The Makefile

Make reads its instructions from text files. An initialization file is read first, followed by the makefile. The **initialization file** holds instructions for all "makes" and is used to customize the operation of Make. Make automatically reads the initialization file whenever it starts up. Typically the initialization file is named `make.ini` and it resides in the directory of `make.exe` and `mkmf.exe`. The name and location of the initialization file is discussed in detail on Page .

The **makefile** has instructions for a specific project. The default name of the makefile is literally `makefile`, but the name can be specified with a command-line option.

With a few exceptions, the initialization file holds the same kind of information as does a makefile. Both the initialization file and the makefile are composed of the following components: **comments**, **dependency lines**, **directives**, **macros**, **response files**, **rules** and **shell lines**.

Continued Makefile Lines

Lines in the makefile can be very long. For easier reading a long line can be broken up by putting "**\enter**" as the last characters of this line and the rest of this (logical) line on the next (physical) line of the makefile. For example:

```
first_part_of_line second_part_of_line
```

is the same as:

```
first_part_of_line \
second_part_of_line
```

Comments [Top]

The simplest makefile statement is a **comment**, which is indicated by the comment character "#". All text from the comment character to the end of the line is ignored. Here is a large comment as might appear in a makefile to describe its contents:

```
#
# Makefile for Opus Make 6.1
#
# Compiler: Microsoft C 6.0
# Linker: Microsoft Link 5.10
#
```

The comment character could also be used at the end of another makefile statement:

```
some makefile statement # a comment
```

Comments and Continued Makefile Lines

If "\enter" appears on a commented line, the comment acts until the end of the line and the following line is still continued. For example:

```
line_one \
line_two # more_line_two \
line_three
```

is the same as:

```
line_one line_two line_three
```

Rules [Top]

A **rule** tells Make both *when* and *how* to make a file. As an example, suppose your project involves compiling source files `main.c` and `io.c` then linking them to produce the executable `project.exe`. Withholding a detailed explanation for a bit, here is a makefile using Borland C which will manage the task of making `project.exe`:

The Example Makefile

```
project.exe : main.obj io.obj
        tlink c0s main.obj io.obj, project.exe,, cs /Lf:\bc\lib

main.obj : main.c
        bcc –ms –c main.c

io.obj : io.c
        bcc –ms –c io.c
```

This makefile shows three rules, one each for making `project.exe`, `main.obj`, and `io.obj`. The rules as shown above are called **explicit rules** since they are supplied explicitly in the makefile. Make also has **inference rules** that generalize the make process. Inference rules are discussed on Page and in the User's Guide on Page .

We will return to and modify this example during the course of this tutorial.

Dependency Lines: When to Build a Target [Top]

The lines with the colon "**:**" in them are called **dependency lines**. They determine when the target is to be rebuilt.

To the left of the colon is the target of the dependency. To the right of the colon are the **sources** [1] needed to make the target. A dependency line says “the target depends on the sources.” For example, the line:

```
project.exe : main.obj io.obj
```

states that `project.exe` depends on `main.obj` and `io.obj`. At run time Make compares the time that `project.exe` was last changed to the times `main.obj` and `io.obj` were last changed. If either source is *newer* than `project.exe`, Make rebuilds `project.exe`. The last-changed time is the target's time as it appears in the file-system directory. This time is also known as the target's **timestamp**.

The Make Process is Recursive

It is a basic feature of Make that a target's sources are made before the timestamp comparison occurs. The line:

```
project.exe : main.obj io.obj
```

implies “make `main.obj` and `io.obj` before comparing their timestamps with `project.exe`.” In turn:

```
main.obj : main.c
```

says “make `main.c` before comparing its timestamp with `main.obj`.” You can see that if `main.c` is newer than `main.obj`, `main.obj` will be rebuilt. Now `main.obj` will be newer than `project.exe`, causing `project.exe` to be rebuilt.

Additional Dependencies

In C and in other programming languages it is possible to include the contents of a file into the file currently being compiled. Since the compiled *object* depends on the contents of the included file, we add the included file as a source of the *object* file.

Assume each of `main.c` and `io.c` include `def.h`. We can either change two dependency lines in the makefile:

```
main.obj : main.c becomes main.obj : main.c def.h
io.obj : io.c becomes io.obj : io.c def.h
```

or add a new line which lists only the additional dependencies:

```
main.obj io.obj : def.h
```

Notice that there are two targets on the left of the colon. This line means that both `main.obj` and `io.obj` depend on `def.h`. Either of these methods are equivalent. The example makefile now looks like:

```
project.exe : main.obj io.obj
        tlink c0s main.obj io.obj, project.exe,, cs /Lf:\bc\lib

main.obj : main.c
        bcc –ms –c main.c
```

```
io.obj : io.c
        bcc –ms –c io.c

main.obj io.obj : incl.h
```

Shell Lines: How to Build a Target [Top]

The indented lines that follow each dependency line are called **shell lines**. Shell lines tell Make how to build the target. For example:

```
project.exe : main.obj io.obj
        tlink c0s main.obj io.obj, project.exe,, cs /Lf:\bc\lib
```

tells Make that making `project.exe` requires running the program `tlink` to link `main.obj` and `io.obj`. This shell line would be run only if `main.obj` or `io.obj` was newer than `project.exe`.

For `tlink`, `c0s` is the small model start-up object file and the `cs` is the small model library. The `/Lf:\bc\lib` flag tells `tlink` that the start-up object file and library files can be found in the `f:\bc\lib` directory.

A target can have more than one shell line, listed one after the other, such as:

```
project.exe : main.obj io.obj
        echo Linking project.exe
        tlink c0s main.obj io.obj, project.exe,, cs /Lf:\bc\lib >tlink.out
```

The first line shows that command processor commands can be executed by Make. The second line shows redirection of output, where the output of the `tlink` program is redirected to the `tlink.out` file.

After each shell line is executed, Make checks the shell line **exit status**. By convention, programs return a 0 (zero) exit status if they finish without error and non-zero if there was an error. The first shell line returning a **non-zero exit status** causes Make to display the message:

```
OPUS MAKE: Shell line exit status exit_status. Stop.
```

This usually means the program being executed failed. Some programs return a non-zero exit status inappropriately and you can have Make ignore the exit status by using a **shell-line prefix**. Prefixes are characters that appear before the program name and modify the way Make handles the shell line. For example:

```
project.exe : main.obj io.obj
        – tlink c0s main.obj io.obj, project.exe,, cs /Lf:\bc\lib
```

The “–” prefix tells Make to ignore the exit status of shell line. If the exit status was non-zero Make would display the message:

```
OPUS MAKE: Shell line exit status exit_status (ignored)
```

See Page for more information on shell lines and shell-line prefixes.

Macros [Top]

The example makefile is reproduced here:

```
project.exe : main.obj io.obj
        tlink c0s main.obj io.obj, project.exe,, cs /Lf:\bc\lib

main.obj : main.c
        bcc –ms –c main.c

io.obj : io.c
        bcc –ms –c io.c

main.obj io.obj : def.h
```

We see that the text "`main.obj io.obj`" occurs repeatedly. To cut down on the amount of repeated text, we can use a macro definition to assign a symbol to the text.

Defining Macros in the Makefile

A **macro definition** line is a makefile line with a macro *name*, an equals sign "**=**", and a macro *value*. In the makefile, expressions of the form `$(`*name*`)` or `${`*name*`}` are replaced with *value*. If the macro name is a single letter, the parentheses or braces are optional (*i.e.* `$X`, `$(X)` and `${X}` all mean "the value of macro `X`").

Here is the above example written with the introduction of four macros:

```
OBJS = main.obj io.obj
MODEL = s
CC = bcc
CFLAGS = –m$(MODEL)

project.exe : $(OBJS)
        tlink c0$(MODEL) $(OBJS), project.exe,, c$(MODEL) /Lf:\bc\lib

main.obj : main.c
        $(CC) $(CFLAGS) –c main.c

io.obj : io.c
        $(CC) $(CFLAGS) –c io.c

$(OBJS) : incl.h
```

The value of the `OBJS` macro is the list of object files to be compiled. The macro definitions for `MODEL`, `CC` and `CFLAGS` were introduced so that it is easier to change the compiler memory model, name of the C compiler and its options.

Make automatically imports environment variables as macros, so you can reference an environment variable such as `PATH` with the makefile expression `$(PATH)`.

Defining Macros on the Command Line

Macros can be defined on the Make command line. For example:

```
make CFLAGS=-ms
```

would start up Make and define the macro `CFLAGS` with the value "`-ms`". Macros defined on the command line take precedence over macros of the same name defined in the makefile.

If a command-line macro contains spaces, it must be enclosed in double quotes as in:

```
make "CFLAGS=-ms -z -p"
```

Run-Time Macros

Make defines some special macros whose values are set dynamically. These macros return information about the current target being built. As examples, the **`.TARGET`** macro is name of the current target, the **`.SOURCE`** [2] macro is the name of the inferred source (from an inference rule) or the first of the explicit sources and the **`.SOURCES`** [3] macro is the list of all sources.

Using run-time macros the example can be written:

```
OBJS = main.obj io.obj
CC = bcc
MODEL = s
CFLAGS = -m$(MODEL)

project.exe : $(OBJS)
        tlink c0$(MODEL) $(OBJS), $(.TARGET),, c$(MODEL) /Lf:\bc\lib

main.obj : main.c
        $(CC) $(CFLAGS) -c $(.SOURCE)

io.obj : io.c
        $(CC) $(CFLAGS) -c $(.SOURCE)

        $(OBJS) : incl.h
```

As you can see, the shell lines for updating `main.obj` and `io.obj` are identical when run-time macros are used. Run-time macros are important for generalizing the build process with inference rules, as shown on Page .

Macro Modifiers [Top]

Macros are used to reduce the amount of repeated text. They are also used in inference rules to generalize the build process. We often want to start with the value of a macro and modify it in some manner. For example, to get the list of source files from the `OBJS` macro we can do:

```
SRCS = $(OBJS,.obj=.c)
```

This example uses the "*from=to*" **macro modifier** to replace the *from* text in the expansion of `OBJS` with the *to* text. The result is that `$(SRCS)` is "`main.c io.c`". In general, to modify a macro expand it with:

```
$(name,modifier[,modifier ...])
```

Each *modifier* is applied in succession to the expanded value of *name*. Each modifier is separated from the next with a comma.

Filename Components

There is a set of macro modifiers for accessing parts of file names. For example, with the macro definition:

```
SRCS = d:\src\main.c io.asm
```

Some of the modifiers are:

Modifier, and description	Example	Value
D, the directory	`$(SRCS,D)`	`d:\src .`
E, the extension (or suffix)	`$(SRCS,E)`	`.c .asm`
F, the file name	`$(SRCS,F)`	`main.c io.asm`

Tokenize

Another modifier is the "W*str*" modifier, which replaces whitespace between elements of the macro with *str*, a string. The *str* can be a mix of regular characters and special sequences, the most important sequence being "`\n`" which represents a newline character (like hitting the enter key). For example:

```
$(OBJS,W space +\n) is          main.obj +
                                io.obj
```

Other Modifiers

Other modifiers include: "@" (include file contents), "LC" (lower case), "UC" (upper case), "M" (member) and "N" (non-member). The "M" and "N" modifiers and the "S" (substitute) modifier use *regular expressions* for powerful and flexible pattern-matching. See Page for more information on all macro modifiers.

Inference Rules [Top]

Inference rules generalize the build process so you don't have to give an explicit rule for each target. As an example, compiling C source (`.c` files) into object files (`.obj` files) is a common occurrence. Rather than requiring a statement that each `.obj` file depends on a like-named `.c` file, Make uses an inference rule to infer that dependency. The source determined by an inference rule is called the **inferred source**.

Inference rules are rules distinguished by the use of the character "%" in the dependency line. The "%" (**rule character**) is a wild card, matching zero or more characters. As an example, here is an inference rule for building `.obj` files from `.c` files:

```
%.obj : %.c
        $(CC) $(CFLAGS) -c $(.SOURCE)
```

This rule states that a `.obj` file can be built from a corresponding `.c` file with the shell line "`$(CC) $(CFLAGS) -c $(.SOURCE)`". The `.c` and `.obj` files share the same root of the file name.

When the source and target have the same file name except for their extensions, this rule can be specified in an alternative way:

```
.c.obj :
        $(CC) $(CFLAGS) -c $(.SOURCE)
```

The alternative form is compatible with Opus Make prior to this version and with other make utilities and is discussed in more detail on Page .

Make predefines the "`%.obj : %.c`" inference rule as listed above so the example we have been working on now becomes much simpler:

```
OBJS = main.obj io.obj
CC = bcc
MODEL = s
CFLAGS = -m$(MODEL)

project.exe : $(OBJS)
        tlink c0$(MODEL) $(OBJS), $(.TARGET),, c$(MODEL) /Lf:\bc\lib

$(OBJS) : incl.h
```

Response Files [Top]

For MS-DOS, OS/2 & Win95 there is a rather severe restriction on the length of a shell line with the result that the shell line is often too short for many compilers and far too short for linkers and librarians.

To overcome this restriction many programs can receive command-line input from a response file. Opus Make has two kinds of support for response files: **automatic response files**, where Make decides when to build a response file or; **inline response files**, where you write response file-creating statements directly in the makefile.

Automatic Response Files

Make has predefined support for several linkers, librarians and compilers, and you can augment Make's support by writing your own definitions (see Page). With Make's predefined support you can just add the following statement to your makefile:

```
.RESPONSE.LINK : tlink
```

This tells Make that the program `tlink` accepts LINK-style response files. When a shell line executes `tlink`, Make checks if the shell line is longer than allowed by the operating system and automatically produces a response file if necessary.

Inline Response Files

Response files can also be coded "inline" in your makefile. Here is the `tlink` shell line of the example, written to use an inline response file:

```
project.exe : $(OBJS)
        tlink @<<
c0$(MODEL) $(.SOURCES,W+\n)
$(.TARGET)

c$(MODEL) /Lf:\bc\lib
<<
```

The `tlink` program is invoked as "`tlink @`*response_file*" where *response_file* is a name generated by Make. The "`W+\n`" macro modification replaces whitespace between elements of `$(.SOURCES)` with "+enter". The *response_file* contains:

```
c0s main.obj+
io.obj
project.exe

c0 /f:\bc\lib
```

Makefile Directives [Top]

Makefile directives control the makefile lines Make reads at read time. Here is our example extended with conditional directives (`%if`, `%elif`, `%else` and `%endif`) to support both Borland and Microsoft compilers. Comments have been added for documentation:

```
# This makefile compiles the project listed in the PROJ macro
#
PROJ = project                  # the name of the project
OBJS = main.obj io.obj          # list of object files

# Configuration:
#
MODEL = s                       # memory model
CC = bcc                        # name of compiler

# Compiler-dependent section
#
%if $(CC) == bcc                # if compiler is bcc
  CFLAGS = -m$(MODEL)           # $(CFLAGS) is -ms
  LDSTART = c0$(MODEL)          # the start-up object file
  LDLIBS = c$(MODEL)            # the library
```

```
  LDFLAGS = /Lf:\bc\lib         # f:\bc\lib is library directory
%elif $(CC) == cl               # else if compiler is cl
  CFLAGS = -A$(MODEL,UC)        # $(CFLAGS) is -AS
  LDSTART =                     # no special start-up
  LDLIBS =                      # no special library
  LDFLAGS = /Lf:\c6\lib;        # f:\c6\lib is library directory
%else                           # else
% abort Unsupported CC==$(CC)   # compiler is not supported
%endif                          # endif

# The project to be built
#
$(PROJ).exe : $(OBJS)
        tlink $(LDSTART) $(OBJS), $(.TARGET),, $(LDLIBS) $(LDFLAGS)

$(OBJS) : incl.h
```

The layout of this makefile is fairly traditional — macros are defined first, the primary target follows the macros and the extra dependency information is last.

This example also uses the `%abort` directive to abort Make if the makefile does not support a particular compiler. Directives can also be used at run time, to control the shell lines Make executes. For more information about directives, see Page .

Footnotes

[1]You may be accustomed to “source” meaning a file containing source code. We will say “source file” when we mean this.

[2]This was called `.IMPLICIT` in Opus Make v5.2x and you can still use that name for `.SOURCE`.

[3]This was called `.ALL` in Opus Make v5.2x and you can still use that name for `.SOURCES`.