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 2.41

Copyright ©1997 Gerd Neugebauer

BibTool is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 1, or (at your option) any later version.

BibTool is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this documentation; see the file COPYING. If not, write to the Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA.

Gerd Neugebauer
Mainzer Str. 8
56321 Rhens (Germany)

WWW: http://www.uni-koblenz.de/~gerd

Net: gerd@informatik.uni-koblenz.de
    gerd@imn.th-leipzig.de
    gerd@intellektik.informatik.th-darmstadt.de
## Contents

1 Introduction  
   1.1 The Module main.c ........................................... 5  

2 The BibTool C Library  
   2.1 The Header File database.h .................................. 7  
   2.2 The Module database.c ....................................... 8  
   2.3 The Header File entry.h ...................................... 11  
   2.4 The Module entry.c ........................................... 13  
   2.5 The Header File error.h ..................................... 14  
   2.6 The Module error.c ........................................... 18  
   2.7 The Header File expand.h .................................... 19  
   2.8 The Module expand.c .......................................... 20  
   2.9 The Header File key.h ........................................ 20  
   2.10 The Module key.c ............................................. 20  
   2.11 The Header File macros.h ................................... 24  
   2.12 The Module macros.c ......................................... 25  
   2.13 The Header File names.h .................................... 27  
   2.14 The Module names.c .......................................... 28  
   2.15 The Header File parse.h ..................................... 28  
   2.16 The Module parse.c ........................................... 29  
   2.17 The Header File print.h ..................................... 30  
   2.18 The Module print.c ........................................... 30  
   2.19 The Header File pxfile.h .................................... 32  
   2.20 The Module pxfile.c .......................................... 32
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.21</td>
<td>The Header File <code>record.h</code></td>
<td>32</td>
</tr>
<tr>
<td>2.22</td>
<td>The Module <code>record.c</code></td>
<td>36</td>
</tr>
<tr>
<td>2.23</td>
<td>The Header File <code>rewrite.h</code></td>
<td>37</td>
</tr>
<tr>
<td>2.24</td>
<td>The Module <code>rewrite.c</code></td>
<td>37</td>
</tr>
<tr>
<td>2.25</td>
<td>The Header File <code>resource.h</code></td>
<td>39</td>
</tr>
<tr>
<td>2.26</td>
<td>The Header File <code>rsc.h</code></td>
<td>40</td>
</tr>
<tr>
<td>2.27</td>
<td>The Module <code>rsc.c</code></td>
<td>40</td>
</tr>
<tr>
<td>2.28</td>
<td>The Header File <code>parse.h</code></td>
<td>41</td>
</tr>
<tr>
<td>2.29</td>
<td>The Module <code>parse.c</code></td>
<td>41</td>
</tr>
<tr>
<td>2.30</td>
<td>The Header File <code>stack.h</code></td>
<td>43</td>
</tr>
<tr>
<td>2.31</td>
<td>The Module <code>stack.c</code></td>
<td>43</td>
</tr>
<tr>
<td>2.32</td>
<td>The Header File <code>sbuffer.h</code></td>
<td>43</td>
</tr>
<tr>
<td>2.33</td>
<td>The Module <code>sbuffer.c</code></td>
<td>44</td>
</tr>
<tr>
<td>2.34</td>
<td>The Header File <code>symbols.h</code></td>
<td>45</td>
</tr>
<tr>
<td>2.35</td>
<td>The Module <code>symbols.c</code></td>
<td>47</td>
</tr>
<tr>
<td>2.36</td>
<td>The Header File <code>tex_aux.h</code></td>
<td>49</td>
</tr>
<tr>
<td>2.37</td>
<td>The Module <code>tex_aux.c</code></td>
<td>49</td>
</tr>
<tr>
<td>2.38</td>
<td>The Header File <code>tex_read.h</code></td>
<td>50</td>
</tr>
<tr>
<td>2.39</td>
<td>The Module <code>tex_read.c</code></td>
<td>50</td>
</tr>
<tr>
<td>2.40</td>
<td>The Header File <code>type.h</code></td>
<td>52</td>
</tr>
<tr>
<td>2.41</td>
<td>The Module <code>type.c</code></td>
<td>54</td>
</tr>
<tr>
<td>2.42</td>
<td>The Header File <code>version.h</code></td>
<td>54</td>
</tr>
<tr>
<td>2.43</td>
<td>The Module <code>version.c</code></td>
<td>54</td>
</tr>
<tr>
<td>2.44</td>
<td>The Header File <code>wordlist.h</code></td>
<td>55</td>
</tr>
<tr>
<td>2.45</td>
<td>The Module <code>wordlist.c</code></td>
<td>55</td>
</tr>
</tbody>
</table>

### 3 Coding Standards

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>K&amp;R-C vs. ANSI-C</td>
<td>57</td>
</tr>
</tbody>
</table>
Introduction

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

This documentation can not only be used to write new C programs dealing with Bib\TeX 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. Especially the section

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 library 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 Bib\TeX files are read and one or more Bib\TeX 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 behaviour 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.

```c
int main()
{
    char *argv[];
    Number of arguments
    int argc;
    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 envi-
    ronment.
}
```

```c
void save_input_file()
{
    char *file;
    File name to save.

    The input file pipe is a dynamic array of strings. This fifo stack is used to store the
    input BibTeX files to be processed by BibTool.

    This function is called to push an string into the pipe. If necesary the array has to be
    allocated or enlarged. This is done in larger junks to avoid lots of calls to realloc().

    Returns: nothing
}
```

```c
void save_macro_file()
{
    char *file;
    File name to save

    Simply feed the macro file name into the static variable. This function is useful since
    it can be called from rsc.c

    Returns: nothing
}
```

```c
void save_output_file()
{
    char * file;
    File name to save

    Simply feed the output file name into the static variable. This function is useful since
    it can be called from rsc.c

    Returns: nothing
}
```
2

The BibTool C Library

2.1 The Header File 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 as well.

DB

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 access 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

This is an invalid database. In fact it is NULL of the type DB.
2. The BibTool C Library

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.

2.2 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.

void db_add_record()

Function

DB db;
Record rec;

Database to insert the record into.
Record to add to the database.

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

Returns: nothing

Record db_find()

Function

DB db;
char *key;

Database to search in.

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 behaviour is nondeterministic.
2.2. The Module database.c

Returns: nothing

void db_forall()
    Database containing rec
    int (*fct)(Record);
    Boolean valued function determining the end of the processing.

Visit all normal records in the database 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_mac_sort()
    Database to sort.
    Sort the macros of a database. The sorting uses increasing lexicographic order accord-
ing 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

char * db_new_key()
    Database to search in.
    Key to find.
    Search the database for a record with a given old key and return the new one.

Returns: nothing

void db_rewind()
    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

void db_sort()
    Database to sort.
    int (*less)(Record,Record);
    Comparison function to use. This boolean function
takes two records and returns TRUE 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
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 as well. If all fails `NULL` is returned.

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

### delete_record()

Delete a record from a database. It is not checked, that the record really is part of the database.

Returns: nothing

### free_db()

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

Returns: nothing

### new_db()

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.

### print_db()

Print a database to a file in a way which is readable by BibTeX. The flags determine which parts should be printed. The symbolic names for the certain bits are defined in `database.h`. The are processed in the following order:

- `DB_PRINT_PREAMBLE`
- `DB_PRINT_STRING`
- `DB_PRINT_OTHER`
2.3 The Header File entry.h

DB_PRINT_COMMENT

The symbolic constant DB_PRINT_ALL turns on the printing for all types.

Returns: nothing

```c
int read_db()
{
    DB db;
    char *file;
    int (*fct)(DB,Record);
    int verbose;
}
```

Function

- `DB` Database to augment.
- `char *file;` File name to read from.
- `int (*fct)(DB,Record);` Function to determine whether to store a given record.
- `int 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: 1 if the file can not be opened. 0 otherwise.

2.3 The Header File 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.

`StringTab * entry_type` Variable

This is an array of `StringTab`s which represent entry types which 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.

`char * EntryName()` Macro

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 EntryCount()` Macro

Index of the entry.

This is the functional representation of the count 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 EntryUsed() Macro

   Entry

   Index of the entry.

   This is the functional representation of the use count 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 EndOfFile 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 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 STRING Macro

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

int PREAMBLE Macro

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

int COMMENT Macro

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

int ALIAS Macro

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

int MODIFY Macro

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

int INCLUDE Macro

   This symbolic constant representing a record type of a \texttt{BibTeX} inclusion instruction (\texttt{@Include}) which is proposed for \texttt{BibTeX} 1.0. This is a special record type which is provided automatically.
2.4 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 \texttt{@Comment}, \texttt{@Preamble}, \texttt{@String}, \texttt{@Include}, \texttt{@Modify}, and \texttt{@Alias}.

In addition to those special records the user can define additional record types which are denoted as “normal”. E.g. usually \texttt{@Article} and \texttt{@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.

\textbf{void def_entry_type()}

\begin{verbatim}
char * s;
\end{verbatim}

\textit{String containing the name of the entry.}

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

\textbf{void entry_statistics()}

\begin{verbatim}
int all;
\end{verbatim}

\textit{boolean. If all==0 only the used entry types are listed.}

Print a statistics on used entry types.

Returns: nothing

\textbf{IsSpecialRecord()}

\begin{verbatim}
Macro
Type
\end{verbatim}

\textit{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 \LaTeX{} is supposed to do so as well.

Returns: \texttt{TRUE} if the record type denoted a special record.

\textbf{IsNormalRecord()}

\begin{verbatim}
Macro
Type
\end{verbatim}

\textit{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: \texttt{TRUE} if the record type denoted a normal record.
int find_entry_type()
  char *s;

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

Returns: The index in the array or NOOP

char * get_entry_type()
  int idx;  

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()

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:

  STRING
  PREAMBLE
  COMMENT
  ALIAS
  MODIFY
  INCLUDE

Returns: nothing

2.5 The Header File 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 unnecessary arguments of the error() function providing appropriate values.

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

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

int ERR_WARNING
  Error type: Indicate that the error is in fact a warning which can be suppressed. The
  message is marked as warning. This flag is only in effect if the ERR_ERROR flag is not
  set.
int **ERROR\_POINT**
Macro
Error type: Indicate that the line and the error pointer should be displayed (if not
suppressed via other flags).

int **ERROR\_FILE**
Macro
Error type: Indicate that the file name and line number should be displayed (if not
suppressed via other flags).

int **ERROR\_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

```c
void WARNING() // Macro
  X             // Warning message.
Raise a warning. Print the argument as warning message and continue.
Returns: nothing
```

```c
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
```

```c
void WARNING3() // Macro
  X             // First warning message.
  Y             // Continuation of warning message.
  Z             // Second continuation of warning message.
Raise a warning. Print the three arguments as warning message and continue.
Returns: nothing
```

```c
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
```

```c
void ErrC()    // Macro
  CHAR         // Character to send to output.
Print a single character to the error stream.
Returns: nothing
```

```c
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 \texttt{ErrPrintF}() \quad \text{Macro}
\texttt{ErrPrintF} \quad \text{Format}.
\texttt{A} \quad \text{Argument.}

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

Returns: nothing

void \texttt{ErrPrintF2}() \quad \text{Macro}
\texttt{ErrPrintF2} \quad \text{Format}
\texttt{A} \quad \text{First argument.}
\texttt{B} \quad \text{Second argument.}

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

Returns: nothing

void \texttt{FlushErr} \quad \text{Macro}
Flush the error stream. This can be useful when single characters are written to an error stream which does buffering.

void \texttt{VerbosePrint1}() \quad \text{Macro}
\texttt{VerbosePrint1} \quad \text{Verbose message.}
\texttt{A} \quad \text{Print an informative message to the error stream.}

Returns: nothing

void \texttt{VerbosePrint2}() \quad \text{Macro}
\texttt{VerbosePrint2} \quad \text{Verbose message.}
\texttt{A} \quad \text{First argument.}
\texttt{B} \quad \text{Second argument.}

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

Returns: nothing

void \texttt{VerbosePrint3}() \quad \text{Macro}
\texttt{VerbosePrint3} \quad \text{Verbose message.}
\texttt{A} \quad \text{First argument.}
\texttt{B} \quad \text{Second argument.}
\texttt{C} \quad \text{Third argument.}

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

2.6 The Module error.c

To ensure a consistent appearance of error messages BibTool provides one generic error
reporting routine. This routine is controlled by several arguments to allow maximum flexi-
bility.

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 unnece-
sary details.
2.7 The Header File expand.h

This header file makes available the function defined in expand.c. This file includes the header file database.h.
2.8 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.

```c
char * expand_rhs() {
    char *s;
    char *pre;
    char *post;
    DB db;

    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.
}
```

2.9 The Header File `key.h`

This header file provides functions to deal with keys as they are defined in `keys.h`. This header file automatically includes the header files `database.h` and `sbuffer.h` since datatypes defined there are required.

2.10 The Module `key.c`

```c
void add_format() {
    char *s;

    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 as well.

    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.

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

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

Returns: nothing

```c
void add_ignored_word()
char *s;

Function
Add a new word to the list of ignored words for title key generation. The argument 
has to be saved by the caller!

Returns: nothing
```

```c
void add_sort_format()
char *s;

Function
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:

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

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

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

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

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

Returns: nothing
```

```c
void def_format_type()
char *s;

Function
```

Returns: nothing
char* fmt_expand()  
StringBuffer *sb;  
char * cp;  
DB db;  
Record rec;

Expands a format specification of the string buffer.

Returns: The first character after the

void free_key_node()  
KeyNode kn;  

Here KeyNodes should be freed. Well, in a future release ...

Returns: nothing

char *get_field()  
DB db;  
Record rec;  
char * name;

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.

void make_key()  
Record rec;  
DB db;  

Generate a key for a given record.

Returns: nothing

void make_sort_key()  
Record rec;  
DB db;  

Returns: nothing

int mark_key()  
Record rec;  
DB db;  

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

Returns: nothing
void set_base()
  char *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 uppercase letters.
- If the value is **lower** or starts with a lower case letter then the disambiguation is done with lowercase 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

int set_field()
  DB db;
  Record rec;  // Record to receive the value.
  char * name;  // Field name to add.
  char * value;  // String representation of 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: 0 if the assignment has succeeded.

void set_separator()
  int n;  // Array index to modify.
  char *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

2.11 The Header File 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

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

MacroMacroNULL

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

char * MacroName()

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.
char * MacroValue()  
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 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 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.

2.12 The Module macros.c

void def_field_type()  
String containing an equation.
This function adds a printing representation for a field name to the used list. The argument is an equation of the following form

\[ \text{type} = \text{value} \]

\text{type} is translated to lower case and compared against the internal representation. \text{value} is printed at the appropriate places instead.

Returns: nothing

int def_macro()  
name of the macro.
NULL or the value of the new macro
initial count for the macro.
Define or undefine a macro.

Returns: nothing

void dump_mac()  
File name of the target file.
if \( = 0 \) only the used macros are written.
Write macros to a file.

Returns: nothing
void foreach_macro()
    int (*fct)(char *, char *);
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.
Returns: nothing

void free_macro()
    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

char * get_item()
    char * name;
    int type;
Returns:

char * get_key_name()
    char *s;
Returns:

void init_macros()
Initialize some macros from a table
Returns: nothing

char * look_macro()
    char *name;
    int add;
Return the value of a macro. If the macro is undefined its name is returned.
Returns: The value or NULL

Macro new_macro()
    char *name;
    char *val;
    int count;
    Macro next;
Allocate a new macro structure and fill it with initial values. Upon failure exit() is called.
2.13. The Header File names.h

Returns: The new Macro

void save_key()
    char * s;
    char * key;

    Returns: nothing

2.13 The Header File names.h

SNameNode

typedef struct nameNODE {
    int nn_type;
    int nn_strip;
    char * nn_pre;
    char * nn_mid;
    char * nn_post;
    struct nameNODE * nn_next;
} SNameNode, * NameNode;

NameNULL

NameType()
    NN

    Returns:

NameStrip()
    NN

    Returns:

NamePre()
    NN

    Returns:

NameMid()
    NN

    Returns:
Returns:

NamePost()  
NN

Returns:

NextName()  
NN

Returns:

2.14 The Module names.c

NameNode name_format()  
char *s;

Returns:

char * pp_list_of_names()  
char ** wa;  
NameNode format;  
char * trans;  
int max;  
char * comma;  
char * and;  
char * namesep;  
char * etal;

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

void set_name_format()  
NameNode *nodep;  
char * s;

Returns: nothing

2.15 The Header File 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.
2.16 The Module parse.c

void init_read()
Function
Initialize the reading apparatus. Primarily try to figure out the file search path.

Returns: nothing

void normalize_symbol()
Function
char * s;
Function to translate a symbol into a normal form. This will translate the symbol to
lower case.

Returns: nothing

int parse()
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.
EndOfFile is returned if nothing could be read and the end of the file has been
encountered.
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, EndOfFile, or NOOP.

int read_rsc()
Function
char * 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:

int see_bib()
Function
char * 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 \texttt{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 \texttt{seen()}. This function is for internal purposes mainly. See \texttt{read\_db()} for a higher level function to read a database.

Returns: \texttt{TRUE} iff the file could be opened for reading.

\begin{verbatim}
int see_rsc()
char * fname;
Open a rsc file to read from.

Returns:
\end{verbatim}

\begin{verbatim}
int seen()
Function
Close input file for the BibTeX reading apparatus. After this function has been called \texttt{parse()} might not return sensible results.

This function is for internal purposes mainly. See \texttt{read\_db()} for a higher level function to read a database.

Returns: \texttt{FALSE} if an attempt was made to close an already closed file.
\end{verbatim}

\begin{verbatim}
void set_rsc_path()
char * val;
Initialize the resource file reading apparatus. Primarily try to figure out the file search path.

Returns: nothing
\end{verbatim}

\section{2.17 The Header File print.h}

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

This header file automatically includes \texttt{record.h} and \texttt{database.h}.

\section{2.18 The Module print.c}

This module provides also access to the functions and variables defined in \texttt{entry.c}. Consult also the documentation of this file for details.
void print_record()

FILE * file; Stream to print onto.
DB db; Database containing the record.
Record rec; Record to print.
char * 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_colp This integer variable controls the indentation of preamble records.
rsc_cols 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()

char * 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 rsc.c.

Returns: nothing
2.19 The Header File \texttt{pxfile.h}

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

This header file automatically includes \texttt{bibtool.h} and \texttt{<stdio.h>}.

2.20 The Module \texttt{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.

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

\begin{verbatim}
FILE \* px_fopen()
char \* name;
(char) name of the file to open.

char \* mode;
Mode for opening the file like used with \texttt{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.
\end{verbatim}

\begin{verbatim}
char **px_s2p()
char \* s;

int sep;
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
\end{verbatim}

2.21 The Header File \texttt{record.h}

This module contains functions which deal with records in databases.
2.21. The Header File record.h

Record

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_token. The different types are defined in typedef struct rECORD {

  char * rc_key; The sort key.
  char * rc_old_key; The old sort key.
  int rc_token; The type of the record.
  char * rc_source; The source of the record.
  int rc_free; The size of the heap.
  char ** rc_heap; The heap.
  struct rECORD * rc_next; Pointer to the next record.
  struct rECORD * rc_prev; Pointer to the previous record.
  char * rc_comment; The comment following the given record.
}

} SRecord, *Record;

Record NULL

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

int RecordTokenMask

Macro

Bit mask to extract the pure token from a record token. This is usually not used directly but implicitly with other macros from this header file.

int RecordNotTokenMask

Macro

Bit mask to extract the non-token bits from a record token. This is usually not used directly but implicitly with other macros from this header file.

int RecordTokenXREF

Macro

Bit mask for the XREF flag of a record. This is usually not used directly but implicitly with other macros from this header file.

int RecordTokenDELETED

Macro

Bit mask for the DELETED flag of a record. This is usually not used directly but implicitly with other macros from this header file.
int SetRecordXREF()

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 token.

int IsRecordXREF()

R

Record to consider.
Check whether the XREF flag of a record is set.

Returns: FALSE if the XREF flag is not set.

int SetRecordDELETED()

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 token.

int IsRecordDELETED()

R

Record to consider.
Check whether the record is marked as deleted.

Returns: FALSE if the DELETED flag is not set.

int RecordToken()

R

Record to consider.
Get the pure token without the special bits of a record.

Returns: The pure token.

int Record_Full_Token()

R

Record to consider
Functional representation of the full record token. This can be used to access the
token component of a record. It can also be used as lvalue.
This macro should be used with care. It is preferable to use the other macros to
modify the normal part and the special bits separately.

Returns: The full token of a record.
SetRecordType()

R  
T  

Record to consider.
New token type.

Set the token type of a record. Care is taken not to influence the special bits which will be left unchanged.

The type can have a value of at most `RecordTokenMask`. Any bits exceeding this value are ignored.

Returns: The new token of the record.

char * RecordOldKey()

R  

Record to consider.

char * RecordKey()

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.

char ** RecordHeap()

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()

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()

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.

char * RecordComment()

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.
char * **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:

### 2.22 The Module record.c

**void add_sort_order()**

```c
char *val;
```

string resource of the order.

Insert the sort order into the order list.

Returns: nothing

**Record copy_record()**

```c
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.

**void free_1_record()**

```c
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()**

```c
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()**

```c
int token;
int size;
```

The token type of the record. 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
char * s;	Initial string to fill in the WordList structure
Allocate a WordList and fill its slots.
Returns:

void push_to_record()
Function
Record rec;	Record to free.
char * s;	Left hand side of the equation.
char * t;	Right hand side of the equation.
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 neces-
sary to make private copies and it is possible to avoid expensive string comparisons.
Returns: nothing

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

2.23 The Header File rewrite.h

2.24 The Module rewrite.c

void add_check_rule()
Function
char *s;	ruire save.
Save a check rule for later use. The main task is performed by add_rule.
Returns: nothing

```c
void add_extract()
char *s;                  /* rule to save. */
Save an extraction rule for later use. The main task is performed by add_rule.
Returns: nothing
```

```c
void add_field()
char *spec;               /* A string of the form token=value */
Save a token and value for addition.
Returns: nothing
```

```c
void add_rewrite_rule()
char *s;                  /* rule to save. */
Save a rewrite rule for later use. The main task is performed by add_rule.
Returns: nothing
```

```c
void add_rule()
char *s;
Rule *rp;
Rule *rp_end;
int casep;
Generic addition of a rule to a list of rules.
Returns: nothing
```

```c
void clear_addlist()
Reset the addlist to the empty list.
Returns: nothing
```

```c
void free_rule()
Rule rule;                /* First rule in the list. */
Free a list of rules.
Returns: nothing
```

```c
int is_selected()
DB db;                    /* The database record is belonging to. */
Record rec;               /* Record to look at. */
Boolean function to decide whether a record should be considered. This function
selects all records in no regular expression support has been enabled.
Returns: TRUE iff the record is selected be a regexp or none is given.

**Rule new_rule()**

```c
char *field;
char *pattern;
char *frame;
int casep;
```

Allocate a new Rule and fill some slots

Returns: A pointer to the allocated structure or NULL upon failure.

**void remove_field()**

```c
char *field;
Record rec;
```

This is a symbol containing the name of the field to remove.

Record in which the field should be removed.

Remove the given field from record.

Returns: nothing

**void rewrite_record()**

```c
DB db;
Record rec;
```

The database record is belonging to.

Actual record to apply things to.

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

Returns: nothing

**void save_regex()**

```c
char *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

### 2.25 The Header File 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 make live easy. It should also be fun to find 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.

### 2.26 The Header File `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.

### 2.27 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.

```c
int load_rsc()
    char *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 if the reading failed.

void rsc_print()
    char *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

int search_rsc()
    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 config.h)

Returns: TRUE iff the resource loading succeeds somewhere.

```c
int set_rsc()
char * name;        // Name of the resource to set.
char * 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.

```c
int use_rsc()
char *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: FALSE iff no error has occurred.

## 2.28 The Header File s_parse.h

## 2.29 The Module s_parse.c

```c
char * s_parse()
```

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:

- **StringParseException** The string is analyzed and the proper type is determined au-
domatically. 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 a 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 a 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.

**StringParseSkip** The string is analyzed and the first position not containing whitespace, =, or # is returned. In this case the returned value is not translated into a symbol.

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

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 sp.h should be used instead.

Returns: A symbol containing the requested entity or NULL.

```c
int sp_open()
```

**Function**

```
char * 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: TRUE
2.30 The Header File stack.h

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

2.31 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 problematic since this memory is not assumed to be really large. Normally just a few strings are pushed to the stack at any time.

```c
char * pop_string()

Pop a string from the stack. If 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.
```

```c
void push_string()

char * 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
```

2.32 The Header File 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.

```c
sbputchar()

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

2.33 The Module sbuffer.c

This module contains functions for dealing with strings of arbitrary 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.

int sbclose()
Function
    StringBuffer* sb;  
    Pointer to string buffer which should be closed

Free an old string buffer.

Returns: Return 0 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

int 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 signalled via the return value.

Returns: FALSE if no memory is left.
2.34 The Header File 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.
char * symbol()

Macro

STR

String to translate into a symbol.

Translate a string into a symbol. The symbol returned is either created or an existing symbol is returned.

Returns: The symbol corresponding to the argument.

void ReleaseSymbol()

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

StringTab

This is the pointer type representing an entry in the symbol table. It contains a string and some integers.

typedef struct STAB {
  char * st_name;            // The string representation of the symbol
  int st_count;
  int st_flags;              // Bits of certain flags.
  int st_used;               // Counter for determining the number of uses
  struct STAB *st_next;      // Pointer to the next item.
} *StringTab;

StringTab NextSymbol()

Macro

ST

Current StringTab

The next StringTab of the argument. This macro can also be used as lvalue.

Returns: The next StringTab or NULL.

int SymbolCount()

Macro

ST

Current StringTab

The count slot of a StringTab. This macro can also be used as lvalue.

Returns: The count slot of ST.

int SymbolUsed()

Macro

ST

Current StringTab

The used slot of a StringTab. This macro can also be used as lvalue.
Returns: The used slot of \texttt{ST}.

\begin{verbatim}
char * SymbolName()
ST Current StringTab
\end{verbatim}

The name slot of a \texttt{StringTab}, i.e. the string representation. This macro can also be
used as lvalue.

Returns: The name slot of \texttt{ST}.

\begin{verbatim}
int SymbolFlags()
ST Current StringTab
\end{verbatim}

The flags slot of a \texttt{StringTab}. This macro can also be used as lvalue.

Returns: The flags slot of \texttt{ST}.

\begin{verbatim}
char * sym_empty
\end{verbatim}

The empty symbol. This is a symbol pointing immediately to a \texttt{\null} byte. This needs
\texttt{init_symbols()} to be called first.

\begin{verbatim}
char * sym_crossref
\end{verbatim}

The symbol \texttt{crossref}. This variable needs \texttt{init_symbols()} to be called first.

\section*{2.35 The Module \texttt{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.

\begin{verbatim}
void init_symbols()
\end{verbatim}

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 assumtion. 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.

Returns: nothing

\begin{verbatim}
char * new_string()
char * s; String to duplicate
\end{verbatim}

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.

**StringTab new_string_tab()**

| char *name; | String value of the StringTab node. |
| int count; | Initial use count of the StringTab node. |
| int flags; | Flags of the new StringTab node. |

Allocate a new StringTab structure and fill it with initial values.

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

Returns: Pointer to a new instance of a StringTab.

**char * sym_add()**

| char *s; | String which should be translated into a symbol. |
| int count; | The use count which should be added to the 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.

**void sym_dump()**

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

**int sym_flag()**

| char *s; | Symbol |

Get the flags of the symbol given as argument.

Returns: The flags of the recently touched StringTab.
void sym_gc()  
Function
This is the garbage collector. It analyzes the symbol table and releases all SymbolTab nodes not needed any more.

Right now it is purely experimental. Better let your hands off.

Returns: nothing

void sym_set_flag()  
Function
char *s;  Symbol to augment.
int flags;  New flags to add.

Add the flags to the symbol corresponding to the argument s by oring them together with the given value.

Returns: nothing

void sym_unlink()  
Function
char *s;  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 that 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

2.36 The Header File tex_aux.h

2.37 The Module tex_aux.c

int aux_used()  
Function
char *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:

void clear_aux()  
Function
Reset the aux table to the initial state.

Returns: nothing
2. The BibTool C Library

**int foreach_aux()**

Function

```c
int (fct)(char*);
```

Funtion to apply

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

Returns: cite_star

**int read_aux()**

Function

```c
char * fname;
void (*fct)(char*);
int verbose;
```

Funtion to apply

Analyse an aux file.

Returns: nothing

### 2.38 The Header File 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.

### 2.39 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

```c
int c;
int arity;
char * s;
```

Character to make active.

Arity of the macro assigned to the active character.

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

```c
char *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()
   char *name;
   int arity;
   char *body;
Add a new \TeX macro definition.

Returns: nothing

void \TeX_open_file()
   FILE * file;
   File pointer of the file to read from.
Prepare things to parse from a file.

Returns: nothing

void \TeX_open_string()
   char * s;
   String to read from.
Prepare things to parse from a string.

Returns: nothing

int \TeX_read()
   char * cp;
   Pointer to position where the character is stored.
   char **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()
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
2.40 The Header File 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 xor array like \TeX has one (some day...)

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

This header file also provides the functions and variables 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.

**int is_allowed()**

**Macro**

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.

**int is_upper()**

**Macro**

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.

**int is_lower()**

**Macro**

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.

```c
int is_alpha()
{
    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.
}
```

```c
int is_digit()
{
    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.
}
```

```c
int is_space()
{
    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.
}
```

```c
int is_extended()
{
    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.
}
```

```c
int is_wordsep()
{
    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.
}
```

```c
charToLower()
{
    Character to translate
    Translate a character to its 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.
}
```
2. The BibTool C Library

2.41 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.

```c
int case_cmp()

char * s; /* First string to consider. */
char * 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: FALSE iff the strings differ.
```

```c
void init_type()

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.

Returns: nothing
```

2.42 The Header File version.h

2.43 The Module version.c

```c
void show_version()

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

Returns: nothing
```
2.44 The Header File wordlist.h

WordNULL

ThisWord()
X

Returns:

NextWord()
X

Returns:

2.45 The Module wordlist.c

This module contains functions which deal with lists of words.

int find_word()
    char * s;  \textit{String to find.}
    WordList wl;  \textit{Word list to search in.}

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

    Returns: FALSE if the word does not occur in the wordlist.

int foreach_word()
    WordList wl;  \textit{WordList to traverse.}
    int (* fct)(char*);  \textit{Function to apply.}

    Applies the given function to all elements in the wordlist as long as the function
does not return 0.

    Returns: return value of last function or 1.

void free_words()
    WordList *wlp;
    void (* fct)(char*);  \textit{Function to apply}

    Release the memory allocated for a list of words.

    Returns: nothing
void list_words()
    WordList wl;  // Word list to display
    List all words in the word list on the error stream (see error.c and error.h. Each word is presented on a line by its own without additional characters.

    Returns: nothing

void save_word()
    char * s;  // String to add to the wordlist
    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.
    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
3

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 developed independently but should be used for consistency. In the following sections these coding standards are described.

3.1 K&R-C vs. ANSI-C

BibTool tries hard to be portable to a 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.

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 od trick to introduce a macro which expands to nothing on the old compilers and to its argument on ANSI compilers. This macro is defined appropriately according to the existence of the macro __STDC__ which should indicate an ANSI compliant compiler.
Index

add_check_rule() .................................. 37
add_db_record() .................................. 8
add_extract() ................................... 38
add_field() ..................................... 38
add_format() ..................................... 20
add_ignored_word() .............................. 21
add_rewrite_rule() .............................. 38
add_rule() ...................................... 38
add_sort_format() ............................... 21
add_sort_order() ................................ 36
ALIAS ............................................ 12
aux_used() .................................... 49
case_cmp() ...................................... 54
clear_addlist() .................................. 38
clear_aux() ...................................... 49
COMMENT ........................................ 12
copy_record() ................................... 36
db_find() ......................................... 8
dbforall() ....................................... 9
db_mac_sort() ................................... 9
db_new_key() ..................................... 9
db_rewind() ...................................... 9
db_sort() ......................................... 9
db_string() ...................................... 10
DBcomment() ..................................... 8
DBnormal() ....................................... 8
DBpreamble() ..................................... 8
DBstring() ........................................ 8
DebugPrint1() ................................... 18
DebugPrint2() ................................... 18
DebugPrint3() ................................... 18
def_entry_type() ................................ 13
def_field_type() ................................ 25
def_format_type() ................................ 21
def_macro() ..................................... 25
delete_record() .................................. 10
dump_mac() ....................................... 25
EndOfFile ......................................... 12
entry_statistics() ................................ 13
EntryCount() ..................................... 11
EntryName() ...................................... 11
tenrytype ......................................... 11
EntryUsed() ...................................... 12
Err() ............................................. 16
ErrC() ........................................... 16
ERRERROR ....................................... 14
ERREXIT ......................................... 15
ERRFILE .......................................... 15
ERROR() ........................................... 15
ero() ............................................... 19
ERROR2() ......................................... 15
ERROR3() ......................................... 15
ERROREXIT() ..................................... 15
ERRPOINT ......................................... 15
EnPrint() ......................................... 16
EnPrintF() ........................................... 17
EnPrintF2() ....................................... 17
ERRORWARNING .................................. 14
expand_lhs() ..................................... 20
find_entry_type() ................................ 14
find_word() ....................................... 55
FlushErr .......................................... 17
fmt_expand() ..................................... 22
foreach_aux() .................................... 50
foreach_macro() .................................. 26
foreach_word() ................................... 55
free1_record() ................................... 36
free_db() ......................................... 10
free_key_node() ................................... 22
free_macro() ..................................... 26
free_record() .................................... 36
free_rule() ....................................... 38
free_words() ..................................... 55
g_entry_type() ................................... 14
g_field() .......................................... 22
g_item() .......................................... 26
g_key_name() ..................................... 26
INCLUDE .......................................... 12
init_entries() .................................... 14
init_macros() ..................................... 26
init_read() ........................................ 29
init_symbols() ................................... 47
58