Amiga Programmer's Make

Version 1.4

Reference Manual

by Ben Eng

This software is a brought to you by

Jet Penguin Lavatories

Ben Eng

150 Beverley St. Apt.#1L
Toronto, Ontario
M5T 1Y6
CANADA

telephone: (416)–979–8761
e-mail: becker!jetpen!ben
BIX: jetpen

Contents

Abstract

Make is a programming utility used to automate the process of recompiling multiple interdependent source files into an output file (the goal). The Make program uses rules of inference to make the goal. The rules for making the goal are explicitly stated in an input file called the Makefile, and implicitly determined from builtin rules. Normally, the Makefile for a goal is written so that the only thing that needs to be done to recompile newly modified source files is to run the Make program.

Chapter 1

Distribution

This is version 1.4 of the Make utility (BMake) for the Amiga. For information regarding an updated version, if one exists, please contact the author by electronic mail, telephone or through the postal service in that order of priority. Details of how to contact me can be found in the `README` file associated with the distribution.

If you choose to make use of this software or distribute this software to other users, please read the GNU General Public License which is enclosed with the distribution as the file called `COPYING`.

If you did not receive this software from the author by magnetic media or direct transmission, then please take the time to register your version so that updates will continue to be offered in the future. A lack of response will indicate that the product does not deserve further attention and the software will be orphaned; a strong response will demand rapid improvements in future upgrades of the product.

To register simply send a letter or postcard with your name (and company, if it is using the product), address, and telephone number, along with the name and version number of the product you are registering.

If you are already registered or if you are presently registering, you may receive an upgrade direct from the author by sending a money order in the amount of CDN$20.00 (**twenty Canadian dollars**) payable to the author, Ben Eng, to the address given on the inside cover of this manual or in the `README` file, along with a letter requesting an upgrade (please, mention the name of the product). The latest release of the requested product will be promptly sent out to you, or at your request a later release will be mailed to you when it is available (ie. if you are waiting for a feature to be added). The price covers the cost of the media and shipping.

Chapter 2

The Make Program

2.1 Introduction

The Make program was written to reduce the tedium involved in developing software. A program under development usually needs to be recompiled many times per day. Typing out the full command line for each step of the compilation becomes exceedingly tedious, and a shell script is too crude to figure out the most efficient way to recompile a large set of source files. The Make program suits this purpose well.

With a properly written Makefile in the current directory, all that needs to be typed to build the project under development is `make`. To build only a subset of the project defined in the Makefile, simply type `make` *goalname*, where *goalname* is the name of the goal, that requires rebuilding. The actual dependencies that must be checked and the commands which must be executed are all taken care of in the Makefile and the builtin rules.

2.2 Command Line Arguments

Sometimes it is necessary to alter the behaviour of the Make program due to special circumstances. Command line arguments are provided to control how the Make program operates.

-a	make all dependencies regardless of modification time
-b	do not use builtin rules
-d	debug mode (enable printf)
+m#	sets the maximum line length to # (minimum=1024)
-n	prints commands to be executed, but do not execute them
-p	prints the database, but do not run
-t	make targets up-to-date by touching (commands not executed)
-v#	set verbose level to #; 0=silent
-f*makefile*	specify the makefile to run
var=*value*	assigns the value to a variable
wildcards	specify targets to make

The Make program requires Amiga OS 2.0 to run with maximum capability. Under Amiga OS 1.3, there is no wildcard support. Run the program with the argument `?` to see its usage, or `-h` to get more descriptive help on acceptable command line arguments.

The environment variable `ENV:system` can be set to `yes` if you wish the command execution to use the Amiga OS 2.0 System() call. If `ENV:system` is set to `no`, the Amiga OS Execute() call is used. The latter is desirable if you are still running with a shell that does not support the 2.0 conventions (WShell 1.2 falls under this category).

Chapter 3

The Makefile

3.1 Introduction

The Makefile is a text file written to maintain the construction of an output file from a set of source files. Normally, a Makefile is used to define the rules for building an executable binary file from a set of object files, which are created by compiling source files.

The Make program will read in the Makefile and use the rules to determine which source files require recompilation, and whether or not the executable file needs to be relinked. For large projects with dozens of source files, and complex interdependencies between the source and include files, the use of a Make program with a properly written Makefile is essential for ensuring that the executable is up-to-date. During the development cycle, the tedium of manually recompiling newly modified source files is removed from the programmer's responsiblity. The procedure is automated by the Make program, so that the programmer does not need to worry about which files need to be updated and how to perform the updating.

For most applications a Makefile is easy to write. A few minutes spent writing a Makefile can save the programmer a great deal of time that may have been wasted tediously typing in command lines to recompile a project. The Make program was designed to make the rebuilding procedure trivial and reliable.

3.2 Description

A typical Makefile is composed of three basic constructs: comments, variable definitions and rules. Any line beginning with a `#` character is considered to be a comment line. Comment lines are ignored by the Make program. Variables (also known as macros) and rules are complicated topics and they will be discussed in separate sections.

The Makefile is written to define the rules for making a goal up-to-date. A goal file is up-to-date if it has a modification datestamp that is more recent than all of its dependencies, after all dependencies are updated with respect to their dependencies. This criterion is applied recursively, such that the most deeply nested dependency will be made before any target file that depends on it.

3.3 Rules in a Makefile

Several types of rules are recognized by the Make program; they can be grouped into two categories: explicit rules and implicit rules. The explicit rules are known as target rules. The implicit rules exist as suffix rules and pattern rules (internally these two types of rules are represented in the same way).

3.3.1 Explicit Rules

Target rules are defined by a line that begins with one or more target names, followed by a colon, an optional list of dependency names, an optional semicolon and an optional command. Subsequent lines that begin with a tab character are command lines associated with the rule. The next line that does not begin with a tab character indicates the end of the current rule and the beginning of a new rule.

```
goal: target.o ; command three

target.o: source.c header.h
    command one
    command two
```

In the above example, the Make program will read the Makefile and find the first target name to be `goal`. It will then attempt to make `goal` up-to-date by first making its dependencies (namely, `target.o`) up-to-date. In order to make `target.o` up-to-date, Make requires a rule that contains `target.o` as a target name. Such a rule exists in the example Makefile with dependencies `source.c` and `header.h`, which must in turn be made before `target.o` can be made up-to-date. Since `source.c` and `header.h` are not defined as targets in the Makefile, the Make program must use builtin rules of inference to make those files up-to-date, or it must give up and report that is does not know how to make the target. As it turns out, the source and header files are known by builtin rules of inference to require no further action to be made up-to-date, so the Make program can continue to make `target.o` now that its dependencies have fulfilled their requirements. In order to make `target.o` up-to-date, `command one` must be executed successfully, and then `command two` must be executed successfully after that. Finally, the goal can be made by executing the command line, `command three`, and the Make program will terminate.

If there are no dependencies for a target, and a target needs to be remade by the Make program, then the target is always remade. A target with no dependencies will always have its commands executed when that target needs to be remade.

Dependencies

The dependencies of a target are the objects which a particular target depends upon. If a target depends upon an object, then the dependent object must be updated before the target may be updated. A target is not up-to-date if its dependencies are not all up-to-date.

Additional dependencies can be added to a target defined in another rule by writing a new rule which declares the additional dependencies. Only one of the rules may contain command lines, otherwise it is an error. An example where this might be done is given below:

```
target1.o target2.o : global.h
target1.o : one.h
target2.o : two.h
```

.ALWAYS

If a target has `.ALWAYS` as a dependency then that target will always have its dependencies remade and its commands executed.

.NEVER

If a target has `.NEVER` as a dependency then that target will never have its dependencies remade or its commands executed.

.ONCE

If a target has `.ONCE` as a dependency then that target will be made only once. When the target is encountered again, it will be skipped.

.INVISIBLE

If a target has `.INVISIBLE` as a dependency then that target will be invisible to all other targets. That is, the invisible target may be made, but the target that depended upon the invisible target will not be aware of the fact.

3.3.2 Implicit Rules

Pattern Rules

When no explicit rule is defined for a target, the Make program must use other means to determine how to make the target up-to-date. Rules of inference are defined by pattern rules. An example of a double pattern rule is:

```
%.o: %.c
    $(CC) -c $(CFLAGS) -o $@ $<
```

The above double pattern rule defines how to make any target with a name ending in `.o` if there is a corresponding dependency name that ends in `.c`. The commands associated with a pattern rule are executed if the dependent file is newer than the target file, which matches the pattern. The `%` character matches any number of characters in the target name. Only one such wildcard character may exist per name.

There may be only one dependency in a pattern rule (extra dependencies given on the line are ignored). The `%` character in the dependency pattern is replaced with the stem (the sequence of characters which were matched by the `%`) from the target name.

A single pattern rule of the form:

```
%.Z:
    compress $*
```

is the same as a double pattern rule, except that there is no dependency pattern. As a result, the automatic variable `$<` is not defined in the scope of a single suffix rule. Since there are no dependencies, any target matching a single suffix rule will always be remade; the commands will always be executed.

There may be more than one pattern rule applicable per target name. The first matching pattern rule will have priority. If several pattern rules are defined with the same target pattern, then the pattern rule that was defined earlier will have priority over all later definitions. Thus, pattern rules with the same target but different dependencies can be defined in natural prioritized order; the first one defined will be the first one applied to inference rules.

However, if a new definition is given that exactly duplicates both the target pattern and the dependency pattern then the new definition will replace the old pattern rule.

Suffix Rules

To maintain compatibility with older versions of Make, suffix rules can be defined. They are a poor man's style of pattern rules. An example of a double suffix rule is:

```
.c.o:
    $(CC) -c $(CFLAGS) -o $@ $<
.SUFFIXES: .c .o
```

The above suffix rule defines how to make any target with a name ending in `.o` if there is a corresponding dependency filename that ends in `.c`. The commands associated with a suffix rule are executed if the dependent file is newer than the target file.

A suffix rule has no dependencies listed to the right of the colon. If such dependencies exist, then the rule definition is considered to be an ordinary target rule.

The `.SUFFIXES` directive is used to cause the Make program to find all ordinary rules that can be interpreted as suffix rules matching the suffix arguments. Each suffix rule is transformed into its equivalent pattern rule at this time.

Pattern Rules versus Suffix Rules

Pattern rules should be used because they are far more flexible than suffix rules. Suffix rules are internally represented by pattern rules anyway, but an extra `.SUFFIXES` directive must be given before a suffix rule is recognized.

3.3.3 Command Lines

Command lines are executed in order of appearance if any dependency is newer than the target. The commands executed for a rule should do something that causes the target's modification date to be updated.

The Make program knows nothing about what the command lines mean or do. The command lines are simply passed to the shell for execution. If executing the command line results in an error, then the Make program will terminate.

Sometimes it is desirable to ignore the error returned by a command, because failure to execute the command does not affect the successful completion of the Makefile. If this is true then the error returned by a command can be ignored by preceding the command with a `-` character. For example:

```
clean:
    -delete #?.o
```

Normally, each command is printed (echoed) before execution. Echoing for each command can be disabled by preceding it with a `@` character. This does not affect its execution. The `@` character does not redirect the standard output of the program being executed. Redirection must be stated explicitly.

Special Commands

The `cd` command can be used to change the current directory of the Make program. This command is special; it is handled by the Make program itself, instead of being executed by the shell. After the current directory is changed, all subsequent commands which are executed from within Make will inherit the new current directory, until another `cd` command is issued. If no argument is given to `cd` then the original current directory is restored. The original current directory, where the Make program was executed, will be restored before the Make program exits.

Conditional commands are also handled internally by the Make program. Refer to section 3.6.2 on the behaviour of conditional commands.

Variable Assignment Commands

A command line containing the = character (before macro expansion) will be interpreted as a variable assignment (see section 3.4). Thus, variables can be given values which depend upon which rule was resolved. Variable assignment commands are executed in the context of the rule which is being resolved at run time.

3.3.4 Default Target Rule

Sometimes it might be desirable to define a rule that is only executed when all other rules have failed to make a target up-to-date. As a last resort, one might wish to simply `touch` the target file, or perform no action at all, and allow Make to continue without terminating with an error.

```
.DEFAULT:
    command lines
```

If the `.DEFAULT` target is given no commands (as is the default behaviour) then the Make program will terminate with an error whenever it encounters a target name that cannot be made.

3.3.5 Builtin Rules

By default, the Make program knows several builtin pattern rules. Here is a list of their definitions:

```
%,v:

%.a: RCS/%.a,v
    $(CO) -u $@
%.a:

%.c: RCS/%.c,v
    $(CO) -u $@
%.c:

%.h: RCS/%.h,v
    $(CO) -u $@
%.h:

%.i: RCS/%.i,v
    $(CO) -u $@
```

```
%.i:

%.o: %.c
    $(CC) -c $(CFLAGS) -o $@ $<

%.o: %.a
    $(AS) -c $(AFLAGS) -o $@ $<
```

Notice that assembly and C language headers and source files have no dependencies and require no commands to be executed in order to make them up-to-date. User defined rules should be added to the Makefile if that is not the case.

3.3.6 User Defined Builtin Rules

If the user needs to supplement the internal builtin rules with addition suffix rules and variable definitions, they can be declared in one of two files: `S:builtins.make` or `builtins.make`. If they exist, these two files are read (in the order listed above) before the Makefile by the Make program. All rules and variables in these files may be referenced (or overridden) in the Makefile.

3.4 Variables and Macros

Variables can contain macro definitions of multiple names or other text. They are useful for replacing multiple occurrences of the same body of text with references to a variable (macro expansion). A variable is defined in the Makefile with an assignment statement. Only the last assignment of a variable is recognized, because the rules are processed after the entire Makefile has been interpreted into internal rules by the Make program.

In a makefile, the line:

```
myvariable = source.c header.h
```

is used to assign the string `source.c header.h` to the variable named `myvariable`. In the following rule definition:

```
target.o: $(myvariable)
```

the $(myvariable) reference is expanded into the string assigned to `myvariable`. If the macro expansion contains further references to other variables, then they are also recursively expanded until the resulting string is fully expanded. Undefined macros are expanded to nothing (empty strings). ${myvariable} is equivalent to $(myvariable). For single character variable names, the parentheses or braces are not necessary to expand the macro; the single character may immediately follow the `$` .

Additional text can be concatenated to the end of the value of an existing variable by using the += operator, like this:

```
alpha = abc
alpha += def
```

The result of `$(alpha)` is now `abc def`. Notice the space inserted at the point of concatenation. This ensures that filenames are not split between different lines.

It is possible to define a variable which results in an infinitely recursive macro expansion. This is illegal, and it will result in an error when that variable is referenced.

A variable can be defined, such that its value is expanded immediately at the time of assignment. These are called simple variables, and they are assigned with the `:=` operator. Note that when a simple variable is assigned, any other references to variables will expand to their value as assigned above the line of the simple variable being assigned. Here is an example:

```
simplevar := $(myvariable)
```

The only difference between an ordinary variable and a simple variable is that the value of the simple variable is expanded only once during its assignment, whereas the value of an ordinary variable needs to be expanded every time it is referenced. Referencing a simple variable requires only a simple substitution, whereas referencing an ordinary variable requires a recursive macro expansion, which requires more work (memory, stack, and CPU instructions).

Variables referenced in the target and dependencies of a rule are expanded during the rule definition. Variables referenced in each command line associated with a rule are expanded when the command line is executed.

Both \$ and $$ expand to the character $, but the latter is recommended.

3.4.1 Automatic Variables

There is a set of variables that are automatically defined at runtime by the Make program.

$@	expands to: target name
$*	expands to: target name without its suffix
$<	expands to: the first dependency name
$^	expands to: all dependencies of target newer than target (not implemented)
$%	expands to: dependent member of the target archive (not implemented)
$?	expands to: all dependencies of the target archive (not implemented)

The value of an automatic variable depends on where it is referenced. An automatic variable has a different value according to the objects of the rule to which it applies. $@ does not expand in the target position. Automatic variables making reference to dependencies do not make sense when used in the place of a target name or dependency, so they will not be defined when referenced in those situations; those variables will expand to their proper values in command lines.

If a `D` extension appears in the automatic variable name then only the directory part of the value is returned. A `F` extension returns only the file part of the value. Thus, `$(@D)` is equivalent to `$(dir $@)`; in fact, all of these extensions are implemented as builtin macros.

3.4.2 Complex Variable names and Macro Expansions

Rarely is it necessary to apply what is discussed in this topic, but the information will be given for completeness. Because of the recursive nature of the macro expansion algorithm, it is possible to construct variables that act like arrays (or other complex data types) through the use of variable names that themselves contain a nested macro expansion. Here is an example of an obscure sort of Makefile that uses this technique:

```
A1 = one.c
A2 = two.c

target.o: ${A$(B)}
    $(CC) -c $(CFLAGS) -o $@ $<
```

Depending on whether the value of the variable `B` is set to `1` or `2`, the target file will depend on `one.c` or `two.c`.

Although there is no internal limit restricting the level to which macros can be nested (in the variable name and its value), it is recommended that nesting be kept to a minimum to conserve on memory consumption and stack usage. Each level of nesting requires at least an additional 2K of memory.

Please note that the maximum length of a variable name is limited to 256 characters.

3.5 Function Calls

Function calls of the form `$(function arguments)` or `${function arguments}` will be processed differently than a normal macro expansion. A function call acts like a macro expansion in that it returns a string. All spaces, except for the sequence of spaces before the first argument (after expansion), are significant.

String Functions

(see section 9.2 of the GNU Make manual)

String functions generally accept a set of input strings and perform a transformation to return a result string.

$ (filter *pattern*, text)

A pattern is a word that optionally contains a single wildcard character `%`. All words in `text` that do not match the `pattern` are removed, so that this function call returns only matching words.

$ (filter-out *pattern*, text)

All words in `text` that match the `pattern` are removed, so that this function call returns only non-matching words. The result is the complement of the result of $(filter pattern,text).

$ (findstring *find*, in)

If the `in` string contains the `find` string then the `find` string is returned, otherwise the result is an empty string

---

$ (patsubst *pattern*, replacement, text)

The result is a list of words derived from `text`. `text` is a whitespace separated list of words. Each word is compared with `pattern`. A `%` character in the pattern matches any number of characters; only one `%` may appear in the pattern. If the pattern does not match the word then the word is appended unchanged to the result, otherwise a substitution is performed on the word, the result of which is appended to the result.

If a substitution is necessary, then the resulting word is formed by any characters preceding the `%` of `replacement`, plus every character of the current word of `text` that was matched by the `%` of `pattern`, plus any characters following the `%` of `replacement`. In other words, if the `%` character occurs in both `pattern` and `replacement` then the substring matched by the `%` will be substituted for the `%` in the `replacement` string, which will then be used as the next word of the result. For example, $(patsubst pre%post,a%b,dogfood preAApost) will result in the string `dogfood aAAb`.

$ (sort *list*)

sorts the list of names into ascending lexical order.

$ (strip *string*)

strips leading and trailing whitespace from string, and replaces each embedded sequence of spaces with a single space.

$ (subst *from*, to, text)

replaces all occurrences in `text` that match the string `from` with the string `to`.

Filename Functions

Filename functions perform transformations on parts of pathnames. The disk objects do not have to actually exist in the filesystem, but the pathnames generally should follow the usual naming conventions. Three parts of a pathname may be operated upon: the directory part, the filename part and the suffix part of the filename.

$ (dir *names*)

Extracts the directory part of each pathname in `names`, including the trailing slash or colon. No checking is done to determine whether the result is actually a directory or not; this command acts only as a string function, and knows nothing about the organization of the filesystem.

$ (notdir *names*)

Extracts the file part of each pathname in `names`. The result is the complement of $(dir names).

$ (suffix *names*)

Extracts only the suffix of each pathname in `names`. The suffix is the rightmost part of the name starting at the last `.` character.

$ (basename *names*)

Extracts all but the suffix of each pathname in `names`. The result is the complement of $(suffix names).

$ (addsuffix *suffix*, names)

The result is the list of names with `suffix` appended to each word of the list.

$ (addprefix *prefix*, names)

The result is the list of names with `prefix` prepended to each word of the list.

$ (join *list1*, list2)

The result is the wordwise concatenation of `list1` with `list2`.

$ (word *n*, text)

Returns the nth word of `text`. The first word of text is numbered 1, and the last word of text is numbered $(words text).

$ (words *text*)

Returns the number of words in `text`.

$ (firstword *names*)

Returns the first word of `names`. The result is the same as $(word 1,names)

$ (wildcard *pattern*)

The `pattern` argument is an Amiga OS wildcard pattern. The result is a list of files matching that pattern.

NOTE: Wildcards are only supported under Amiga OS 2.0

An example of a generic makefile that can be used to compile a program, that depends only upon all of the .c files in the current directory, is given below:

```
SRCS := $(wildcard #?.c)
OBJS := $(subst .c,.o,$(SRCS))

a.out: $(OBJS)
    $(CC) -o $@ $(CFLAGS) $(OBJS)
```

Special Functions

$ (foreach *var*, list, text)

(see section 9.4 of the GNU Make manual)

This function call is not implemented. A workaround might be to unroll the loop manually.

$ (getenv *name*)

The result is the contents of the environment variable `ENV:name`. This function call is offered to allow the Makefile to explicitly reference an environment variable, instead of the internal Make variable.

Note: referencing an undefined variable for the first time automatically assigns the result of $(getenv macro) to the variable.

$ (origin *variable*)

(see section 9.5 of the GNU Make manual)

Not implemented.

$ (shell *command line*)

(see section 9.6 of the GNU Make manual)

This function call is not implemented. A workaround might be to redirect the `command line` standard output into an environment variable and then capture the environment variable's value into a macro for Make to use.

3.6 Conditionals

3.6.1 Conditional Directives

A conditional directive can be used to control which parts of a Makefile are executed, based upon the value assigned to certain variables. The familiar

```
if condition
    ...true...
else
    ...false...
endif
```

syntax is used for this purpose, where ...`true`... represents any number of lines which are executed if the condition is true, and ...`false`... represents any number of lines which are executed if the condition is false. Each of the three conditional directives must appear as the first word on a line. The `else` directive is optional. The condition may be one of the following:

eq(arg1,arg2)	true if arg1 is exactly equal to arg2
neq(arg1,arg2)	true if arg1 is not equal to arg2
def(variable)	true if variable is defined
ndef(variable)	true if variable is undefined
exists(pathname)	true if pathname exists in the filesystem
nexists(pathname)	true if pathname does not exist in the filesystem

Nested conditionals are acceptable. There is no way to perform logical **AND** and logical **OR** operations within the condition expression, so nested conditionals will have to be used instead to do the same job. Each `if` and `else` must have a matching `endif` directive associated with its conditional construct.

The `else` and `endif` directives must reside in the same Makefile as their corresponding `if` directive. A conditional construct cannot be split across Makefiles.

The use of conditional directives in a Makefile is vaguely analogous to the use of `#if`, `#else`, and `#endif` preprocessor directives in C source code to control conditional compilation.

3.6.2 Conditional Commands

The conditional commands can be used to control the behaviour of the command lines of rules. The syntax is exactly the same as the syntax for conditional directives, except that conditional commands are indented with a tab; they appear in the body of a rule along with other commands. Nesting is supported.

Here is an example of where a conditional command might be useful for checking out RCS files, when the RCS directory is not accessible by the Make program (when knowledge of RCS file organization is private to RCS itself):

```
%.c:
    if nexists($@)
        $(CO) -u $@
    endif
```

3.7 Pragma Directive

Command line arguments for the Make program may be added to a line beginning in `pragma`. For example,

```
pragma +m2048
```

will set the maximum line length to at least 2048 bytes, if that parameter has not already been set to a higher value.

3.8 Include Directive

Other Makefiles can be included from a Makefile with the `.INCLUDE` directive (it must appear at the beginning of a line, like any other directive). For example,

```
.INCLUDE submakefile
```

will read in all the rules and macros from a file called `submakefile` and then continue reading the current Makefile. This can be done from the builtin Makefiles as well as any user Makefile. Nesting of includes is allowed; the maximum nesting level is indefinite.

3.9 Phony Directive

The `.PHONY` directive can be used to mark a target as having no correspondence to a real file. This is useful for propagating the proper up-to-date condition past a phony target rule. For example,

```
all:  target1 target2

.PHONY: all
```

Chapter 4

Building Make

4.1 Compiling

The source code for this program was written for the DICE C compiler, version 2.06.37, by Matthew Dillon. The tab size is set to 4. You will require the 2.0 Amiga include files from CATS of Commodore-Amiga to compile this program. The source code should compile under any ANSI C compiler for the Amiga.

If the executable needs to be recompiled then the following procedure should be used.

Rename the Make program to `make` and move it to a directory in your shell's execution path. The Makefile supplied with the program is probably incompatible with more primitive versions of Make.

The `include/scdir.h` header file should be copied to a directory in the include path for your compiler. The compiler should be set up to compile using the Amiga OS 2.0 (or later) include files.

The link library `ben.lib` needs to be made first. Change your directory to the `ben` directory and type `make`. If all goes well, then the `ben.lib` library will be created; it should be moved to the library search path for your compiler, so that the linker will be able to find it.

Change your directory to the main source directory for the make utility and type `make` to create the binary executable file bmake. This file can be renamed to the standard `make`, and placed in the command search path for convenience.

4.2 Parameters

`DEBUG` = (1 default) non-zero if debug calls are activated

Defining `DEBUG` to 0 will disable code generation for the debugging option. The -d command line option will be disabled if `DEBUG` is 0. `DEBUG` is enabled by default (in `make.h`). Enabling `DEBUG` adds about 2400 bytes to the size of the final executable.

`FNCALLS` = (1 default) non-zero if function calls are activated

If function calls are to be recognized, this parameter should be defined to be non-zero. Disabling function calls will make the executable somewhat smaller, and memory requirements will be lower. Enabling `FNCALLS` adds about 4600 bytes to the size of the final executable.

`MAXSUFFIX` = (16 default) the maximum number of characters in a suffix

This parameter limits the maximum size of filename extensions recognized by the Make program. This affects suffix rules and other operations that operate on suffixes. Suffixes are allocated on the stack and dynamically in this fixed size.

`MAX_MACRONAME` = (256 default) the maximum number of characters in a name

This parameter limits the size of a macro name.

`MAXPATHNAME` = (108 default) the maximum number of characters in a pathname

This parameter limits the size of a pathname that can be used to resolve suffix rules. Dynamically allocated.

4.3 Options

In the Makefile, the `-gs` option can be added to `CFLAGS` to enable DICE's dynamic stack code. It makes the program grow in size by a couple of kilobytes, and it slows down execution by a few milliseconds, but it does make it safe to run the Make program on a huge Makefile that makes extensive use of recursive macros and rules, without having to worry about overflowing the stack.

4.4 Debugging

If debugging is enabled (`DEBUG`=1), then the command line arguments `-d -v9 +l` will generate a verbose log file called `make.log`, which will give some insight into what the Make program was attempting to do. The +`l` parameter directs Make's output to `make.log`. Higher verbosity levels will cause more details to be logged.

Chapter 5

Compatibility

What follows is a list of features, which are not yet supported in this implementation of Make:

- the `override` directive
- the `foreach` function
- the `origin` function
- the `shell` function
- static pattern rules
- double colon rules
- `VPATH` search path for dependencies
- the `vpath` directive
- communicating options to a submake
- $(macro:from=to) macro substitutions
- the automatic variables $ˆ, $? and $%
- the directives `.IGNORE`, `.PRECIOUS` and `.SILENT`
- archives are not supported.
- the automatic variable `MAKELEVEL` is not implemented.
- the `MAKEFILES` variable is not supported
- parallel execution is not supported.
- the special significance of + in a command line

What follows is a list of incompatibilities:

- the command line arguments are different
- GNU Make allows conditional expressions in five forms:
  ifeq (arg1,arg2)
  ifeq 'arg1' 'arg2'
  ifeq "arg1" "arg2"
  ifeq 'arg1' "arg2"
  ifeq "arg1" 'arg2'
  this implementation only allows the first form.
- the AmigaDOS wildcard syntax is different than that used in Unix.
- the behaviour of certain pattern rules may not be what is expected because GNU Make only matches patterns in the file part of the pathname, whereas the entire pathname is matched in this implementation.
- executing the Make program from within a Makefile is untested, and the author does not know what to expect.
- None of the builtin rules are the same. Only a very basic set of C language rules have been included to save memory. Embellishing the builtin rules can be done in one of the Makefiles which are read in before the user Makefile.
- When using the `dir` function, pathnames without a directory part are not translated into `./` for the current directory, because in AmigaDOS, the current directory is the null string. Thus `dir` and `nodir` may not be undone with the `join` function.
- GNU Make does not allow conditional commands, but they are supported in this program.
- GNU Make does not support the `exists` or `nexists` conditions, but they are supported in this program.
- The use of quotation marks and backslash characters to delimit strings and escape characters respectively is ignored on the most part in this program.
- GNU Make may consider the dependency of an implicit rule (a pattern rule) to be terminal; ie. there is no need for the dependency of an implicit rule to be made first. This program does not consider such a dependency as terminal. A single suffix (pattern) rule with no dependencies and no commands may be used to terminate the search.