normal records of a database. As a side effect the records are kept in sorted order in the database. The sorting order can be determined by the argument `less` which is called to compare two records.

Returns: nothing

`Symbol` **`db_string()`** Function

`DB` **`db;`**	*Database*
`Symbol` **`sym;`**	*Name of the BibTeX macro to expand.*
`bool` **`localp;`**	*Boolean determining whether the search is only local to the db.*

Try to find the definition of a macro. First, the local values in the database `db` are considered. If this fails and `localp` is `false` then the global list is searched aswell. If all fails `NULL` is returned.

Returns: The macro expansion or `NULL` upon failure.

`void` **`db_xref_undelete()`** Function

`DB` **`db;`**	*Database to treat*

Scan through the database and undelete all entries which are in the transitive closure wrt the crossref relation. Initially all not deleted entries are in the set to consider.

Returns: nothing

`void` **`delete_record()`** Function

`DB` **`db;`**	*Database containing* `rec`.
`Record` **`rec;`**	*Record to delete.*

Delete a record from a database. It is not checked, that the record really is part of the database. The record is just unlinked from its list. Just in case the record should be the first one the database record is modified.

Returns: nothing

`void` **`free_db()`** Function

`DB` **`db;`**	*Database to release.*

Deallocate the memory occupied by a database. Note that any references to this database becomes invalid.

Returns: nothing

`DB` **`new_db()`** Function

Create a new database and initialize it to contain no information. If no memory is left then an error is raised and the program is terminated.

Returns: The new database.

`void` **`print_db()`** Function

`FILE *file;`	*The file handle for printing.*
`DB db;`	*The database to print*
`char *spec;`	*String containing the specification of the parts to print.*

Print a database to a file in a way which is readable by BIBTEX. The spec determines which parts should be printed and the order of this parts. The spec is processed from left to right. Each unknown character is silently ignored. The following characters correspond to parts of the database:

p The preamble.

$ All strings (macros) contained in the database.

S The strings (macros) which are used in the database.

s The strings (macros) contained in the database where the resource print.all.strings determines whether all strings should be printed or the used strings only.

n The normal records.

c The comments.

i The includes.

a The aliases.

m The modifies.

Upper-case letters which are not mentioned are silently folded to their lower-case counterparts.

Returns: nothing

`bool` **`read_db()`** Function

`DB db;`	*Database to augment.*
`String file;`	*File name to read from.*
`bool verbose;`	*Boolean to determine whether progress should be reported.*

Read records from a file and add them to a database. A function has to be given as one argument. This function is called for each record. If this function returns `true` then the record is added to the database. Otherwise the record is discarded.

The progress of reading is reported to `stderr` if the boolean argument `verbose` is `true`.

Returns: `true` if the file can not be opened. `false` otherwise.

2.8 The Header File `bibtool/entry.h`

This module provides also access to the functions and variables defined in `entry.c`. Consult also the documentation of this file for details.

This header file automatically includes `symbols.h`.

`Symbol * entry_type` Variable

This is an array of strings which represent entry types. They are either built-in or user defined. Use the function `def_entry_type()` to allocate a new entry type and the function `get_entry_type()` to find a certain entry type.

`String EntryName()` Macro

`Entry`	*Index of the entry.*

This is the functional representation of the name component for an entry type. The argument is the index of an entry type. This macro can also be used as lvalue. No range checks are performed.

`int BIB_EOF` Macro

This symbolic constant is returned when a record has to be read and the end of file has been encountered. It is some negative value for which no entry type is defined.

`int BIB_NOOP` Macro

This symbolic constant is returned when a record has to be read and something has been encountered which should be ignored. It is some negative value for which no entry type is defined.

`int BIB_STRING` Macro

This symbolic constant representing a record type of a BibTeX macro (`@String`). This is a special record type which is provided automatically.

`int BIB_PREAMBLE` Macro

This symbolic constant representing a record type of a BibTeX preamble (`@Preamble`). This is a special record type which is provided automatically.

`int BIB_COMMENT` Macro

This symbolic constant representing a record type of a BibTeX comment (`@Comment`). This is a special record type which is provided automatically.

`int BIB_ALIAS` Macro

This symbolic constant representing a record type of a BibTeX alias (`@Alias`) which is proposed for BibTeX 1.0. This is a special record type which is provided automatically.

`int BIB_MODIFY` Macro

This symbolic constant representing a record type of a BibTeX modification rule (`@Modify`) which is proposed for BibTeX 1.0. This is a special record type which is provided automatically.

`int BIB_INCLUDE` Macro

This symbolic constant representing a record type of a BibTeX inclusion instruction (`@Include`) which is proposed for BibTeX 1.0. This is a special record type which is provided automatically

`IsSpecialRecord()` Macro

`Type`	*Record type which should be checked.*

Test whether a given record type denotes a special record. Special records are those defined above. They are provided automatically since BibTeX is supposed to do so as well.

Returns: `TRUE` iff the rcord type denoted a special record.

`IsNormalRecord()` Macro

`Type`	*Record type which should be checked.*

Test whether a given record is a normal record. A normal record is one defined by a user. Normal records have indices larger than those of special records.

Returns: `TRUE` iff the rcord type denoted a normal record.

2.9 The Module `entry.c`

This module contains functions which deal with entry types. Right from the beginning only the special record types are known. Those special record types are `@Comment`, `@Preamble`, `@String`, `@Include`, `@Modify`, and `@Alias`.

In addition to those special records the user can define additional record types which are denoted as “normal”. E.g. usually `@Article` and `@Book` are defined which are “normal” record types.

The record types are are managed in this module. In the other modules only numerical representations are used. This module provides means to map those numerical ids to the string representation and back. It is also possible to define additional record types.

Part of this module is likely to be integrated into databases.

`void def_entry_type()` Function

`Symbol sym;`	*Name of the BibTeX macro to expand.*

Dynamically define an entry type. If the entry type already exists then a new printing representation is stored.

If no memory is left then an error is raised and the program is terminated

Returns: nothing

`rec_type find_entry_type()` Function

`String s;` *String of the potential entry name.*

Look up an entry type in the array of defined entries.

Returns: The index in the array or `NOOP`.

`Symbol get_entry_type()` Function

`int idx;` *Index of entry type.*

Get the printable string representation corresponding to the numerical entry type given as argument. If no entry type is defined for the given index then `NULL` is returned.

Returns: Print representation of the entry type or `NULL`.

`void init_entries()` Function

Predefine some entry types which are stored at startup time in an array. The following entry types are predefined because they are considered special by BIBTEX:

BIB_STRING denotes a BIBTEX macro definition.

BIB_PREAMBLE denotes a preamble item which goes before the bibliography environment.

BIB_COMMENT denotes a comment entry which is passed to the output file.

BIB_ALIAS denotes an alias entry which renames an existing entry.

BIB_MODIFY denotes a modification request which alters an existing entry.

BIB_INCLUDE denotes an include request which reads in another BIBTEX file.

Note that this function is for internal purposes only. The normal user should call `init_bibtool()` instead.

Returns: nothing

2.10 The Header File `bibtool/error.h`

This header file provides means for issuing error messages. Most of the macros provided in this header file are based on the function `error()` described in `error.c`. Nevertheless this function covers the general cases the macros in this header file are more convenient since they hide the unneccesary arguments of the `error()` function providing appropriate values.

This header file makes availlable the function `error()` as defined in `error.c`.

`int ERR_NONE` Macro

No error flags.

int ERR_ERROR Macro

Error type: Indicate that the error can not be suppressed and the messaged is marked as error.

int ERR_WARNING Macro

Error type: Indicate that the error is in fact a warning which can be suppressed. The messaged is marked as warning. This flag is only in effect if the `ERR_ERROR` flag is not set.

int ERR_MESSAGE Macro

Error type: Indicate that the error is in fact a message.

int ERR_POINT Macro

Error type: Indicate that the line and the error pointer should be displayed (if not suppressed via other flags).

int ERR_FILE Macro

Error type: Indicate that the file name and line number should be displayed (if not suppressed via other flags).

int ERR_NO_NL Macro

Error type: Indicate that the terminating newline should be suppressed.

int ERR_EXIT Macro

Error type: Indicate that the `error()` function should be terminated by `exit()` instead of returning.

void ERROR_EXIT() Macro

X	*Error message.*

Raise an error, print the single string argument as error message and terminate the program with `exit()`.

Returns: nothing

void OUT_OF_MEMORY() Macro

X	*String denoting the type of unallocatable memory.*

Raise an error because `malloc()` or `realloc()` failed. The argument denoted the type of memory for which the allocation failed. The program is terminated.

Returns: nothing

void ERROR() Macro

X	*Error message.*

Raise an error. Print the argument as error message and continue.

Returns: nothing

void ERROR2() Macro

X	*First error message.*
Y	*Continuation of the error message.*

Raise an error. Print the two arguments as error message and continue.

Returns: nothing

`void` **`ERROR3()`** Macro

`X`	*First error message.*
`Y`	*Continuation of the error message.*
`Z`	*Second continuation of the error message.*

Raise an error. Print the three arguments as error message and continue.

Returns: nothing

`void` **`WARNING()`** Macro

`X`	*Warning message.*

Raise a warning. Print the argument as warning message and continue.

Returns: nothing

`void` **`WARNING2()`** Macro

`X`	*First warning message.*
`Y`	*Continuation of warning message.*

Raise a warning. Print the two arguments as warning message and continue.

Returns: nothing

`void` **`WARNING3()`** Macro

`X`	*First warning message.*
`Y`	*Continuation of warning message.*
`Z`	*Second continuation of warning message.*

Raise a warning. Print the thre arguments as warning message and continue.

Returns: nothing

`void` **`Err()`** Macro

`S`	*String to print.*

Print a string to the error stream. This message is preceded with an indicator. The message is *not* automatically terminated by a newline.

Returns: nothing

`void` **`ErrC()`** Macro

`CHAR`	*Character to send to output.*

Print a single character to the error stream.

Returns: nothing

`void` **`ErrPrint()`** Macro

`F`	*String to print.*

Print a string to the error stream. The string is not preceded by any indicator not is it automatically terminated by a newline.

Returns: nothing

`void` **`ErrPrintF()`** Macro

`F`	*Format.*
`A`	*Argument.*

Apply a formatting instruction (with `printf()`). This macro takes a format string and a second argument which is determined by the formatting string.

Returns: nothing

`void` **`ErrPrintF2()`** Macro

`F`	*Format*
`A`	*First argument.*
`B`	*Second argument.*

Apply a formatting instruction (with `printf()`). This macro takes a format string and two additional arguments which are determined by the formatting string.

Returns: nothing

`void` **`ErrPrintF3()`** Macro

`F`	*Format*
`A`	*First argument.*
`B`	*Second argument.*
`C`	*Third argument.*

Apply a formatting instruction (with `printf()`). This macro takes a format string and three additional arguments which are determined by the formatting string.

Returns: nothing

`void` **`FlushErr`** Macro

Flush the error stream. This can be useful when single characters are written to an error stream which does buffering.

`void` **`VerbosePrint1()`** Macro

`A`	*Verbose message.*

Print an informative message to the error stream.

Returns: nothing

`void` **`VerbosePrint2()`** Macro

`A`	*Verbose message.*
`B`	*Continuation of verbose message.*

Print an informative message consisting of two substrings to the error stream.

Returns: nothing

`void` **`VerbosePrint3()`** Macro

`A`	*Verbose message.*
`B`	*Continuation of verbose message.*
`C`	*Second continuation of verbose message.*

Print an informative message consisting of three substrings to the error stream.

Returns: nothing

void VerbosePrint4() Macro

`A`	*Verbose message.*
`B`	*Continuation of verbose message.*
`C`	*Second continuation of verbose message.*
`D`	*Third continuation of verbose message.*

Print an informative message consisting of four substrings to the error stream.

Returns: nothing

void DebugPrint1() Macro

`A`	*Debug message.*

This Macro is for debugging purposes. The compilation determines whether this macro prints its argument or simply ignores it. This is achieved by defining or undefining the macro `DEBUG` when compiling.

Returns: nothing

void DebugPrint2() Macro

`A`	*Debug message.*
`B`	*Continuation of the debug message.*

This Macro is for debugging purposes. The compilation determines whether this macro prints its arguments or simply ignores them. This is achieved by defining or undefining the macro `DEBUG` when compiling.

Returns: nothing

void DebugPrint3() Macro

`A`	*Debug message.*
`B`	*Continuation of the debug message.*
`C`	*Second continuation of the debug message.*

This Macro is for debugging purposes. The compilation determines whether this macro prints its arguments or simply ignores them. This is achieved by defining or undefining the macro `DEBUG` when compiling.

Returns: nothing

void DebugPrintF1() Macro

`A`	*Debug message.*

This Macro is for debugging purposes. The compilation determines whether this macro prints its argument or simply ignores it. This is achieved by defining or undefining the macro `DEBUG` when compiling.

Returns: nothing

void DebugPrintF2() Macro

F	*The format for the debug message.*
A	*Debug message.*

This Macro is for debugging purposes. The compilation determines whether this macro prints its arguments or simply ignores them. This is achieved by defining or undefining the macro `DEBUG` when compiling.

Returns: nothing

`void` **`DebugPrintF3()`** Macro

F	*The format for the debug message.*
A	*Debug message.*
B	*Continuation of the debug message.*

This Macro is for debugging purposes. The compilation determines whether this macro prints its arguments or simply ignores them. This is achieved by defining or undefining the macro `DEBUG` when compiling.

Returns: nothing

2.11 The Module `error.c`

To ensure a consistent appearence of error messages BibTool provides one generic error reporting routine. This routine is controlled by several arguments to allow maximum flexibility.

Usually it is awkward to fill out all those arguments. To avoid this trouble the header file `error.h` provides some macros which cover the most common situation and hide unneccesary details.

`void` **`err_location()`** Function

`int`	**`lineno;`**	*the line number*
`String`	**`fname;`**	*the file name*
`char*`	**`s1;`**	*the optional postfix string*

Print the error location to the error stream.

Returns: nothing

`void` **`error()`** Function

`int`	**`type;`**	*Error type: boolean combination of the error bits as defined in* `error.h`*.*
`String`	**`s1;`**	*1st error message or* `NULL`*.*
`String`	**`s2;`**	*2nd error message or* `NULL`*.*
`String`	**`s3;`**	*3rd error message or* `NULL`*.*
`String`	**`line;`**	*Current line when error occured (for reading errors).*
`String`	**`err_pos;`**	*Error position in line* `line`*.*
`int`	**`line_no;`**	*The line number where the error occurred.*
`Symbol`	**`fname;`**	*The file name where the error occurred.*

This is the generic error printing routine. It prints an error message together with an optional filename, the line number, the erroneous line and a pointer to the problematic position.

All parts of an error message are optional and can be suppressed under certain conditions. The error type determines which parts are actually shown. It is a boolean combination of the following flags which are defined in `error.h`:

ERR_ERROR If this bit is set then the error message is marked as "error". The flag `ERR_WARNING` is ignored in this case. This kind of messages can not be suppressed.

ERR_WARNING If this bit is set and `ERR_ERROR` is not set then the error message is marked as "warning". `ERR_WARNING` is ignored in this case.

ERR_POINT If this bit is set then the line `line` is shown together with a pointer to the byte pointed to by `err_pos`. Otherwise the line is not shown.

ERR_FILE If this bit is set then the name of the file `file_name` and the line number `lineno` are shown. Otherwise the file name and the line number are suppressed.

ERR_EXIT If this bit is set then the error routine calls `exit(-1)` at the end. This implicitly sets the `ERR_ERROR` bit as well.

The error message itself can be split in up to three strings `s1`, `s2`, and `s3`. Those strings are concatenated. They can also be `NULL` in which case they are ignored.

The error message is written to the stream determined by the variable `err_file`. This variable refers to the `stderr` stream initially but can be redirected to any other destination.

Returns: nothing

`void` **`init_error()`** Function

`FILE *` **`file;`**	*the output file to write error messages to*

Initialize the error reporting.

Returns: nothing

2.12 The Header File `bibtool/expand.h`

This header file makes available the function defined in `expand.c`. This file includes the header file `database.h`.

2.13 The Module `expand.c`

This module contains functions to expand macros as they are appearing in right hand sides of equations. This can be used to get rid of additional macro definitions.

`Symbol` **`expand_rhs()`** Function

`Symbol` **`sym;`**	*Name of the BibTeX macro to expand.*
`Symbol` **`pre;`**	*This is the opening brace character to be used. For BibTeX the valid values are* `{` *or* `"`*. This value has to match to* `post`*.*
`Symbol` **`post;`**	*This is the closing brace character to be used. For BibTeX the valid values are* `}` *or* `"`*. This value has to match to* `pre`*.*
`DB` **`db;`**	*Database containing the macros.*
`bool` **`lowercase;`**	

Expand the right hand side of an item. Each macro which is defined in this database is replaced by its value. The result is kept in a static variable until the next invocation of this function overwrites it.

Returns: A pointer to the expanded string. This value is kept in a static variable of this function and will be overwritten with the next invocation.

2.14 The Header File `bibtool/init.h`

This header file provides the prototype for the global initialization function which is required to be called before any action can be performed.

2.15 The Module `init.c`

This module contains the global initialization function which has to be called before any modules in BibTool are activated. This is for convenience, thus nobody has to call the various initialization functions for the different modules by hand.

`void` **`init_bibtool()`** Function

`char *` **`progname;`**	*Name of the program for* `KPATHSEA`*.*

Perform any initializations necessary for BibTool.

Returns: nothing

2.16 The Header File `bibtool/keynode.h`

This header file provides the datatype of a keynode. This is an internal structure which is used to built parse trees from format specifications. Usually this is done in `key.c` and should not be visible outside.

KeyNode Type

```
typedef struct kEYnODE {
  short int       kn_type;
  short int       kn_pre;
  short int       kn_post;
  Symbol          kn_symbol;
  String          kn_from;
  String          kn_to;
  struct kEYnODE *kn_next;
  struct kEYnODE *kn_then;
  struct kEYnODE *kn_else;
} *KeyNode, SKeyNode;
```

2.17 The Header File `bibtool/key.h`

This header file provides functions to deal with keys as they are defined in `keys.h`.

This header file automaticall includes the header files `database.h` and `sbuffer.h` since datatypes defined there are required.

2.18 The Module `key.c`

`void` **`add_format()`** Function

`char *s;` *Specification string*

Add a key format specification to the current specification. This specification is used for generating new reference keys. Thus the resource `rsc_make_key` is turned on aswell.

Several strings are treated special. If a special format is encountered then the effect is that the old key specification is cleared first before the new format is added:

empty The empty format is activated. This means that the format is cleared and without further action the default key will be used.

long The long format is activated. This means that authors names with initials and the first word of the title are used.

short The short format is activated. This means that authors last names and the first word of the title are used.

new.long This means that the long format will be used but only if the record does not have a key already.

new.short This means that the short format will be used but only if the record does not have a key already.

Returns: nothing

`void add_ignored_word()` Function

`Symbol word;`	*Word to add.*

Add a new word to the list of ignored words for title key generation. The argument has to be saved by the caller! This means that it is assumed that the argument is a symbol.

Returns: nothing

`void add_sort_format()` Function

`char *s;`	*Specification string*

Add a sort key format specification to the current specification. This specification is used for generating new sort keys.

Several strings are treated special. If a special format is encountered then the effect is that the old key specification is cleared first before the new format is added:

empty The empty format is activated. This means that the format is cleared and without further action the default key will be used.

long The long format is activated. This means that authors names with initials and the first word of the title are used.

short The short format is activated. This means that authors last names and the first word of the title are used.

new.long This means that the long format will be used but only if the record does not have a key already.

new.short This means that the short format will be used but only if the record does not have a key already.

Returns: nothing

`int apply_fmt()` Function

`StringBuffer`	`*sb;`	*Destination string buffer.*
`char *`	`fmt;`	*Format specification,*
`Record`	`rec;`	*Record to consider.*
`DB`	`db;`	*Database containing* `rec`.

Expands an arbitrary format specification for a given record. The format specification is given as a string. The result is stored in a string buffer.

Returns: `1` iff the format is invalid or the evaluation fails. `0` otherwise.

`void clear_ignored_words()` Function

Delete the list of ignored words. Afterwards no words are recognized as ignored words.

Returns: nothing

`void` **`def_format_type()`** Function

`String s;`

Returns: nothing

`void` **`end_key_gen()`** Function

Finalize the key generation. Any previously recorded keys are discarded.

Returns: nothing

`String` **`fmt_expand()`** Function

`StringBuffer`	**`*sb;`**	*destination string buffer*
`Uchar *`	**`cp;`**	*format*
`DB`	**`db;`**	*Database containing* `rec`.
`Record`	**`rec;`**	*Record to consider.*

Expands a format specification of the string buffer.

Returns: The first character after the

`bool` **`foreach_ignored_word()`** Function

`bool (*fct)(Symbol);`	*Function to apply.*

Iterator a given function `fct` is applied to each ignored word in turn. If the function returns 0 then the loop is terminated. The different words are visited in a fixed order which does not necessarily coincide with the natural order of words. Thus don't assume this.

Returns: The return status of the last `fct` call.

`void` **`free_key_node()`** Function

`KeyNode` **`kn;`**	*KeyNode to be freed.*

A tree rooted at a given `KeyNode` will be freed.

Returns: nothing

`String` **`get_base()`** Function

Getter for the printable representation of the key_base.

Returns:

`Symbol` **`get_field()`** Function

`DB`	**`db;`**	*Database containing* `rec`.
`Record`	**`rec;`**	*Record to analyze.*
`Symbol`	**`name;`**	*Field name to search for. This has to be a symbol if a normal field is sought. For pseudo fields it can be an arbitrary string.*

Evaluate the record `rec`. If name starts with `@` then check the record name. If name starts with `$` then return the special info. Else search in Record `rec` for the field name and return its value. `NULL` is returned to indicate failure.

Returns: The address of the value or `NULL`.

`Symbol` **`get_separator()`** Function

`int` **`n;`**	*the index*

Getter for the key separator. The elements under the index have the following meaning:

0 The default key which is used when the formatting instruction fails completely.

1 The separator which is inserted between different names of a multi-authored publication.

2 The separator inserted between the first name and the last name when a name is formatted.

3 The separator inserted between the last names when more then one last name is present

4 The separator between the name and the title of a publication.

5 The separator inserted between words of the title.

6 The separator inserted before the number which might be added to disambiguate reference keys.

7 The string which is added when a list of names is truncated. (`.ea`)

Returns: the separator for the given index or NULL

`void` **`make_key()`** Function

`DB` **`db;`**	*Database containing the record.*
`Record` **`rec;`**	*Record to consider.*

Generate a key for a given record.

Returns: nothing

`void` **`make_sort_key()`** Function

`DB` **`db;`**	*Database containing the record.*
`Record` **`rec;`**	*Record to consider.*

Returns: nothing

`bool` **`mark_key()`** Function

`DB` **`db;`**	*Database containing the record.*
`Record` **`rec;`**	*Record to consider*

Set the key mark for the key symbol of a record.

Returns: nothing

`void` **`set_base()`** Function

`String` **`value;`**	*String representation of the new value.*

Define the key base. This value determines the format of the disambiguation string added to a key if required. The following values are considered:

- If the value is `upper` or starts with an upper case letter then the disambiguation is done with upper-case letters.
- If the value is `lower` or starts with a lower case letter then the disambiguation is done with lower-case letters.
- If the value is `digit` or starts with an digit then the disambiguation is done with arabic numbers.

The comparison of the keywords is done case insensitive. The special values take precedence before the first character rules.

If an invalid value is given to this function then an error is raised and the program is terminated.

Returns: nothing

`bool` **`set_field()`** Function

`DB` **`db;`**	*Database containing* `rec`.
`Record` **`rec;`**	*Record to receive the value.*
`Symbol` **`name;`**	*Field name to add.*
`Symbol` **`value;`**	*the new value*

Store the given field or pseudo-field in a record. If the field is present then the old value is overwritten. Otherwise a new field is added. Fields starting with a `$` or `@` are treated special. They denote pseudo fields. If such a pseudo field is undefined then the assignment simply fails.

In contrast to the function `push_to_record()` this function does not assume that the arguments are symbols. In addition to `push_to_record()` it also handles pseudo-fields.

Returns: `false` if the asignment has succeeded.

`void` **`set_separator()`** Function

`int` **`n;`**	*Array index to modify.*
`String` **`s;`**	*New value for the given separator. The new value is stored as a symbol. Thus the memory of* `s` *need not to be preserved after this function is completed. The characters which are not allowed are silently suppressed.*

Modify the `key\_seps` array. This array contains the different separators used during key formatting. The elements of the array have the following meaning:

0 The default key which is used when the formatting instruction fails completely.

1 The separator which is inserted between different names of a multi-authored publication.

2 The separator inserted between the first name and the last name when a name is formatted.

3 The separator inserted between the last names when more then one last name is present

4 The separator between the name and the title of a publication.

5 The separator inserted between words of the title.

6 The separator inserted before the number which might be added to disambiguate reference keys.

7 The string which is added when a list of names is truncated. (`.ea`)

Returns: nothing

`void start_key_gen()` Function

Start the key generation. Any recorded keys are discarded.

Returns: nothing

2.19 The Header File `bibtool/macros.h`

This header file contains definitions for the `Macro` structure. `Macro` is the pointer type corresponding to the structure `SMacro`. All C macros and functions provided through this header file deal with the pointer type. The structure itself is used in the allocation function only.

Macro Type

This is a pointer type to represent a mapping from a string to another string. This mapping is accompanied by a counter which can be used as a reference count.

```
typedef struct mACRO {
  Symbol        mc_name;    Name of the macro.
  Symbol        mc_value;   Value of the macro.
  int           mc_used;    Reference count.
  struct mACRO *mc_next;    Pointer the next macro.
} SMacro, *Macro;
```

`Macro MacroNULL` Macro

This is the `NULL` pointer for the `Macro` type. It can be used as a special or illlegal macro.

`String MacroName()` Macro

`M` `Macro` *to consider*

This is the functional representation of the name component of a `Macro`. It can be used to extract this information. It can also be used as a lvalue.

`String MacroValue()` Macro

`M` `Macro` *to consider*

This is the functional representation of the value component of a `Macro`. It can be used to extract this information. It can also be used as a lvalue.

int **MacroCount()** Macro

`M`	`Macro` *to consider*

This is the functional representation of the counter component of a `Macro`. It can be used to extract this information. It can also be used as a lvalue.

Macro **NextMacro()** Macro

`M`	`Macro` *to consider*

This is the functional representation of the next `Macro`. It can be used to extract this information. It can also be used as a lvalue.

2.20 The Module `macros.c`

void **def_field_type()** Function

`String s;`	*String containing an equation. This string is modified during the process.*

This function adds a printing representation for a field name to the used list. The argument is an equation of the following form

type = value

type is translated to lower case and compared against the internal representation. *value* is printed at the appropriate places instead.

Returns: nothing

int **def_macro()** Function

`Symbol name;`	*name of the macro.*
`Symbol val;`	*NULL or the value of the new macro*
`int count;`	*initial count for the macro.*

Define or undefine a macro.

Returns: nothing

void **dump_mac()** Function

`char *fname;`	*File name of the target file.*
`int allp;`	*if == 0 only the used macros are written.*

Write macros to a file.

Returns: nothing

bool **each_macro()** Function

`Macro m;`	*the macro to start with*
`bool (*fct) (Symbol ,Symbol);`	

Iterate over a linked list of macros. A function is applied to each macro found. The loop terminates if the function returns `false`.

Returns: `true` if the function has terminated the loop and `false` in case the end of the list has been reached

`void foreach_macro()` Function

`bool (*fct) (Symbol ,Symbol );`

Apply a function to each macro in turn. The function is called with the name and the value of the macro. If it returns `false` then the processing of further macros is suppressed.

The function given as argument is called with two string arguments. The first is the name of the macro and the second is its value. Both are symbols and must not be modified in any way.

The order of the enumeration of the macros is determined by the implementation. No specific assumptions should be made about this order.

Returns: nothing

`void free_macro()` Function

`Macro mac;`	*First Macro to release.*

Free a list of macros. The memory allocated for the `Macro` given as argument and all structures reachable via the `NextMacro` pointer are released.

Returns: nothing

`Symbol get_item()` Function

`Symbol name;`	*Symbol to get the print representation for.*
`int type;`	*One of the values* `SYMBOL\_TYPE\_UPPER`*,* `SYMBOL\_TYPE\_LOWER`*, or* `SYMBOL\_TYPE\_CASED` *as they are defined in* `type.h`*.*

Return the print representation of a BIBTEX string. The appearance is determined by the `items` mapping. If no appropriate entry is found then `type` is used to decide whether the item should be returned as upper-case, lower-case or first upper only.

Returns: A pointer to a static string. This location is reused upon the next invocation of this function.

`Symbol get_key()` Function

`Symbol name;`	*the name of the key to find. This must be in lower-case*

Get the printable representation of a key. If a special representation has been registered then this representation is returned. Otherwise the (lower-case) key is returned.

Returns: the requested representation

`void init_macros()` Function

Initialize some macros from a table defined in the configuration file or given as define to the C compiler. This function has to be called to initialize the global macros.

Note that this function is for internal purposes only. The normal user should call `init_bibtool()` instead.

Returns: nothing

`Symbol` **`look_macro()`** Function

`Symbol` **`name;`**	*The name of the macros to find. This needs not to be a symbol.*
`int` **`add;`**	*Initial reference count or indicator that no new macro is required.*

Return the value of a macro if it is defined. This value is a symbol. If the macro is undefined then `NULL` is returned. In this case the value of `add` determines whether or not the macro should be defined. If it is less than 0 then no new macros is defined. Otherwise a new macro is defined. The value is the empty string and the initial reference count is `add`.

Returns: The value or `NULL`.

`Macro` **`new_macro()`** Function

`Symbol` **`name;`**	*Name of the macro. This must be a symbol.*
`Symbol` **`value;`**	*The value of the macro. This must be a symbol.*
`int` **`count;`**	*The initial reference count.*
`Macro` **`next;`**	*The next pointer of the* `Macro` *structure.*

Allocate a new macro structure and fill it with initial values. Upon failure an error is raised and `exit()` is called.

Returns: The new `Macro`.

`void` **`save_key()`** Function

`Symbol` **`name;`**	*the name of the key in lower*
`Symbol` **`key;`**	*the key as printed*

Save a mapping of a lower-case key to a printed representation.

Returns: nothing

2.21 The Header File `bibtool/names.h`

SNameNode Type

The name format is translated internally into a list of nodes which are easier to evaluate since they avoid the reparsing of the format. This structure contains such a node.

`typedef struct` **nameNODE** `{`

```
  int              nn_type;
  int              nn_strip;
  int              nn_trim;
  Symbol           nn_pre;
  Symbol           nn_mid;
  Symbol           nn_post;
  struct nameNODE *nn_next; Pointer to the next name node
} SNameNode, *NameNode;
```

`NameNode` **`NameNULL`** Macro

The `NULL` pointer to a `NameNode` which can be used as a special value to indicate the end of a `NameNode` list.

`int` **`NameType()`** Macro

`NN`	*the name node*

Returns:

`int` **`NameStrip()`** Macro

`NN`	*the name node*

Returns:

`int` **`NameTrim()`** Macro

`NN`	*the name node*

Returns:

`String` **`NamePre()`** Macro

`NN`	*the name node*

Returns:

`String` **`NameMid()`** Macro

`NN`	*the name node*

Returns:

`String` **`NamePost()`** Macro

`NN`	*the name node*

Returns:

`NameNode` **`NextName()`** Macro

`NN`	`NameNode` *to consider.*

Functional representation of the pointer to the next `NameNode`.

Returns: The next `NameNode`.

2.22 The Module `names.c`

`NameNode` **`name_format()`** Function

`String` **`s;`**

Returns:

`String` **`pp_list_of_names()`** Function

`String *` **`wa;`**	*Word array of name constituents*
`NameNode` **`format;`**	*the format of the name*
`String` **`trans;`**	*the translation table*
`int` **`max;`**	*the limit for the number of names to be formatted*
`String` **`comma;`**	*”,”*
`String` **`and;`**	*name separator*
`char *` **`namesep;`**	
`char *` **`etal;`**	

Pretty-print a list of names.

Returns: Pointer to static string which is reused upon the next invocation of this function.

`void` **`pp_one_name()`** Function

Returns: nothing

`void` **`set_name_format()`** Function

`NameNode` **`*nodep;`**

`char *` **`s;`**

Returns: nothing

2.23 The Header File `bibtool/parse.h`

This header file contains functions which deal with the parsing of BibTeX files. They are defined in `parse.c` and declared in this file.

2.24 The Module `parse.c`

`void` **`init_read()`** Function

Initialize the reading apparatus. Primarily try to figure out the file search path.

Note that this function is for internal purposes mainly. The normal user should call `init_bibtool()` instead. Just in case the search paths are changed afterwards this function has to be called again to propagate the information.

Returns: nothing

int parse_bib() Function

`Record rec;` *Record to store the result in.*

Read one entry and fill the internal record structure. Return the type of the entry read.

`BIB_EOF` is returned if nothing could be read and the end of the file has been encountered.

`BIB_NOOP` is returned when an error has occurred. This is an indicator that no record has been read but the error recovery is ready to try it again.

This function is for internal purposes mainly. See `read_db()` for a higher level function to read a database.

Returns: The type of the entry read, `BIB_EOF`, or `BIB_NOOP`.

bool read_rsc() Function

`String name;` *Name of the file to read from.*

Read a resource file and evaluate all instructions contained.

The characters `#`, `%`, and `;` start an endline comment but only between resource instructions. They are not recognized between a resource instruction and its value or inside the value braces.

This function is contained in this module because it shares several functions with the BibTeX parsing routines.

Returns: `true` if an error has occured

bool see_bib() Function

`String fname;` *Name of the file or* `NULL`.

Open a BibTeX file to read from. If the argument is `NULL` then `stdin` is used as input stream.

This function has to be called before `parse()` can be called. It initializes the parser routine and takes care that the next reading is done from the given file.

The file opened with this function has to be closed with `seen()`.

This function is for internal purposes mainly. See `read_db()` for a higher level function to read a database.

Returns: `true` iff the file could be opened for reading.

bool seen() Function

Close input file for the BibTeX reading apparatus. After this function has been called `parse()` might not return sensible results.

This function is for internal purposes mainly. See `read_db()` for a higher level function to read a database.

Returns: `false` if an attempt was made to close an already closed file.

`void` **`set_rsc_path()`** Function

`String` **`val;`**	*The string representation of the file search path.*

Initialize the resource file reading apparatus. Primarily try to figure out the file search path.

Returns: nothing

2.25 The Header File `bibtool/print.h`

This header file provides access to the functions and variables defined in `print.c`. Consult also the documentation of this file for details.

This header file automatically includes `record.h` and `database.h`.

2.26 The Module `print.c`

This module provides also access to the functions and variables defined in `entry.c`. Consult also the documentation of this file for details.

`void` **`fput_record()`** Function

`FILE *` **`file;`**	*Stream to print onto.*
`DB` **`db;`**	*Database containing the record.*
`Record` **`rec;`**	*Record to print.*
`String` **`start;`**	*Initial string used before the type. Should be "@" normally.*

Format and print a complete record onto a given stream. for further details see `put_record()`.

Returns: nothing

`String` **`get_symbol_type()`** Function

Getter for the symbol type.

Returns: one of the values SYMBOL_TYPE_UPPER, SYMBOL_TYPE_LOWER, or SYMBOL_TYPE_CASED as defined in header type.h

`void` **`put_record()`** Function

`int (*` **`fct`**`)(int);`	*function to use for writing a character.*
`Record` **`rec;`**	*Record to print.*
`DB` **`db;`**	*Database containing the record.*
`String` **`start;`**	*Initial string used before the type. Should be "@" normally.*

Format and print a complete record. The record type and several resources are taken into account. The following external variables (from `rsc.c`) are taken into account:

rsc_parentheses If this boolean variable is `true` then `(` and `)` are used to delimit the record. Otherwise `{` and `}` are used.

rsc_col_p This integer variable controls the indentation of preamble records.

rsc_col_s This integer variable controls the indentation of string records.

rsc_expand_macros If this boolean variable is set then macros are expanded before the record is printed. This does not effect the internal representation.

rsc_col This integer variable controls the indentation of normal records.

rsc_col_key This integer variable controls the indentation of the key in a normal record.

rsc_newlines This integer variable controls the number of newlines printed after a normal record.

rsc_linelen This integer variable controls the length of the line. The line breaking algorithm is applied if this column is about to be violated.

rsc_indent This integer variable controls the indentation of equations.

rsc_eq_right This boolean variable controls the alignment of the `=` in equations. It it is set then the equality sign is flushed right. Otherwise it is flushed left.

The field in the record are sorted with `sort_record()` before they are printed.

In normal records all fields not starting with an allowed character are ignored. Thus it is possible to store private and invisible information in a field. Simply start the field name with an not allowed character like `%`.

Returns: nothing

`void set_symbol_type()` Function

`String s;` *String description of the value.*

Function to set the symbol type which is used by the printing routine. The argument is a string describing the value to use. Possible values are `"upper"`, `"lower"`, and `"cased"`. The comparison of the values is performed case insensitive.

If no appropriate value is found then an error message is issued as the only action.

This function is called from

2.27 The File `rsc.c`

Returns: nothing

`char * sput_record()` Function

`DB     db;`	*Database containing the record.*
`Record rec;`	*Record to print.*
`String start;`	*Initial string used before the type. Should be ”@” normally.*

Format and print a complete record into a string and return it. The string returned points to static memory which is reused upon the next invocation of this function.

Returns: The string containing the printed representation.

2.28 The Header File `bibtool/pxfile.h`

This module provides access to the functions and variables defined in `pxfile.c`. Consult also the documentation of this file for details.

This header file automatically includes `bibtool.h` and `<stdio.h>`.

2.29 The Module `pxfile.c`

This file provides routines for extended file opening. Files are sought in a list of directories and optionally with a set of extensions appended to them.

Patterns may be given which are used to determine the full file name. The patterns are stored in a special data structure. A function is provided to allocate a pattern structure and fill it from a string specification.

`px_filename` Variable

This variable contains the file name actually used by the last `px_fopen()` call. The memory is automatically managed and will be reused by the next call to `px_fopen()`. Thus if you need to use it make a private copy immediately after the call to the function `px_fopen()`.

`FILE * px_fopen()` Function

`char * name;`	*(base) name of the file to open.*
`char * mode;`	*Mode for opening the file like used with* `fopen()`.
`char **pattern;`	*A* `NULL` *terminated array of patterns.*
`char **path;`	*The* `NULL` *terminated array of directories.*
`int (* show)(char*);`	*A function pointer or* `NULL`.

Open a file using path and pattern.

Returns: A file pointer refering to the file or `NULL`.

`char ** px_s2p()` Function

`char * s;`	*String to analyze*
`int    sep;`	*Separator*

Translate a path string specification into an array of the components. The memory of the array is malloced and should be freed when not used any longer.

Returns: The array of the components

2.30 The Header File `bibtool/record.h`

This module contains functions which deal with records in databases.

Record Type

This data type represents a record in a BibTeX database. Since the record can contain an arbitrary number of fields the central rôle is taken by the dynamic array `rc_heap`. This array contains at even positions the name of the field and the following odd position the associated value. In normal records the position 0 contains the reference key of the record.

If a field is deleted then the name is replaced by a `NULL`. The structure member `rc_free` contains the size of the heap.

The type of the record is determined by the integer `rc_type`.

```
typedef struct rECORD {
  Symbol          rc_key;     The sort key.
  Symbol          rc_old_key; The old sort key.
  rec_type        rc_type;    The type of the record.
  int             rc_flags;   Some bits; e.g.  used during selecting aux
                              records.
  int             rc_free;    The size of the heap.  This is purely
                              internal and must not be modified.
  Symbol *        rc_heap;    The heap.
  Symbol          rc_comment; The comment following the given record.
  Symbol          rc_source;  The source of the record.  I.e.  the file
                              name it has been read from.
  int             rc_lineno;  Line number or -1.
  struct rECORD *rc_next;     Pointer to the next record.
  struct rECORD *rc_prev;     Pointer to the previous record.
} SRecord, *Record;
```

`Record RecordNULL` Macro

Symbolic constant for the NULL pointer of type `Record`. This is used as special (invalid) record.

`rec_type RecordType()` Macro

`R`	*Record to consider.*

Functional representation of the record token. This can be used to access the token component of a record. It can also be used as lvalue.

Returns: The pure token.

`int RecordFlags()` Macro

`R`	*Record to consider.*

Functional representation of the record type. This can be used to access the token component of a record. It can also be used as lvalue.

Returns: The flags as integer.

`int RecordFlagMARKED` Macro

Bit mask for the `MARKED` flag of a record. The mark is used temporarily to determine certain records; e.g. during gc.

This macro is usually not used directly but implicitly with other macros from this header file.

`int RecordFlagXREF` Macro

Bit mask for the `XREF` flag of a record. This flag is maintained to indicate that the record contains an `crossref` field. This is done for efficiency reasons only.

This macro is usually not used directly but implicitly with other macros from this header file.

`int RecordFlagDELETED` Macro

Bit mask for the `DELETED` flag of a record. This flag indicates that the record has been deleted. To avoid dangling pointers the deleted records are not removed from the database immediately but a call to `record_gc()` performs this cleanup. In the meantime the deleted records are just left in the chain. Many operations automatically ignore deleted records.

This macro is usually not used directly but implicitly with other macros from this header file.

`int SetRecordXREF()` Macro

`R`	*The record to consider.*

Mark the record with the `XREF` flag. If it is marked already nothing is done.

The `XREF` flag is used to mark those records which contain a `crossref` field. This is done for efficiency only.

Returns: The new value of the record flags.

`int ClearRecordXREF()` Macro

`R`	*The record to consider.*

Remove the XREF mark.

Returns: The new value of the record flags.

`int RecordIsXREF()` Macro

`R`	*Record to consider.*

Check whether the `XREF` flag of a record is set.

Returns: `FALSE` iff the `XREF` flag is not set.

`int SetRecordDELETED()` Macro

`R`	*Record to consider.*

Mark the record with the `DELETED` flag. If it is marked already nothing is done.

The `DELETED` flag is used to mark those records which should be treated as non existent. Deleted records are ignored for most operations.

Returns: The new value of the record flags.

`int ClearRecordDELETED()` Macro

`R`	*Record to consider.*

Remove the deleted flag. Thus you can effictively undelete a record as long as its memory has not been released.

Returns: The new value of the record flags.

`int RecordIsDELETED()` Macro

`R`	*Record to consider.*

Check whether the record is marked as deleted.

Returns: `FALSE` iff the `DELETED` flag is not set.

`int SetRecordMARK()` Macro

`R`	*Record to consider.*

Mark the record. The mark is used temporarily. Do not assume that the mark is preserved in each function.

Returns: The new value of the record flags.

`int ClearRecordMARK()` Macro

`R`	*Record to consider.*

Remove the deleted flag. Thus you can effictively undelete a record as long as its memory has not been released.

Returns: The new value of the record flags.

`int RecordIsMARKED()` Macro

`R`	*Record to consider.*

Check whether the record is marked as deleted.

Returns: `FALSE` iff the `DELETED` flag is not set.

`String RecordKey()` Macro

`R`	*Record to consider*

`String RecordOldKey()` Macro

`R`	*Record to consider*

`String RecordSortkey()` Macro

`R` *Record to consider.*

This is the functional representation of the sort key of a record. This can be used to access the key component of a record. It can also be used as lvalue.

Note that the reference key of a normal record is stored in the heap at position 0.

`String * RecordHeap()` Macro

`R` *Record to consider.*

The heap of a record is a array of strings. The even positions contain the names of fields and the following array cell contains its value. If the name or value is `NULL` then this slot is not used. Thus it is easy to delete a field. Simply write a `NULL` into the appropriate place.

`Record NextRecord()` Macro

`R` *Record to consider*

This is the functional representation of the next record of a record. It can be used to get this value as well as an lvalue to set it.

`Record PrevRecord()` Macro

`R` *Record to consider*

This is the functional representation of the previous record of a record. It can be used to get this value as well as an lvalue to set it.

`String RecordComment()` Macro

`R` *Record to consider*

This is the functional representation of the comment component of a record. It can be used to get this value as well as an lvalue to set it.

`String RecordSource()` Macro

`R` *Record to consider*

This is the functional representation of the source indicator of a record. It is a string containing the file name from which this record has been read. The empty string is used to denote unknown sources.

Returns:

`int RecordLineno()` Macro

`R` *Record to consider*

This is the line number where the record has been read from. The value -1 is used for an unknown line number.

Returns:

`int RecordFlags()` Macro

`R` *Record to consider*

This is the functional representation of the record flags. They are extra bits used for arbitrary purposes. Right now only the bit with the mask 1 is used for selecting the records found in an aux file.

Returns:

2.31 The Module `record.c`

`void` **`add_sort_order()`** Function

`Symbol` **`val;`** *string resource of the order.*

Insert the sort order into the order list.

Returns: nothing

`Record` **`copy_record()`** Function

`Record` **`rec;`** *The record to copy.*

Copy a record and return a new instance. If no memory is left then an error is raised and the program is terminated.

Returns: The new copy of `rec`.

`int` **`count_record()`** Function

`Record` **`rec;`** *the record*

Returns:

`void` **`free_1_record()`** Function

`Record` **`rec;`** *record to free*

Free the memory occupied by a single record. This does not ensure that there is no dangling pointer to the record. Thus beware!

Returns: nothing

`void` **`free_record()`** Function

`Record` **`rec;`** *Arbitrary Record in the chain.*

Release a list of records. All records reachable through a previous/next chain are deallocated.

Returns: nothing

`Record` **`new_record()`** Function

`int` **`token;`** *The token type of the record.*

`int` **`size;`** *The initial heap size.*

Create a new record and return it. If no memory is left then an error is raised and the program is terminated.

Returns: The new record.

`WordList` **`new_wordlist()`** Function

`Symbol s;`	*Initial string to fill in the WordList structure*

Allocate a WordList and fill its slots.

Returns:

`void` **`provide_to_record()`** Function

`Record rec;`	*the record*
`Symbol s;`	*Left hand side of the equation.*
`Symbol t;`	*Right hand side of the equation.*

Put an equation s=t onto the heap of a record if the key s is not defined already. If a field s is already there then the value is ignored. The arguments are expected to be symbols. Thus it is not necessary to make private copies and it is possible to avoid expensive string comparisons.

Returns: nothing

`void` **`push_to_record()`** Function

`Record rec;`	*the record*
`Symbol s;`	*Left hand side of the equation.*
`Symbol t;`	*Right hand side of the equation.*
`bool err;`	*indicator that a warning for double fields should be issued.*

Put an equation s=t onto the heap of a record. If a field s is already there then the value is overwritten. The arguments are expected to be symbols. Thus it is not necessary to make private copies and it is possible to avoid expensive string comparisons.

Returns: nothing

`Record` **`record_gc()`** Function

`Record rec;`	*Pointer to any entry in the chain.*

Garbage collecting a record list. The entries marked as deleted are unlinked and the memory is freed. Any pointer to such a deleted entry becomes invalid.

Be careful when using this function!

Returns: Pointer to some entry in the cleared chain or `RecordNULL` if none is left.

`Symbol` **`record_get()`** Function

`Record rec;`	*the record*
`Symbol key;`	*the key*

Returns:

`void` **`sort_record()`** Function

`Record rec;`	*Record to sort*

The heap is reordered according to the sorting order determined by the record type. For this purpose a copy of the original record is made and the original record is overwritten. The copy is released at the end. Memory management is easy since all strings are in fact symbols, i.e. they must not be freed and comparison is done by pointer comparison.

Returns: nothing

`Record` **`unlink_record()`** Function

`Record`	**`rec;`**	*Record to free.*

Remove a record from a chain and free its memory. The chain is modified such that the freed Record is not referenced any more. A neighbor in the chain of the given record is returned or `NULL` if there is none.

Returns: nothing

2.32 The Header File `bibtool/rewrite.h`

2.33 The Module `rewrite.c`

`void` **`add_check_rule()`** Function

`String`	**`s;`**	*Rule to save.*
`int`	**`flags;`**	*the additional rule flags*

Save a check rule for later use.

Returns: nothing

`void` **`add_extract()`** Function

`Symbol`	**`s;`**	*Rule to save.*
`int`	**`regexp;`**	*Boolean value indicating whether regular expressions should be used. If not set then plain string matching is performed.*
`int`	**`notp;`**	*Boolean value indicating whether the result should be negated.*

Save an extraction rule for later use. The argument is interpreted as regular expression to be matched against the field value.

The value of `rsc_case_select` at the invocation of this function determines whether the matching is performed case sensitive or not.

Returns: nothing

`void` **`add_field()`** Function

`String`	**`spec;`**	*A string of the form* `token=value`

Save a token and value for addition.

Returns: nothing

`void` **`add_rewrite_rule()`** Function

`String` **`s;`**	*Rule to save*

Save a rewrite rule for later use. The main task is performed by `add_rule()`.

Returns: nothing

`void` **`clear_addlist()`** Function

Reset the addlist to the empty list.

Returns: nothing

`bool` **`foreach_addlist()`** Function

`bool (*fct) (Symbol key, Symbol val);`

Apply a function for every entry in the `addlist`. If the function returns `false` then the iteration is terminated immediately and `false` returned. Otherwise `true` is returned after all entries have been visited.

Returns: the termination indicator

`char*` **`get_regex_syntax()`** Function

Getter for the regex syntax.

Returns:

`bool` **`is_selected()`** Function

`DB` **`db;`**	*Database containing the record.*
`Record` **`rec;`**	*Record to look at.*

Boolean function to decide whether a record should be considered. These selections are described by a set of regular expressions which are applied. If none are given then the match simply succeeds.

Returns: `true` iff the record is seleced by a regexp or none is given.

`void` **`keep_field()`** Function

`Symbol` **`spec;`**	*the specification*

Returns: nothing

`void` **`remove_field()`** Function

`Symbol` **`field;`**	*This is a symbol containing the name of the field to remove.*
`Record` **`rec;`**	*Record in which the field should be removed.*

Remove the given field from record.

Returns: nothing

`void` **`rename_field()`** Function

`Symbol` **`spec;`**	*the argument*

Returns: nothing

`void` **`rewrite_record()`** Function

`DB      db;`	*The database record is belonging to.*
`Record rec;`	*Actual record to apply things to.*

Apply deletions, checks, additions, and rewriting steps in this order.

Returns: nothing

`void save_regex()` Function

`String s;`	*Regular expression to search for.*

Save an extraction rule for later use. Only the regular expression of the rule is given as argument. The fields are taken from the resource select.fields.

Returns: nothing

`int set_regex_syntax()` Function

`char* name;`

experimental

Returns: nothing

2.34 The Header File `bibtool/resource.h`

This file is the central component of the resource evaluator. To reduce redundancy everything in this file is encapsulated with macros. Thus it is possible to adapt the meaning according to the task to be performed.

This file is included several times from different places. One task is the definition of certain variables used in this file. Another task is the execution of the commands associated with a command name.

This is one place where the power and the beauty of the C preprocessor makes live easy. It should also be fun to figure out the three ways in which this file is used. Read the sources and enjoy it!

For the normal user this file is consulted automatically when the header file `rsc.h` is used.

2.35 The Header File `bibtool/rsc.h`

This header file provides definitions for all resource variables, i.e. the variables defined in the header file `resource.h`.

In addition the functions defined in `resource.c` are made accessible to those modules including this header file.

2.36 The Module `rsc.c`

This module contains functions which deal with resources. Resources are commands to configure the behaviour of BibTool. They can be read either from a file or from a string.

The syntax of resources are modelled after the syntax rules for BibTeX files. See the user's guide for details of the syntax.

`bool load_rsc()` Function

`String name;`	*The name of the resource file to read.*

This function tries to load a resource file. Details: Perform initialization if required. The main job is done by `read_rsc()`. This function is located in `parse.c` since it shares subroutines with the parser.

Returns: `false` iff the reading failed.

`bool resource()` Function

`String name;`	*the name of the resource file*

Returns:

`void rsc_print()` Function

`String s;`	*String to print.*

Print a string to the error stream as defined in `error.h`. The string is automatically augmented by a trailing newline. This wrapper function is used for the resource `print`.

Returns: nothing

`bool search_rsc()` Function

Try to open the resource file at different places:

- In the place indicated by the environment variable `RSC_ENV_VAR`. This step is skipped if the macro `RSC_ENV_VAR` is not defined (at compile time of the module).
- In the home directory. The home directory is determined by an environment variable. The macro `HOME_ENV_VAR` contains the name of this environment variable. If this macro is not defined (at compile time of the module) then this step is skipped.
- In the usual place for resource files.

For each step `load_rsc()` is called until it succeeds.

The files sought is determined by the macro `DefaultResourceFile` at compile time of the module. (see `bibtool.h`)

Returns: `true` iff the resource loading succeeds somewhere.

`bool set_rsc()` Function

`Symbol` **`name;`**	*Name of the resource to set.*
`Symbol` **`val;`**	*The new value of the resource.*

Set the resource to a given value. Here the assignment is divided into two parts: the name and the value. Both arguments are assumed to be symbols.

Returns: `false` iff everything went right.

`bool` **`use_rsc()`** Function

`String` **`s;`**	*String containing a resource command.*

This function can be used to evaluate a single resource instruction. The argument is a string which is parsed to extract the resource command.

This is an entry point for command line options which set resources.

Returns: `true` iff an error has occurred.

2.37 The Header File `bibtool/s_parse.h`

This string parser.

2.38 The Module `s_parse.c`

`Symbol` **`s_parse()`** Function

`int` **`type;`**	*is the type of construct to parse. It is defined in s_parse.h*
`String` **`*sp;`**	*is a pointer to the string which is parsed. The value is changed to hold the remaining characters at the end.*
`bool` **`errp;`**	*this boolean indicated whether or not a verbose error message should be created in case of an error.*

Parse a string for a certain entity. Leading whitespace is ignored. `type` determines which kind of entity should be expected. It can take the following values which are defined in `s_parse.h`:

StringParseValue The string is analyzed and the proper type is determined automatically. This can be considered as the normal way of operation.

StringParseSymbol The string is analyzed and only a symbol is accepted, i.e. a sequence of allowed characters.

StringParseNumber The string is analyzed and only a number is accepted.

StringParseBraces The string is analyzed and only an expression in braces is accepted. The braced contained must come in matching pairs. The whole expression – including the braces – is returned.

StringParseUnquotedBraces The string is analyzed and only an expression in braces is accepted. The braced contained must come in matching pairs. The expression without the outer braces is returned.

StringParseString The string is analyzed and only a string enclosed in double quotes is accepted. The string must contain braces in matching pairs. Double quotes which are inside of braces are not considered as end of the string. The whole string – including the double quotes is returned.

StringParseUnquotedString The string is analyzed and only a string enclosed in double quotes is accepted. The string must contain braces in matching pairs. Double quotes which are inside of braces are not considered as end of the string. The string without the outer double quotes is returned.

If an error occurs or the requested entity is not found then `NULL` is returned. As a side effect `sp` is advanced to point to the next unprocessed character.

The string analyzed should be opened at the beginning with `sp_open()` in order to get an appropriate error message.

This function is usually not called directly but the convenience macros defined in `s_parse.h` should be used instead.

Returns: A symbol containing the requested entity or `NULL`.

`void` **`sp_close()`** Function

Release the string parser and rest it to its initial state.

Returns: nothing

`String` **`sp_eos()`** Function

`String *sp;`	*the string pointer*

The string is analyzed and any remaining characters which are not whitespace are reported as error. A pointer to the terminating 0 byte

Returns:

`bool` **`sp_expect()`** Function

`String *sp;`	*the pointer to the source string*
`String expect;`	*the expected string*
`bool verbose;`	*the indicator whether an error message should be produced*

Read a sequence of expected characters after whitespace. The source pointer is advanced until the expected string has been read or to the first character which is not expected.

Returns: `true` iff the expected string is found

`void` **`sp_open()`** Function

`String s;`	*String to open for parsing.*

Open a string for parsing. The argument string is used for the parsing process. Thus this string should not be modified during this time. Especially it should not be freed if it is a pointer to dynamically allocated memory.

Returns: nothing

void sp_skip() Function

`String *sp;` *the string pointer*

Advance a String pointer to the next character which is not a white-space, equals or hash.

Returns:

Symbol* sp_symbols() Function

`String *sp;` *the pointer to the value to be parsed*

Returns: an array of Symbols

2.39 The Header File `bibtool/stack.h`

This module provides access to the functions defined in the module `stack.c`. The the documentation of this module for details.

2.40 The Module `stack.c`

This module provides a single stack of strings. There are two operations on this stack, namely to push a string onto the stack and a pop operation to get the topmost element from the stack and remove it or to get a signal that the stack is empty.

The stack is implemented as an array which grows on demand. Currently the memory of the stack is not returned to the operating system. This seems to be not problemeatic since this memory is not assumed to be really large. Normally just a few strings are pushed to the stack at any time.

Symbol pop_string() Function

Pop a string from the stack. It the stack is empty then `NULL` is returned. Thus the `NULL` value should not be pushed to the stack since this can be confused with the end of the stack.

Returns: The old top element or `NULL` if the stack is empty.

void push_string() Function

`Symbol s;` *String to push to the stack.*

Push a string onto the stack. Only the memory for the stack is allocated. The string is stored as pointer to existing memory. No copy of the string is made.

If no memory is left then an error is raised and the program is terminated.

Returns: nothing

2.41 The Header File `bibtool/sbuffer.h`

This header file makes accessible the functions to treat strings like streams In addition to the functions defined in `sbuffer.c` one macro is defined here.

`sbputchar()` Macro

`C`	*Character to put.*
`SB`	*Destination string buffer.*

Put the character `C` into the string buffer `SB`.

This macro is not sane. The arguments are expanded several times. Thus they must not contain side effects.

Returns: nothing

2.42 The Module `sbuffer.c`

This module contains functions for dealing with strings of aribtrary size. The allocation of memory is done automatically when more characters are added.

The functions are modeled after the stream functions of C. Currently a `printf`-like function is missing because one was not needed yet and it is not so easy to implement—portably.

The functions in this module are very handy to deal with strings of arbitrary length where the length is not known in advance. E.g. consider the case that a line has to be read from a file `file` and the line length should not be restricted by some artificial boundry. This can be implemented as follows:

```
{ StringBuffer *sb = sb_open();      /* Declare and initialize a string buffer. */
  int c;                             /* Variable to store a single character.   */
  char *s;                           /* Variable to hold the string at the end. */
  while ( (c=fgetc(file) != EOF
          && c != '\n')
  { sbputchar(c,sb); }               /* Store each character in the string buffer.
                                     */
  s = sbflush(sb);                   /* Get the string from the string buffer.  */
  puts(s);                           /* Process the string; e.g. print it.      */
  sb_close(sb);                      /* Free the string buffer.                 */
}
```

Note that the flushing of the string buffer returns a C string which is managed by the string buffer. This memory is freed or reused whenever the string buffer needs to. Thus you should make a private copy of this string if it should survive the next operation of the string buffer. Especially, after the call to `sb_close()` this memory has been returned to the operating system and is not available any more.

`bool sbclose()` Function

`StringBuffer*`	`sb;`	*Pointer to string buffer which should be closed*

Free an old string buffer.

Returns: Return `false` upon failure.

`char*` **`sbflush()`** Function

`StringBuffer*`	`sb;`	*String buffer to close.*

Close a string buffer with a trailing `\0` and reset the current pointer to the beginning. The next write operation starts right at the end. Thus additional write operations will overwrite the terminating byte.

Returns: The string contained in the string buffer as a proper C string.

`StringBuffer*` **`sbopen()`** Function

Allocate a new string buffer. Return a pointer to the new string buffer or `NULL` if none was available.

Returns: pointer to new string buffer or NULL

`bool` **`sbputc()`** Function

`int`	`c;`	*Character to put to the string buffer.*
`StringBuffer*`	`sb;`	*Destination string buffer.*

Push a single character onto a string buffer. In contrast to the macro this function handles the reallocation of the memory. For the user it should not make a difference since the macros uses this function when needed.

When no memory is left then the character is discarded and this action is signaled via the return value.

Returns: `false` if no memory is left.

`bool` **`sbputs()`** Function

`char *`	`s;`	*String to be pushed.*
`StringBuffer*`	`sb;`	*Destination string buffer.*

Push a whole string onto a string buffer.

Returns: `false` if something went wrong.

`void` **`sbrewind()`** Function

`StringBuffer*`	`sb;`	*String buffer to consider.*

Reset the string buffer pointer to the beginning. The next write or read will operate there.

Returns: nothing

`bool` **`sbseek()`** Function

`StringBuffer*`	`sb;`	*String buffer to reposition.*
`int`	`pos;`	*New position of the string buffer.*

Reset the current pointer to the position given. If the position is outside the valid region then `true` is returned and the position is left unchanged.

Returns: `false` if everything went right.

`int sbtell()` Function

`StringBuffer* sb;` *String buffer to consider.*

Return the current pointer to the string buffer position. This can be used with `sbseek()` to reset it.

Returns: The relative byte position of the current writing position. This is an integer offset from the beginning of the string buffer.

2.43 The Header File `bibtool/symbols.h`

This header file contains definitions dealing with symbols.

BIBTOOL uses symbols as the basic representation for strings. Symbols are stored in a symbol table and shared among different instances. Thus the same string occurring at different places has to be stored only once.

Another advantage of symbols is that once you have got two symbols at hand it is rather easy to compare them for equality. A simple pointer comparison is enough. It is not necessary to compare them character by character.

The disadvantage of a symbol is that you can not simply modify it temporarily since it is part of the symbol table. This symbol table would be in an insane state otherwise. Thus you always have to make a copy if you want to modify a symbol.

The functions defined in `symbols.c` are exported with this header file as well.

`void UnlinkSymbol()` Macro

`SYM` *Symbol to release.*

The symbol given as argument is released. In fact the memory is not really freed but one instance is marked as not used any more. At other places the symbol might be still required. The freeing of memory is performed by the garbage collector `sym_gc()`.

Returns: nothing

`Symbol NO_SYMBOL` Macro

The NULL pointer for Symbols

`String s_empty` Variable

Unmodifiable value containing the empty string. This variable needs `init_symbols()` to be called first.

`Symbol sym_empty` Variable

The empty symbol. This is a symbol pointing immediately to a `\0` byte. This needs `init_symbols()` to be called first.

Symbol sym_crossref Variable

The symbol `crossref`. This variable needs `init_symbols()` to be called first.

Symbol sym_xref Variable

The symbol `xref`. This variable needs `init_symbols()` to be called first.

Symbol sym_xdata Variable

The symbol `xdata`. This variable needs `init_symbols()` to be called first.

Symbol sym_space Variable

The symbol with a single space character. This variable needs `init_symbols()` to be called first.

Symbol sym_star Variable

The symbol with a single star character. This variable needs `init_symbols()` to be called first.

Symbol sym_comma Variable

The symbol with a single comma character. This variable needs `init_symbols()` to be called first.

Symbol sym_double_quote Variable

The symbol with a single double quote character ("). This variable needs `init_symbols()` to be called first.

Symbol sym_open_brace Variable

The symbol with a single open brace character. This variable needs `init_symbols()` to be called first.

Symbol sym_close_brace Variable

The symbol with a single close brace character. This variable needs `init_symbols()` to be called first.

Symbol sym_et Variable

The symbol with a single et character (&). This variable needs `init_symbols()` to be called first.

String newString() Macro

`S`	*the source of the bytes*

Create a copy of a given String.

Returns: a newly allocated byte array containing the content of the source

2.44 The Module `symbols.c`

This module contains functions which deal with symbols and general memory management. This module implements a single symbol table.

This module required initialization before all functions can be used. Especially the symbol table does not exist before initialization.

`SYM_PIPE_SIZE` Macro

`void` **`free_sym_array()`** Function

`Symbol *sym_arr;` *symbol array*

Returns: nothing

`void` **`init_symbols()`** Function

Initialize the symbols module. The symbol table is cleared. This is not secure when the symbols have already been initialized because it would lead to a memory leak and a violation of the symbol comparison assumption. Thus this case is caught and nothing is done when the initialization seems to be requested for the second time.

If no more memory is available then an error is raised and the program is terminated.

Note that this function is for internal purposes only. The normal user should call `init_bibtool()` instead.

Returns: nothing

`char *` **`new_string()`** Function

`char * s;` *String to duplicate*

Allocate a space for a string and copy the argument there. Note this is just a new copy of the memory not a symbol!

If no more memory is available then an error is raised and the program is terminated.

Returns: Pointer to newly allocated memory containing a duplicate of the argument string.

`void` **`sym_del()`** Function

`Symbol sym;`

Returns: nothing

`void` **`sym_dump()`** Function

Dump the symbol table to the error stream—see module `error.c`. The symbols are printed according to their hash value and the sequence they are occurring in the buckets. A summary of the memory used is also printed.

Returns: nothing

`Symbol` **`sym_extract()`** Function

`String *sp;` *pointer to the string*

`bool lowercase;` *indicate that lowering is requested*

Extract a symbol from a string.

Returns:

void sym_unlink() Function

`Symbol sym;` *Symbol to be released.*

Free a symbol since it is no longer used. This does not mean that the memory is also freed. The symbol can be static or used at other places. The real free operation requires the garbage collector `sym_gc()` to be called.

If the argument is `NULL` or an arbitrary string (no symbol) then this case is also dealt with.

Returns: nothing

Symbol symbol() Function

`String s;` *String which should be translated into a symbol.*

Add a symbol to the global symbol table. If the string already has a symbol assigned to it then this symbol is returned. If the symbol is not static then the use count is incremented by `count`.

If the symbol does not exist already then a new symbol is added to the symbol table and the use count is initialized to `count`. A negative value for `count` indicates that a static symbol is requested. A static symbol will never be deleted from the symbol table. Static can be used at places where one does not care about the memory occupied.

If no more memory is available then an error is raised and the program is terminated.

See also the macro `symbol()` in `symbols.h` for a convenient alternative to this function.

Returns: The new symbol.

2.45 The Header File bibtool/tex_aux.h

2.46 The Module tex_aux.c

bool apply_aux() Function

`DB db;` *Database to clean.*

This function deletes all entries which are not requested by the recently read aux file. This means that the entry to be kept is either mentioned directly, it is cross-referenced, or all entries are requested with the `\nocite{*}` feature.

Note that the entries are in fact not deleted but only marked as deleted. Thus they can be recovered if necessary.

Returns: `false` iff all entries are kept because of an explicit or implicit star (*).

`bool` **`aux_used()`** Function

`Symbol s;`	*reference key to check*

Check whether a reference key has been requested by the previously read aux file. The request can either be explicit or implicit if a * is used.

Returns: `true` if the reference is used.

`void` **`clear_aux()`** Function

Reset the aux table to the initial state.

Returns: nothing

`bool` **`foreach_aux()`** Function

`bool (fct)(Symbol);`	*funtion to apply*

Apply the function to all words in the citation list of the aux file.

Returns: `cite_star`

`bool` **`read_aux()`** Function

`String fname;`	*The file name of the aux file.*
`void (*fct)(Symbol);`	*A function to be called for each BibTeX file requested.*
`bool verbose;`	*Boolean indicating whether messages should be produced indicating the status of the operation.*

Analyze an aux file. If additional files are requested, e.g. by `\include` instructions in the original source file then those are read as well. Each citation found is remembered and can be queried afterwards. If a `\cite{*}` has been used then only a flag is set and all citation keys are discarded.

The aux file contains also the information about the BibTeX files used. For each such file the function `fct` is called with the file name as argument. This function can arrange things that those BibTeX files are read into a database.

This function has only a very simple parser for the aux file. Thus it can be confused by legal contents. But a similar thing can happen to BibTeX as well.

Returns: `true` iff the file could not be opened.

2.47 The Header File `bibtool/tex_read.h`

This header file provides definitions for the use of functions to immitate the reading apparatus of TeX which are defined in `tex_read.c`.

2.48 The Module `tex_read.c`

This module contains functions which immitate the reading apparatus of TeX. Macro expansion can be performed.

`void TeX_active()` Function

`int c;`	*Character to make active.*
`int arity;`	*Arity of the macro assigned to the active character.*
`String s;`	*Body of the definition as string.*

Assign a macro to an active character. If the character is not active then the catcode is changed.

Returns: nothing

`void TeX_close()` Function

Gracefully terminate the reading of TeX tokens. Any remaining pieces of text which have already been consumed are discarted.

Returns: nothing

`void TeX_def()` Function

`String s;`

Define a macro. The argument is a string specification of the following form:

```
\name[arity]=replacement text
\name=replacement text
```

$0 <= arity <= 9$

Returns: nothing

`void TeX_define()` Function

```
String name;
int    arity;
String body;
```

Add a new TeX macro definition.

Returns: nothing

`void TeX_open_file()` Function

`FILE * file;`	*File pointer of the file to read from.*

Prepare things to parse from a file.

Returns: nothing

`void TeX_open_string()` Function

`String s;`	*String to read from.*

Prepare things to parse from a string.

Returns: nothing

`bool TeX_read()` Function

`String cp;`	*Pointer to position where the character is stored.*
`String *sp;`	*Pointer to position where the string is stored.*

Read a single Token and return it as a pair consisting of an ASCII code and possibly a string in case of a macro token.

Returns: `false` iff everything went right.

`void TeX_reset()` Function

Reset the TEX reading apparatus to its initial state. All macros and active characters are cleared and the memory is released. Thus this function can also be used for this purpose.

Returns: nothing

2.49 The Header File `bibtool/type.h`

This module is a replacement for the system header file `ctype.h`. In contrast to some implementations of the `isalpha` and friends the macros in this header are stable. This means that the argument is evaluated exactly once and each macro consists of exactly one C statement. Thus these macros can be used even at those places where only a single statement is allowed (conditionals without braces) or with arguments containing side effects.

In addition this is a starting point to implement an xord array like TEX has one (some day. . .)

This header file requires the initialization function `init_type()` to be called before the macros will work as described.

This header file also provides the functions and varaibles defined in `type.c`

`char* trans_lower` Variable

Translation table mapping upper case letters to lower case. Such a translation table can be used as argument to the regular expression functions.

`char* trans_upper` Variable

Translation table mapping lower case letters to upper case. Such a translation table can be used as argument to the regular expression functions.

`char* trans_id` Variable

Translation table performing no translation. Thus it implements the identity a translation table can be used as argument to the regular expression functions.

`bool is_allowed()` Macro

`C`	*Character to consider*

Decide whether the character given as argument is an allowed character in the sense of BIBTEX.

Returns: `TRUE` iff the argument is an allowed character.

bool is_upper() Macro

`C` *Character to consider*

Decide whether the character given as argument is a upper case letter. (Characters outside the ASCII range are not considered letters yet)

Returns: `TRUE` iff the character is an uppercase letter.

bool is_lower() Macro

`C` *Character to consider*

Decide whether the character given as argument is a lower case letter. (Characters outside the ASCII range are not considered letters yet)

Returns: `TRUE` iff the character is a lowercase letter.

bool is_alpha() Macro

`C` *Character to consider*

Decide whether the character given as argument is a letter. (Characters outside the ASCII range are not considered letters yet)

Returns: `TRUE` iff the character is a letter.

bool is_digit() Macro

`C` *Character to consider*

Decide whether the character given as argument is a digit. (Characters outside the ASCII range are not considered letters yet)

Returns: `TRUE` iff the character is a digit.

bool is_space() Macro

`C` *Character to consider*

Decide whether the character given as argument is a space character. `'\0'` is not a space character.

Returns: `TRUE` iff the character is a space character.

bool is_extended() Macro

`C` *Character to consider*

Decide whether the character given as argument is an extended character outside the ASCII range.

Returns: `TRUE` iff the character is an extended character.

bool is_wordsep() Macro

`C` *Character to consider*

Decide whether the character given as argument is a word separator which denotes no word constituent.

Returns: `TRUE` iff the character is a word separator.

`char` **`ToLower()`** Macro

`C`	*Character to translate*

Translate a character to it's lower case dual. If the character is no upper case letter then the character is returned unchanged.

Returns: The lower case letter or the character itself.

`char` **`ToUpper()`** Macro

`C`	*Character to translate*

Translate a character to it's upper case dual. If the character is no lower case letter then the character is returned unchanged.

Returns: The upper case letter or the character itself.

2.50 The Module `type.c`

This file contains functions to support a separate treatment of character types. The normal functions and macros in `ctype.h` are replaced by those in `type.h`. This file contains an initialization function which is required for the macros in `type.h` to work properly.

See also the documentation of the header file `type.h` for further information.

`void` **`add_word_sep()`** Function

`String s;`	*the allowed word separator characters*

Mark some characters as word separator.

Returns: nothing

`bool` **`case_eq()`** Function

`String s;`	*First string to consider.*
`String t;`	*Second string to consider.*

Compare two strings ignoring cases. If the strings are identical up to differences in case then this function returns `true`.

Returns: `true` iff the strings are equal.

`int` **`cmp()`** Function

`String s;`	*the first string*
`String t;`	*the second string*

Compare two strings.

Returns:

`void` **`init_type()`** Function

This is the initialization routine for this file. This has to be called before some of the macros in `type.h` will work as described. It does no harm to call this initialization more than once. It just takes some time.

Note that this function is for internal purposes only. The normal user should call `init_bibtool()` instead.

Returns: nothing

`String lower()` Function

`String s;` *string to convert*

Function to translate all letters in a string to lower case.

Returns: The converted string.

2.51 The Header File `bibtool/version.h`

2.52 The Module `version.c`

`char * bibtool_version` Variable

This string variable contains the version number of BibTool. Usually it is of the form *major.minor* where *major* and *minor* are the major and minor version numbers. In addition a post-fix like `alpha` or a patch level like `p1` can be present.

`char * bibtool_year` Variable

This string variable contains the publication year for this version.

`void show_version()` Function

Print the version number and a short copyright notice onto the error stream.

Returns: nothing

2.53 The Header File `bibtool/wordlist.h`

WordList Type

This data type represents a node in a list of strings. This list only provides a next pointer. and is pretty generic.

```
typedef struct wORDlIST {
  Symbol          wl_word; String value of this node.
  struct wORDlIST *wl_next; Pointer to the next node.
} SWordList, *WordList;
```

`WordList WordNULL` Macro

This is the `NULL` value for a `WordList`. It terminates the list and represents the empty node.

`String ThisWord()` Macro

`WL`	`WordList` *to consider which is not* `WordNULL`.

This macro returns the string of a `WordList` node.

Returns: The word stored in this node.

`WordList NextWord()` Macro

`WL`	`WordList` *to consider which is not* `WordNULL`.

This macro returns the next `WordList` node of a given `WordList` if this is not `WordNULL`.

Returns: The next `WordList`.

2.54 The Module `wordlist.c`

This module contains functions which deal with lists of words. Those words are in fact simple strings. Thus this module provides a very general functionality, namely a list of strings and the associated methods.

`void` **`add_word()`** Function

`Symbol sym;`	
`WordList *wlp;`	*Pointer to a wordlist.*

Put a string into a word list. The string itself is *not* copied. Thus it is highly recommended to use symbols as words nevertheless this is not required as long as the string `s` persists as long as the word list exists.

The second argument is a pointer to a `WordList`. This destination is modified by adding a new node. The use of a pointer allows a uniform treatment of empty and not empty word lists.

If no memory is left then an error is raised and the program is terminated.

Returns: nothing

`int` **`delete_word()`** Function

`Symbol sym;`	
`WordList *wlp;`	*Pointer to the word list to modify.*
`void (* fct)(String);`	*Function to call to free the memory occupied by the word.*

Remove a word from a `WordList`. Only the first appearance of such a word is removed. I a word is found which contains the same string as `s` then the associated node is removed from the list and the function `fct` is called to free the memory of the string in the `WordList` node if the function is not `NULL`. In this case the function returns `0`. Otherwise `1` is returned.

Returns: `0` if the word was not found. `1` otherwise.

`bool` **`find_word()`** Function

`String s;`	*String to find.*
`WordList wl;`	*Word list to search in.*

Look up a word in a word list. The comparison is done case insensitive.

Returns: `false` iff the word does not occur in the word list.

`bool` **`foreach_word()`** Function

`WordList wl;`	*WordList to traverse.*
`bool (* fct)(Symbol);`	*function to apply.*

Applies the given function `fct` to all elements in the `WordList` as long as the function does not return 0. Thus it can be used to search for a specified word – e.g. determined by matching against a template. Another application the the processing of all elements in the `WordList`. In this case `fct` must always return `true`.

Returns: return value of last function or `true`.

`void` **`free_words()`** Function

`WordList *wlp;`	*Pointer to the* `WordList`*.*
`void (* fct)(Symbol);`	*Function to be called to free the memory of the word itself. If it is* `NULL` *then no function is called.*

Release the memory allocated for a list of words. All nodes in the list are freed. The function `fct` is called to free the memory occupied by the string component if it is not `NULL`.

Returns: nothing

3 Creating and Using the BibTool C Library

3.1 Creating the BibTool C Library

Creating the BibTool library should not be too hard. Mainly make BibTool in the main directory according to the instructions given there. As a side effect various object files are created. These object files—except the one for main.c—have to be put into the library.

For UNIX this is prepared in the makefile. Usually an invocation of make should be enough:

```
make libbibtool.a
```

This invocation of make is in fact the same as the following two commands:

```
ar r libbibtool.a $OFILES
ranlib libbibtool.a
```

Here `$OFILES` denotes the list of object files as described above. On some systems no ranlib program is present and needed. In this case the second command can be omitted.

For other operating systems I simply do not know how things work there. I would be grateful to receive descriptions what to do there.

3.2 Using the BibTool C Library

If you have written a program which uses the BibTool C Library you have to include the library into the linking list. In addition the directory where the library can be found has to be specified. On UNIX this can be done with the compiler switches `-l` and `-L` respectively. Thus consider you have a program named `mybib.c` and you have created the object file `mybib.o` for it. The linking step can be performed with the following command:

```
cc mybib.o -L$DIR -lbib -o mybib
```

Here `$DIR` denotes the path containing the file `libbibtool.a`. This path can be omitted if the library has been installed in a “standard” place like `/usr/lib`.

4 Coding Standards

Several tools are used for the development of BibTool. Mostly they are home grown—maybe they will be replaced by some wider used tools some day. Among those tools are indentation routines for Emacs to format the comments contained in the source. There is also a Lisp function to generate the function prototypes contained in the header files and sometimes in the C files as well. And finally there is a Program to extract the documentation from the source files and generate a printable manual.

All those support programs rely on standards for coding. Some of those standards have been develped independantly but should be used for consistency. In the following sections these coding standards are described.

4.1 K&R-C vs. ANSI-C

BibTool tries hard to be portable to wide variety of C systems. Thus it can not be assumed that an ANSI C compiler is at hand. As a consequence the function heads are written in the old style which is also tolerated by ANSI compliant compilers. This means that the argument types are given after the argument list.

Here it is essential that the arguments type declarations are given in the same order as the arguments of the function. Each type variable must have a new type declaration in a line by it's own. This feature is used by the program which extracts the function prototypes.

Those function heads are use to generate function prototypes which can be understood by ANSI-C compilers as well as by of K&R compilers. This is achieved by the old trick to introduce a macro which expands to nothing on the old compilers and to its aregument on ANSI compilers. This macro is defined appropriately according to the existence of the macro `__STDC__` which should indicate an ANSI compliant compiler.