ZMAKE

Another way to handle compilation on various systems

User's manual

F. Chabaud
fchabaud@free.fr
R. Lercier
lercier@celar.fr

DCSSI
18 rue du Dr. Zamenhoff
92131 Issy-les-Moulinaux
France

Centre d'Électronique de l'Armement
CASSI/SCY/EC
35998 Rennes Armées
France

December 2, 2004

Contents

Chapter 1

Introduction

In [3], it is stated that

> "In the UNIX world, software development typically involves make, using a Makefile describing how to build and install the programs in which we're interested. In essence, make functions as a command-generating engine, and the Makefile controls which command to generate for specific targets. This is a tremendous convenience for the programmer, since it eliminates the need to type out a bunch of commands each time you want to build something.
>
> Unfortunately, Makefiles are not portable. The commands that make so conveniently generates at the drop of a hat are subject to variation from system to system, so a Makefile that works on my system may not work on yours. Ideally, we'd be able to take a software project, put it on another machine, compile and install it, and it would run. But it doesn't always work that way."

This point of view is developed in details in section 2.

Two alternatives are known to avoid such troubles, the first one was developed for X-WINDOWS, it is called imake. The second one was developed by the GNU community, it is called configure.

Imake : With this solution, you don't write a Makefile, instead you write an imakefile, a machine independent description of the targets you want to build. Static machine dependencies are centralized into a set of configuration files isolated in their own directory, config, so they don't appear in the imakefile. The program imake merges information obtained from config and imakefile to generates a properly configured Makefile. Then you run make to build your programs.
From a practical point of few, imake is just an interface for the preprocessor cpp. In a first approximation, the Makefile that imake generates, would be more or less the file obtained with cpp -I. -Iconfig config/Imake.tmpl Makefile. cpp usually defines specific variables for each machine and these variables are tested in Imake.templ to include the corresponding files of config. Then Imake.templ includes the current imakefile.

Configure : This solution generates Makefiles from a machine independent description of the targets you want to build and from machine dependencies dynamically collected at the compilation stage.
From a practical point of few, configure is just a Bourne Script Shell which performs some tests in order to determine the way how the software must be compiled.

In ZMAKE, we tried to get the best of these two approaches. In fact, the strong assumptions we made in our design is that the only thing we can assume about the target system is that it has a C-compiler and that it supports POSIX system calls. Most of the OS we heard about verify this ! So, external tools like imake, sh, make, ... are no more needed.

The way how a software must be compiled is no more specified in a a Makefile, an Imakefile or a configure script shells. It is simply specified in files written in C. These files are typically called zmake.c. Thus, the challenge that we had to face while developing ZMAKE was to give developers APIs which can be easily used. Like this, we expect to have zmake.c files easy to understand. That's why these APIS are very similar to Makefile targets. These APIS are described in section 3.

In order to have a documentation as close as possible to the C implementation, we used to interlace documentation and C code. This is done thanks to a specific parser called zparse, This parser is described in section 4.

Finally, a small example, the C file zmake.c which compiles this documentation is given in section 5.

Remark : Except the original way we process the LaTeX [4] documentation, this work about portable compilation and version control of sources with CVS [1] inherit in parts from the programming environment work used by the LiPS project [2].

Chapter 2

Makefiles are not portable

All this can be found in [3].

Suppose we're writing a small C program, myname consisting of a single source file myname.c:

```
#include <stdio.h>

int main()
{
  char *getenv();
  char *p;

  p = getenv ("USER");
  if (p == (char *)NULL) {
    printf("Cannot tell your user name\n");
  }
  else {
    printf("Your user name is %s\n", p);
  }
  exit(0);
}
```

The Makefile for the program might look like this:

```
myname: myname.o
        cc -o myname myname.o
install: myname
        install myname /usr/local
```

To build and install myname, we'd use these commands:

```
% make myname
cc  -c myname.c
cc -o myname myname.o
% make install
install myname /usr/local
```

All our friends are astonished at the use fullness of this program and immediately begin asking us for the source code. We give it to one of them, together with the Makefile, and he lopes off to install it on his machine. The next day he sends back the following report:

- We don't have any install program, so I used cp.
- Our local installation directory is /usr/local/bin rather than /usr/local.

He sends back a revised Makefile, having made the necessary changes, and also having conveniently parameterized the things he found to be different on his system, using the make variables INSTALL and BINDIR at the beginning of the Makefile:

```
INSTALL = cp
BINDIR = /usr/local/bin
myname: myname.o
        cc -o myname myname.o
install: myname
        install myname /usr/local
```

This is a step forward. The installation program and directory as specified in this Makefile are no longer correct for our machine, but now they're parameterized and thus easily located and modified. We change the values back to install and /usr/local/bin/ so they'll work on our machine, and both we and our friend are happy.

Alas, our idyllic state of mind doesn't last long. We give the program to another friend. She builds it and reports back her changes:

- I prefer to use gcc, not cc.
- Our C library is broken and doesn't contain getenv(), I have to link in -lc_aux as well.
- The environment variable USER isn't used on our system, but LOGNAME is, so I used that instead.

She, too, sends back a revised Makefile, further parameterized:

```
CC = gcc
EXTRA_LIBS = -lc_aux
CFLAGS = -DUSE_LOGNAME
INSTALL = cp
BINDIR = /usr/local/bin
myname: myname.o
        $(CC) $(CFALGS) $(EXTRA_LIBS) -o myname myname.o
install: myname
        install myname /usr/local
```

and a revised file myname.c:

```
#include <stdio.h>

int main()
{
  char *getenv();
  char *p;

#ifdef USE_LOGNAME
  p = getenv ("LOGNAME");
#else
  p = getenv ("USER");
#endif
  if (p == (char *)NULL) {
    printf("Cannot tell your user name\n");
  }
  else {
    printf("Your user name is %s\n", p);
  }
  exit(0);
}
```

Now, with a simple change to the Makefile, myname can be recompiled on machines that use either environment variable. This is encouraging. Using our vast and ever increasing porting experience, myname now works on more machines-three ! And the revised Makefile is better than the original:

- It parameterizes all the non portability we've encountered so far.
- Nonportabilities are identified explicitly at the beginning of the Makefile.
- They're easily modified by editing Makefile.

However, we begin to notice uneasily that the number of things that might need changing from machine to machine is increasing. They were only two differences between machines when we built myname on two systems. Now we

can build it on three systems but the number of differences has increased to five. How many differences will we find when we attempt to port myname to additional systems ? As the Makefile editing job gets bigger, it becomes more difficult. Each nonportability adds another increment to the burden incurred each time myname is built on a different machine.

Also, we'd better keep a list of parameters appropriate for each type of machine on which the program is known to run, or we'll forget them. And we need to distribute the list to other people who want to build the program, so they can consult it to see how they might need to modify the makefile on their systems.

The previous example outlines mainly four drawbacks of make:

- There are no conditionals:
  If we could assign values to make variables conditionally, we could parameterize machine dependencies by assigning variable values appropriate to a given machine type. To some extend, we can use the shell's if-statement syntax to get around lack of conditionals in target entries, but it's clunky to do so.

- There is no flow control:
  Loops and iterators must be done using shell commands to simulate them.

- It's difficult to make global changes:
  For example, suppose you use BINDIR to name your installation directory. You can assign it the proper value in each Makefile in your project. But if you decide to change it, you must find and change every Makefile. Alternatively, you can assign BINDIR, once in the top level Makefile and pass the value to make commands in subdirectories using suitable recursive rules. But, then you can't run make directly in a subdirectory because BINDIR won't have the proper value. Recursive rules are difficult to write correctly, anyway.

- Header file dependencies for C source files are inherently nonportable:
  Different systems organize header files differently. You might even have multiply sets on systems that support development under more than one environment (BSD, System V, ...). This makes it impossible to list header file dependencies statically in the Makefile. They must be computed on the target machine at build time.

Chapter 3

Writing zmake.c files

Basically, a zmake command is the compilation of a zmake.c C file which contains a single main() procedure calling few zmake API. These API are described here.

3.1 Initialization

Procedure 1 *Initialization of a zmake.*

```
void zmake_init(DEFINES, INCLUDES, LOCAL_LDFLAGS, LOCAL_LIBRARY,
   argc, argv)
   char *DEFINES;
   char *INCLUDES;
   char *LOCAL_LDFLAGS;
   char *LOCAL_LIBRARY;
   int argc;
   char *argv[];
```

Input: *Standard defines* DEFINES, *standard includes* INCLUDES, *standard link flags* LOCAL_LDFLAGS *and standard libraries* LOCAL_LIBRARY *to use for any compilation and the well known* argc *and* argv *C arguments.*

Side effect: *If a statement* verbose=*100 is set on the command line, debugging output is done while procesing* ZMAKE *functions. Moreover, this function checks that any needed file for the compilation of* zmake *is older than* zmake, *otherwise recompilation of* zmake *is performed.*

Procedure 2 *Ending a zmake.*

```
void zmake_close()
```

Side effect: *An* exit(0) *is performed.*

3.2 Handling targets

Procedure 3 *Is a target specified on the command line.*

```
int Target(target, argc, argv)
    char *target;
    int argc;
    char *argv[];
```

Input: *A target* target *and the well known C arguments* argc *and* argv.
Output: *1 if target was specified on the command line, 0 otherwise.*

Procedure 4 *Target implications.*

```
void TargetImply(target, subtargets)
    int target;
    int subtargets;
```

Input: *A target* target *and subtargets* subtargers.
Side effect: *Subtargets are set to one if target is non* NULL.

3.3 Handling paths

Procedure 5 *Concatening 2 strings.*

```
char *concat(s1, s2)
  char *s1; char *s2;
```

Input: *Two strings* s1 *and* s2 *to concatenate.*
Output: *An allocated strings which contains the concatenation of* s1 *and* s2.

Procedure 6 *Appending a string to another.*

```
void append(s1, s2)
 char **s1; char *s2;
```

Input: *A string* s2 *to append to* s1.
Side effect: *The string* s1 *contains the concatenation of* s1 *and* s2.

3.4 Processing files

Procedure 7 *Recursively compiling and calling* zmake *command in directories.*

```
void NamedTargetSubdirs(target,dirs,subtarget,argc, argv)
   int target;
   char *dirs, *subtarget;
   int argc; char *argv[];
```

Input: *A target* target, *the list of directories* dirs *containing* zmake.c *files, targets* subtargets *which will be arguments for* zmake *called in these directories and the well known* argc *and* argv *C arguments.*
Side effect: *If target is non zero, in each directory* dirs, zmake *is compiled if this was not already done. The resulting* zmake *are called with target* subtarget.
Note: *If the calling* zmake *was just (re)compiled, the binaries* zmake *in the directories are automatically (re)compiled.*

Procedure 8 *Calling* lex *on source files to obtain C files.*

```
int LexToSrcs(target, odir, cprefix, ydir, yprefix, yfiles,
   includes, flags)
   int target;
   char *odir; char *cprefix;
   char *ydir; char *yprefix; char *yfiles;
   char *includes; char *flags;
```

Input: *A target* target, *the directory* odir *which will receive the obtained C files, a prefix* cprefix *which will be added to these C files, the directory* ydir *which contains the* lex *files, the prefix* yprefix *of these* lex *files, the names* yfiles *of these* lex *files (without the usual suffix* .l*), directories* includes *which must be included and specific flags* flags *to add to* lex.
Output: *The integer 1 if an error occured, 0 otherwise.*
Side effect: *C files are created in* odir *if* target *is non zero.*

Procedure 9 *Calling* yacc *on source files to obtain C files.*

```
int YaccToSrcs(target, odir, cprefix, ydir, yprefix, yfiles,
   includes, flags)
   int target;
   char *odir; char *cprefix;
   char *ydir; char *yprefix; char *yfiles;
   char *includes; char *flags;
```

Input: *A target* target, *the directory* odir *which will receive the obtained C files, a prefix* cprefix *which will be added to these C files, the directory* ydir *which contains the* yacc *files, the prefix* yprefix *of these* yacc *files, the names* yfiles *of these* yacc *files (without the usual suffix* .y*), directories* includes *which must be included and specific flags* flags *to add to* yacc.
Output: *The integer 1 if an error occured, 0 otherwise.*
Side effect: *C files are created in* odir *if* target *is non zero.*

Procedure 10 *Compiling C files to obtain object files.*

```
int SrcToObjects(target, odir, oprefix, cdir, cprefix, cfiles,
   includes, cflags)
   int target;
   char *odir; char *oprefix;
   char *cdir; char *cprefix; char *cfiles;
   char *includes; char *cflags;
```

Input: *A target* target, *the directory* odir *which will receive the obtained object files, a prefix* oprefix *which will be added to these object files, the directory* cdir *which contains the C files, the prefix* cprefix *of these C files, the names* cfiles *of these C files (without the usual suffix* .c*), directories* includes *which must be included and specific flags* cflags *to add to* cc.
Output: *The integer 1 if an error occured, 0 otherwise.*
Side effect: *Object files are created in* cdirs *if* target *is non zero.*

Procedure 11 *Archiving object files in a list file.*

```
int ObjectsToArList(target, libdir, lprefix, lib, objtoar,
   odir, rprefix, ofiles)
   int target;
   char *libdir; char *lprefix; char *lib; char *objtoar;
   char *odir; char *rprefix; char *ofiles;
```

Input: *A target* target, *the directory* libdir *which will receive the list file, a prefix* lprefix *which will be added to list file, the name* lib *of the obtained list file, the suffix* objtoar *of this list, the directory* odir *which contains the object files, the prefix* rprefix *of these object files and the names* ofiles *of these object files (without the usual suffix* .o*).*
Output: *The integer 1 if an error occured, 0 otherwise.*
Side effect: *The list file is created in* libdir *if* target *is non zero.*

Procedure 12 *Creating an executable program from object files.*

```
int ObjectsToBinary(target, bdir, bprefix, bfile, odir, oprefix, ofiles,
   ldflags, ldlibs)
   int target;
   char *bdir; char *bprefix; char *bfile;
   char *odir; char *oprefix; char *ofiles;
   char *ldflags; char *ldlibs;
```

Input: *A target* target, *the directory* bdir *which will receive the executable program* bfile, *a prefix* bprefix *which will be added to this binary, the name* bfile *of this binary, the directory* odir *which contains the object files, the names* ofiles *of these object files (without the usual suffix* .o*), the flags needed for* cc *and the library to link with.*
Output: *The integer 1 if an error occured, 0 otherwise.*
Side effect: *The binary is created in* bdir *if* target *is non zero.*

Procedure 13 *Adding object files to a library.*

```
int ArListToLibrary(target, ldir, lprefix, lname, lsuf,
  mdir, mprefix, mname, msuf, ddir, dprefix, dname, dsuf)
  int target;
  char *ldir; char *lprefix; char *lname; char *lsuf;
  char *mdir; char *mprefix; char *mname; char *msuf;
  char *ddir; char *dprefix; char *dname; char *dsuf;
```

Input: *A target* target, *the directory* ldir *which contains the library, a prefix* lprefix, *the name* lname *and the suffix* lsuf *of the library, the directory* mdir, *prefix* mprefix, *name* mname *and suffix* msuf *of a temporary file and the directory* ddir, *prefix* dprefix, *name* dname *and suffix* dsuf *of a list file obtained with* ObjectsToArList.

Output: *The integer 1 if an error occured, 0 otherwise.*

Side effect: *The tex files are compiled if* target *is non zero.*

Procedure 14 *Parsing C files to obtain* LaTeX *documentation.*

```
int SrcToTex(target, tdir, tprefix, cdir, cprefix, cfiles, csuf,
   fdir, fprefix, pdir, filter)
   int target;
   char *tdir; char *tprefix;
   char *cdir; char *cprefix; char *cfiles; char *csuf;
   char *fdir; char *fprefix; char *pdir; char *filter;
```

Input: *A target* target, *the directory* tdir *which will receive the* LaTeX *files, a prefix* tprefix *which will be added to this* LaTeX *files, the directory* cdir *which contains the C files, the prefix* cprefix *of these C files, the names* cfiles *of these C files (without the usual suffix ".c"), the suffix* csuf *of these C files (usually* .c *or* .h*), the directory* fdir *which will contain the compiled parser, the prefix* fprefix *of this parser, the directory* pdir *which contains the sources of the parser and the name* filter *of the parser.*

Output: *The integer 1 if an error occured, 0 otherwise.*

Side effect: *The parser is compiled and the* LaTeX *files are created if* target *is non zero.*

Procedure 15 *Compiling* LaTeX *files to obtain* dvi *documentation.*

```
int TexToDvi(target, ddir, dprefix, tdir, tprefix, tfiles, inputs)
   int target;
   char *ddir; char *dprefix;
   char *tdir; char *tprefix; char *tfiles;
   char *inputs;
```

Input: *A target* target, *the directory* ddir *which will receive the* dvi *files, a prefix* dprefix *which will be added to this* dvi *files, the directory* tdir *which contains the* LaTeX *files, the prefix* tprefix *of these* LaTeX *files, the names* tfiles *of these* LaTeX *files (without the usual suffix* .tex*) and directories* inputs *where style* LaTeX *files can be found.*
Output: *The integer 1 if an error occured, 0 otherwise.*
Side effect: *The* LaTeX *files are compiled if* target *is non zero.*

3.5 Removing files

Procedure 16 *Removing files*

```
void CleaningRules(target, path, prefix, files, suffix)
   int target;
   char *path;
   char *prefix;
   char *files;
   char *suffix;
```

Input: *A target* target, *the directory* path *which contains files to remove, a prefix* prefix *of the files to remove, the names* files *of to remove, the suffix* suffix *of the files to remove.*
Output: *The integer 1 if an error occured, 0 otherwise.*
Side effect: *The files* prefix-names-suffix *are removed from* path. *If* path *is empty once these files are removed,* path *is removed too.*

Procedure 17 *Removing directories*

```
void TidyingRules(target, path)
   int target;
   char *path;
```

Input: *A target* target, *the directory* path *which contains files to remove.*
Output: *The integer 1 if an error occured, 0 otherwise.*
Side effect: *The directory and the files contained in this directory are removed.*

Chapter 4

The documentation parser

This program builds LaTeX documentation source from the C files comments. A few commands are used to perform clean output using zen.sty file. These commands are only valid inside a C-comment:

- \TEX{ZeNTeX} allows to include LaTeX code in the resulting file as is. It is also intended to enumerate the procedures of the C files. To include a procedure in the LaTeX file, there are fields that should be used to describe the function, all but the first of which are optional.

  1. \title should shortly describe the function. This field is needed but can be set to —.
  2. \input should describe the parameters of the function.
  3. \output should describe the output of the function.
  4. \side should describe the side effects of the procedure.
  5. \note allows to precise some programming features.

  Besides, as soon as a \title{} appears in a \TEX comment, the C-header of the following function is included and index entries for MakeIndex program are generated. This feature is disabled by the -noindex option in the command line.

- \TEX[ZeNTeX} behaves in the same way, except that short descriptions of the functions are included.

- \CiteC makes the C source to be included in verbatim mode in the LaTeX code.

- \EndCiteC ends the citation of C source.

- Pipes symbols | may delimit the programs names and declarations. Within pipes, the underscore symbols _ are automatically backslashed (_) the stars ⋆ transformed in $\star$ and the backslash \ transformed in $\backslash$.

Note 1 *By default, the output format of* LaTeX *is set to* a4paper. *It is possible to use default legal format by modifying the following compilation directive in file* zmake/zparse.h.

```
#define A4PAPER 1
```

Chapter 5

A zmake.c example

This example is the zmake.c source used to compile this documentation.

```
#include "zmake.h"
```

At first, global variables containing main paths and files are defined.

```
char *DOC_ROOT = "doc";
char *DOC_FILES = "intro title compile biblio zmake";
char *ZMAKE_DOC_FILES = "ztools zrules zmake";
char *ZMAKE_DOC_HFILES = "zmake";
char *PARSER = "zparse";
char *IPT = ".";
char *TEX = "tex";
char *DVI = "dvi";
```

The heart of a zmake.c is this procedure.

```
int main(argc, argv)
      int argc;
      char *argv[];
{
```

We will need these variables.

```
  char *LOCAL_TOP = TOPDIR;
  char *LOCAL_BIN = concat(LOCAL_TOP, concat("/", CompilingDir));
  char *LOCAL_TEX = concat(LOCAL_TOP, concat("/", TEX));
  char *LOCAL_DVI = concat(LOCAL_TOP, concat("/", DVI));
  char *LOCAL_IPT = LOCAL_TOP;
  char *LOCAL_INPUTS;
```

Then, Makefiles target are simulated as follows.

```
  int cc = 0;
  int prgm0link = 0; int prgm1link = 0; int prgm2link = 0;
```

```
int ludvi = 0;
int clean_obj = 0; int clean_lib = 0; int clean_bin = 0;
int clean_doc = 0; int clean_tex = 0; int clean_dvi = 0;
int tidy_lib = 0; int tidy_obj = 0; int tidy_bin = 0;
int tidy_doc = 0;
```

Targets which can be called from the command line are defined here.

```
int all = Target("all", argc, argv);

int ar = Target("ar", argc, argv);
int prgm0 = Target("test", argc, argv);
int prgm1 = Target("bench", argc, argv);
int prgm2 = Target("example", argc, argv);

int doc = Target("doc", argc, argv);
int tex = Target("tex", argc, argv);
int dvi = Target("dvi", argc, argv);

int archtidy = Target("archtidy", argc, argv);
int archclean = Target("archclean", argc, argv);
int clean = Target("clean", argc, argv);
int tidy = Target("tidy", argc, argv);

int none = Target("none", argc, argv);
```

Afterwards, ZMAKE initialisation is done here.

```
LOCAL_INPUTS = concat(concat(CURDIR, ":"), IPT);
LOCAL_INPUTS = concat(concat(LOCAL_INPUTS, ":"), LOCAL_TEX);

zmake_init("", "", "", "", argc, argv);
```

And pseudo makefiles targets can be handled as follows.

```
TargetImply(defaults, all);

TargetImply(all, doc);

TargetImply(doc, tex=ludvi);
TargetImply(dvi, tex=ludvi);

SrcToTex(tex,
          LOCAL_TEX, "",
          LOCAL_IPT, "", DOC_FILES, CSUF,
          LOCAL_BIN, "", LOCAL_IPT, PARSER);
```

```
SrcToTex(tex,
          LOCAL_TEX, "",
          ZMAKE_DIR, "", ZMAKE_DOC_FILES, CSUF,
          LOCAL_BIN, "", LOCAL_IPT, PARSER);
SrcToTex(tex,
          LOCAL_TEX, "h",
          ZMAKE_DIR, "", ZMAKE_DOC_HFILES, HSUF,
          LOCAL_BIN, "", LOCAL_IPT, PARSER);
SrcToTex(tex,
          LOCAL_TEX, "",
          ZMAKE_DIR, "", PARSER, HSUF,
          LOCAL_BIN, "", LOCAL_IPT, PARSER);
SrcToTex(tex,
          LOCAL_TEX, "",
          LOCAL_IPT, "", DOC_ROOT, CSUF,
          LOCAL_BIN, "", LOCAL_IPT, PARSER);

TexToDvi(dvi,
          LOCAL_DVI, "",
          LOCAL_TEX, "", ZMAKE_DOC_FILES,
          LOCAL_INPUTS);
TexToDvi(dvi,
          LOCAL_DVI, "h",
          LOCAL_TEX, "h", ZMAKE_DOC_HFILES,
          LOCAL_INPUTS);
TexToDvi(dvi,
          LOCAL_DVI, "",
          LOCAL_TEX, "", DOC_FILES,
          LOCAL_INPUTS);
TexToDvi(dvi,
          LOCAL_DVI, "",
          LOCAL_TEX, "", PARSER,
          LOCAL_INPUTS);
TexToDvi(ludvi,
          LOCAL_DVI, "",
          LOCAL_TEX, "", DOC_ROOT,
          LOCAL_INPUTS);
```

Cleaning rules are defined here.

```
TargetImply(tidy, tidy_bin=tidy_doc);
TargetImply(archtidy, clean_doc=clean_bin);
TargetImply(clean, clean_tex);
TargetImply(archclean, clean_tex);
TargetImply(clean_doc, clean_dvi=clean_tex);

CleaningRules(clean_tex, LOCAL_TEX, "", PARSER, ".tex");
CleaningRules(clean_tex, LOCAL_TEX, "", ZMAKE_DOC_FILES, ".tex");
```

```
  CleaningRules(clean_tex, LOCAL_TEX, "h", ZMAKE_DOC_HFILES, ".tex");
  CleaningRules(clean_tex, LOCAL_TEX, "", DOC_FILES, ".tex");
  CleaningRules(clean_tex, LOCAL_TEX, "", DOC_ROOT, ".tex");

  CleaningRules(clean_dvi, LOCAL_DVI, "", ZMAKE_DOC_FILES, ".dvi");
  CleaningRules(clean_dvi, LOCAL_DVI, "h", ZMAKE_DOC_FILES, ".dvi");
  CleaningRules(clean_dvi, LOCAL_DVI, "", DOC_FILES, ".dvi");
  CleaningRules(clean_dvi, LOCAL_DVI, "", PARSER, ".dvi");
  CleaningRules(clean_dvi, LOCAL_DVI, "", DOC_ROOT, ".dvi");

  TidyingRules(tidy_bin, LOCAL_BIN);
  TidyingRules(tidy_doc, LOCAL_TEX);
  TidyingRules(tidy_doc, LOCAL_DVI);
```

Exit is finally done here.

```
  zmake_close();

  exit(0);
}
```

Chapter 6

Bibliography

[1] D. Grune, B. Berliner, and J. Polk. *Concurrent Versions System.*

[2] T. Setz. *Integration von mechanismen zur unterstützung der fehlertoleranz in LiPS*, PhD thesis, Universitt des Saarlandes, Saarbrücken, Germany, Jan. 1996.

[3] P. DuBois. *Software portability with imake*, O'Reilly & Associates, 1993.

[4] L. Lamport. *LaTeX A documentation preparation system, Second edition*, Addison-Wesley, 1994.

Chapter 7

Concepts index