The `make` utility

Basics

`make` is a utility that helps keep the executable versions of programs current. It automatically updates a target file when changes are made to the files used to build the target.

`make` keeps a *target* program up-to-date by reading a *makefile* and *making* the target selected from it. By default the makefile is named `makefile`.

In its simplest form, a makefile is a sequence of *explicit* rules, each of the form (where brackets enclose optional parts):

```
f: [f1 ... fn]
        [command1]
        [command2]
        ...
```

`f` is said to be the *target* of the rule and to *directly depend* on the *sources* `f1...fn`. Each `command` is a command for the shell from which `make` is invoked.

A target is said to (indirectly) *depend* on `g` if one of its sources either is `g` or one indirectly depends on `g`. It is required that no target depend on itself (*autodependency*).

Each target must appear in exactly one rule.

To make a target `f`, `make` performs the following steps:

1: **for each** target `f1` **in** sources of `f`'s rule **do**
2: make `f1`;
3: **if** target file `f` does not exist **or**
4: `f` has no sources **or**
5: any source file is more recent than target file `f` **or**
6: any commands have been performed to make sources in step 2
7: **then** execute commands in `f`'s rule (if any)

Note that `make` performs a sort of postorder visit of the directed acyclic graph textually represented by the makefile's rules.

`make f` fails ("don't know how to make `f`") if `f` is not an existing file or a target for some rule or is a target for a rule with neither source nor commands.

Structure of rules

Full explicit rules

An explicit rule can also have multiple targets. This is equivalent to a set of rules, one for each target, having all the same body.

The full syntax is:

```
target [target] ...: [[{path}] source] ...
        [command]
        [command]
        ...
```

where:

- line separations are significant; blank lines are ignored;
- target must start in column 1;
- the first source must be preceded by at least one space or tab after the colon;
- commands must be indented, by some whitespace;
- target and source file names can contain complete path names and wildcards,
- `path` is a list of directories where `make` should look for the `source` following `{path}`.

It is possible to have multiple explicit rules with the same target, followed by two colons. These tell `make` to expect and perform (rather than ignore) additional explicit rules for the current target.

Implicit Rules

Implicit rules apply to all files that have certain identifying extensions. The syntax is:

```
[{source-dir}].source-extension.[{target-dir}]target-extension:
        [command]
        ...
```

This defines any file `name.target-extension` to be a target for file `name.source-extension`, for all `name`. Moreover:

- `{source-dir}` tells `make` to search for source files in `source-dir`;
- the period before `source-extension` must not be preceded by white space;
- `{target-dir}` tells `make` to place target files in `source-dir`

To determine the sources for target `name.t`, `make` uses an implicit rule

```
.s.t:
        commands
```

if and only if:

- an explicit rule with target `name.t` either does not exist or has no commands,
- `name.s` exists as a file or is a target in an explicit or implicit rule of the makefile

If no rule with target `name.t` exists, `make` behaves as with the explicit rule:

```
name.t: name.s
        commands
```

If there is a rule with no commands

```
name.t: sources
```

`make` behaves as though this rule were:

```
name.t: sources name.s
        commands
```

If several implicit rules are eligible to make a target, `make` chooses the first, and `make -N` the last, unless a `.suffixes` directive gives a higher priority to the extension starting another implicit rule.

Command Prefixes

A `command` can take the following prefixes: `@`, `-`*num*, `-` and `&`.

`@` prevents `make` from displaying `command` before executing it.

If `command` terminates with an error exit code *err*, `make` stops processing and deletes the current target file. With the prefix `-`*num*, this happens only if *err* *num*. With `-`, `make` does not stop.

`&` causes `make` to execute the command that the `$**` or `$?` macros in an explicit rule expand to.

Command Body

The command body is any shell command.

Borland C's `make` supports `<`, `>`, and `>>` redirection, but not pipes, and adds the `<<` and `&&` operators.

The `&&` operator is used for a `command` in a makefile thus:

```
        command args &&|
line 1
...
line N
| moreargs
```

This causes `make` to create a temporary file `make`*n*`.$$$` and execute

```
        command args make.$$$ moreargs
```


Any character can be used as a delimiter in lieu of `|`.

The `<<` operator is similar, but the temporary file acts as the standard input to the command.

All temporary files are deleted unless `make` has a specific command line option.

Comments and special characters

# indicates that the rest of the line is a comment.

\ as the last character in a line acts as a line continuation character. (Does not apply to comments.)

Directives

A makefile can contain the following directives:

Name	Action
`.autodepend`	turns on autodependency checking
`.noautodepend`	
`!error`	causes `make` to stop and print an error message
`!if`	expressions with macros / C-like syntax
`!elif`	
`!else`	
`!endif`	
`!ifdef`	depends on macro definitions
`!ifndef`	
`!undef`	forget definition for a specified macro
`.ignore`	ignore return value of a command
`.noignore`	
`!include`	specifies a file to include in the makefile
`.path.ext`	`path` is searched for files with extension `.ext`
`.precious`	do not delete specified target even if commands to build it fail
`.silent`	do not print commands before executing them.
`.nosilent`	
`.swap`	(only for real mode `make`): swap `make` in and out of memory
`.noswap`	
`.suffixes`	specifies a list of extensions

Macros

Macro definitions take the form

```
  macro-name = expansion-text
```

where:

- `macro-name` should be a string of letters and digits with no whitespace in it. Case is significant in macros. There can be whitespace between macro-name and the equal sign.
- `expansion-text` is any arbitrary string containing letters, digits, whitespace, and punctuation.

If macro-name has previously been defined, either by a macro definition in the makefile or by the `-D` option on the `make` command line, the new definition replaces the old.

Macros are invoked in the makefile by: `$(macro-name)`; parentheses are always needed, except for predefined macros)

When `make` finds a macro invocation, it replaces the invocation by the macro's `expansion-text`. If the macro is not defined, `make` replaces it with the null string, or, if the macro name is a shell environment variable with the relevant definition string.

Macro expansion issues

Macros cannot be invoked on the left side of a macro definition. They can be used on the right side, but they are not expanded until the macro being defined is invoked.

Macro invocations are expanded immediately in rule lines.

Macro invocations are expanded immediately in `!if` and `!elif` directives. If the macro being invoked in the directive is not currently defined, it is expanded to the value `0` (`FALSE`).

Macro invocations in commands are expanded when the command is executed.

The macro invocation `$(macro-name:text1=text2)` (no spaces around `:` and `=`) is replaced by the definition string for `macro-name` with every occurrence of `text1` replaced by `text2`.

Predefined macros

`make` comes with the following built-in macros.

Macro	Expands To
`$d`	`1` if following macro name is defined; `0` if not
`$*`	base file name (no ext, with path, if generated)
`$<`	full file name (with ext, with path, if generated)
`$:`	file path (no name/ext)
`$.`	file name and extension (no path)
`$&`	file name only (no path, no ext)
`$@`	full target name with path
`$**`	all dependents
`$?`	all out of date dependents
`__MSDOS__`	`1` if running `make` under DOS
`__MAKE__`	`make`'s version number in hex
`MAKE`	`make`'s executable filename
`FLAGS`	any options used on the `make` command line
`MAKEDIR`	directory from which `make` was run

In an explicit/implicit rule `$*` `$<` `$:` `$.` `$&` refer to target/dependent.

If there isn't a predefined filename macro to give the needed parts of a filename, macro modifiers can be used extract any part of a filename macro. The format is:

```
$(macro[D | F | B | R])
```

where the modifiers are:

Mod	What part of the filename	Example
`D`	drive and directory	`$(<D) = C:\OBJS\`
`F`	base and extension	`$(<F) = BOB.OBJ`
`B`	base only	`$(<B) = BOB`
`R`	drive, directory, and base	`$(<R) = C:\OBJS\BOB`

Invoking `make`

```
make [option option ...] target target ...
```

Command-line options are introduced by a `-` or a `/`. Case is significant.

Option	
-? or -h	prints a help message (default options followed by +
-a	automatic dependency check
-B	builds all targets regardless of file dates
-d*dir*	sets *dir* for swap file (must be used with -S
-D*identifier*	defines *identifier* to string consisting only of character 1
-D*iden*=*string*	defines *iden* to *string* (which cannot have whitespace)
-e	environment variables take precedence on macros with same name
-f*filename*	makefile is *filename*, or *filename*.mak if *filename* does not exist and has no extension. The space after -f is optional
-i	continues regardless of exit status of commands
-I*dir*	search for include files in *dir* (in addition to current directory)
-K	do not delete temporary files
-m	display date and time stamp of each file as it is processed
-n	prints commands but do not perform them
-N	increase make's compatibility with Microsoft's nmake
-P	display macro definitions and implicit rules before executing makefile
-r	ignore builtins.mak
-s	do not print commands before executing
-S	Swaps real mode make out of memory while executing commands. This makes room for very large modules to be compiled.
-U*identifier*	undefines any previous definitions of *identifier*
-W	write current non-string command line options to make.exe

Examples

```
# Builtins.mak
#
CC = BCC
AS = TASM
RC = RC
.asm.obj:
      $(AS) $(AFLAGS) $&.asm

.c.exe:
      $(CC) $(CFLAGS) $&.c

.c.obj:
      $(CC) $(CFLAGS) /c $&.c

.cpp.obj:
      $(CC) $(CPPFLAGS) /c $&.cpp
```



```
.rc.res:
      $(RC) $(RFLAGS) /r $&

.suffixes: .exe .obj .asm .c .res .rc
```

```
# * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
#
# This makefile was generated by the utility PRJ2MAK.EXE
#
# PRJ2MAK converts project files (filename.PRJ) into .MAK files.
#
# The .PRJ file for this example is in the EXAMPLES directory
# of the Container Class Library.
#
# * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

.AUTODEPEND

#          *Translator Definitions*
CC = bcc +DIRECTRY.CFG
TASM = TASM
TLIB = tlib
TLINK = tlink
LIBPATH = ..\..\LIB;..\LIB
INCLUDEPATH = ..\..\INCLUDE;..\INCLUDE


#          *Implicit Rules*
.c.obj:
  $(CC) -c {$< }

.cpp.obj:
  $(CC) -c {$< }

#          *List Macros*


EXE_dependencies =  \
 directry.obj \
 filedata.obj \
 testdir.obj \
 ..\lib\tclasss.lib

#          *Explicit Rules*
directry.exe: directry.cfg $(EXE_dependencies)
  $(TLINK) /v/x/c/P-/L$(LIBPATH) @&&|
c0s.obj+
directry.obj+
filedata.obj+
testdir.obj
directry
#          no map file
```



```
..\lib\tclasss.lib+
graphics.lib+
emu.lib+
maths.lib+
cs.lib
|


#          *Individual File Dependencies*
directry.obj: directry.cfg directry.cpp

filedata.obj: directry.cfg filedata.cpp

testdir.obj: directry.cfg testdir.cpp

#          *Compiler Configuration File*
directry.cfg: directry.mak
  copy &&|
-v
-vi
-w-ret
-w-nci
-w-inl
-w-par
-w-cpt
-w-dup
-w-pia
-w-ill
-w-sus
-w-ext
-w-ias
-w-ibc
-w-pre
-w-nst
-I$(INCLUDEPATH)
-L$(LIBPATH)
| directry.cfg
```