Chapter 8 kmmake manual

The make is a program maintenance tool which has been developed for UNIX by AT&T bell laboratory. The kmmake included in this package is an improved version of the make, to which several functions are added for easy use.
This chapter describes the operating procedures for the kmmake.

8.1 Overview

The make is a command for efficient compilation of the program. A sole C program is compiled by just starting up the compiler. However, compilation of a program containing multiple source files may become complicated if a part of the files is modified.

Assuming that 30 parts of the program containing 100 source files are corrected and compiled. Normally, it cannot be remembered which parts of the files are corrected. Thus, a part of the program is not compiled correctly and old object codes are often run.

On the other hand, if a batch file is always made to compile all source files, this may cause a long compilation time and does not have the meaning to divide the program into source files.

The make is a tool to solve such problem. According to the procedures specified by the programmer, the make checks which part of the program is corrected, calculates optimal compiling procedure, and runs it. Additionally, even though the program is composed of only one file, just typing “make” will compile the program. Therefore, the makefile is always added to the program.

The kmmake is a part of the LSI C package. However, if you use the OS containing the standard make command, such as UNIX, use of this command is recommended.

Additionally, since the kmmake is a subset of the make command on UNIX, it is possible to write the makefile which is commonly used on both the LSI C and UNIX.

8.2 How to use the kmmake

To use the kmmake, it is absolutely necessary to prepare the file "makefile" containing instructions for the kmmake.
Fig. 8.1 shows an example of the makefile. The file name is "makefile".

```
#      Makefile for make program
OBJS = make.obj  object.obj  parse.obj  misc.obj
PROGRAM = make.exe
LDFLAGS =

all: $(PROGRAM)

$(PROGRAM) : $(OBJS)
       $(CC) $(LDFLAGS) $(OBJS) -o $(PROGRAM)
```

Fig. 8.1: makefile

8.2.1 Comment

A line starting with # is a comment. It is possible to write any character string in the comment. The first line in Fig. 8.1 shows a comment.

8.2.2 Macro definition

```
name = character string
```

The above shows a macro definition format. In the macro definition, a character string placed on the right of "=" is set to a name (macro name) placed on the left of "=". This makes it possible that $(macro name) refers to the macro definition later. Therefore, the macro is convenient to write the same contents several times.

Additionally, when macros referred to frequently are gathered at the beginning of the makefile even though a macro is referred to only once, this allows easy correction.

Fig. 8.1 shows macro examples, OBJS, PROGRAM, LDFLAGS, and CC. Among them, CC is not defined actually, but is defined in other places. This is explained in later parts.

8.2.3 Dependent description line

```
target file: source file 1, source file 2, …
```

A line written in the above format is called a dependent description line and shows the file dependent relationship. That is, a target file on the left of ":" depends on (is made from) source files on the right of ":".
In the dependent description line, the compilation procedure (any executable process from the command line may be put) may follow the target file. This is called a command line. The command line starts with tab. A space is not allowed at this position. A tab (0x09) must be put.

```
all: $(PROGRAM)

$(PROGRAM): $(OBJS)
```

The above lines in Fig. 8.1 show examples of the dependent description lines. No command lines exist in the first dependent description line, but the commend line below follows the dependent description line for `make.exe`, that is $(PROGRAM) on the second line.

```
  $(CC) $(LDFLAGS) $(OBJS) -o $(PROGRAM)
```

8.2.4 Inference rules

The following is obtained when the macros shown in Fig. 8.1 are developed.

```
all: make.exe

make.exe: make.obj object.obj parse.obj misc.obj
        lcc86 make.obj object.obj parse.obj misc.obj -o
        make.exe
```

In this case, how are the files, such as `make.obj` created? These may be created by compiling the C source file named `make.c`. However, this is not written.

Actually, the `kmmake` contains the function that automatically creates the dependent description line in the following format.

```
make.obj: make.c
  lcc86 -c make.c
```

Therefore, this is not written explicitly.

If commands necessary to create a specific file are not written in the `makefile`, the `kmmake` creates the dependent relationship automatically according to the inference rules.
This step is explained using `make.obj`. It is first checked that files, such as `make.c`, `make.r86`, and `make.asm` exist in the same directory.
For example, when `make.c` is found, it is then checked that the rule to create `*.obj` file from `*.c` exists in the inference rules. If this rule exists, the compilation procedure is generated from the command line according to this rule.

It is also possible that the user redefines the inference rules. This procedure is explained later.

8.2.5 Starting up

A series of make functions shown in Fig. 8.1 is described in previous sections. This section describes what will happen when starting up the `kmmake`.

When the `kmmake` is executed if the `makefile` in Fig. 8.1 exists in the current directory, the following steps are performed internally.

1. Using the inference rules, missing dependent relationships and command lines are generated. In this example, the dependent relationships and command lines to generate `make.obj`, `object.obj`, `parse.obj`, and `misc.obj` from relevant C sources are made.

2. It is checked whether or not the first target "`all`" is updated. To do so, it is checked whether or not the source file `make.exe` that appears on the right of the "`all`" dependent description line is updated.

3. To check whether or not `make.exe` is updated, it is checked whether or not `make.obj`, `object.obj`, `parse.obj`, and `misc.obj` on the left are updated.

4. To check whether or not `make.obj` is updated, the change times of `make.c` and `make.obj` are compared. If `make.obj` is older or does not exist, the specified command lines are executed. In this example, `lcc86 -c make.c` is executed.

5. In the same manner as described above, it is checked whether or not `object.obj`, `parse.obj`, or `misc.obj` is updated. If any file is old, the commands are executed.

6. The execution is returned to the dependent description line of `make.exe`. If `make.exe` does not exist or it is older than any of `make.obj`, `object.obj`, `parse.obj`, and `misc.obj`, the specified commands (`lcc86 make.obj`…) are executed.

7. The execution is returned to the dependent description line of all. Since the file all and command line do not exist, nothing is executed and the program is exited.

As described above, the kmmake always tries to make the first target in the makefile. After the process of the first target is completed, the subsequent targets are not executed. Therefore, if you wish to make multiple execution files using the makefile, you must make the target all: at the beginning, and then enumerate execution files following this target as shown below.

```
all: foo.exe  bar.exe  quux.exe
foo.exe: …
bar.exe: …
quux.exe: …
```

If all: line does not exist in this makefile, the kmmake compiles only foo.exe. and does not compile bar.exe and quux.exe.

8.3 References

This section describes the details of the kmmake not described in the previous section.

8.3.1 makefile

The makefile repeats any of the following 0 time or more.

```
<comment>
<macro definition>
<include statement>
<ifdef - endif>
<dependent description>
```

8.3.1.1 Comment

A comment starts with # and ends with LF. The kmmake disregards comments. Any character string can be written in the comment. Japanese characters (shift JIS, EUC, or ISO-2022-JP) can also be used.

8.3.1.2 Macro definition
<macro definition> has any of the following formats.

```
<name> = <value>
<name> += <value>
```

Any character string not including LF is written in the macro value. In the original make (UNIX), # shows the start of the comment. Therefore, # is not used as a macro value, but the kmmake can uses # as a macro value. For compensation, a comment cannot be placed next to the end of the macro definition line.

```
<name> = <value>
```

In the macro definition having the above format, value itself is set in the macro.

```
<name> += <value>
```

In the macro definition having the above format, a character string is set in which the value on the right follows the current value of the macro.

`foo = $(foo) bar` is not substituted for `foo += bar`.

The latter format is needed.
The macro can be used later in a format of `$(name)`. If the macro name is only one character, ( ) can be omitted.

It is also allowed to use undefined macro. At this time, it is developed to a null character string. Additionally, a value in the environmental variable is used as macro. However, if a macro definition exists in the `makefile` or at start up, this takes precedence.

8.3.1.3 Include statement
<include> takes the contents in other file into the `makefile` and has the following format.

```
include <file name>
```

Where, <file name> must be a character string excluding spaces and considered as a name of file to be included. In this character string, a macro can also be used.

The file specified by this statement is read when the include statement is encountered and processed as if the contents of the file are inserted into that location.

8.3.1.4 ifdef - endif statement

The <ifdef - endif statement> switches the process in the makefile based on the conditions and has the following format.

```
ifdef <macro name>
<then-text>
else
<else-text>
endif
```

If the macro name next to ifdef is defined, <then - text> is executed, otherwise <else-text> is executed. It is possible to omit else and <else-text>.

Desired statements can be put in <then-text> and <else-text>.
The following example shows how to use ifdef.

```
ifdef UNIX
O = .o
X =
else
ifdef DOS
O = .obj
X = .exe
else
O = .sof
X = .com
endif
endif

foo$X:   foo$O
         $(CC) foo$O -o foo$X
```

8.3.1.5 Dependent description

<dependent description> has the following format.

```
<target list> : <source list>
[<command line>]
```

In the above example, <target list> and <source list> contain more than one <name> which is divided by spaces. Names are normally file names, but others can also be used.

<command line> contains commands to be executed next to tab at starting of the line. The desired number of command lines, which is 0 line or more, can be written. However, all command lines must start with tab.

If <target list> contains multiple targets (<name>), this is interpreted to that source lists on the right of ":" are arranged for each target.

For example,

```
foo.obj  bar.obj  baz.obj : quux.h  bluh.h
```

the above is interpreted to the following.

```
foo.obj : quux.h  bluh.h
bar.obj : quux.h  bluh.h
baz.obj : quux.h  bluh.h
```

Additionally, multiple dependent descriptions can be put for the same target. At this time, source lists for that target are gathered automatically.

For example,

```
foo.obj : quux.h
foo.obj : bluh.h
```

the above is the same as the following.

```
foo.obj : quux.h  bluh.h
```

However, it is not allowed to write more than two dependent descriptions containing the command lines for the same target.

8.3.1.6 Command line

Putting the following characters in front of the command line makes it possible to specify several operations when the command is executed.

@ This does not display the command echo during execution of the command.

- This disregards the command end code.

One command line is transferred to the shell (command interpreter), and then executed. Therefore, if the command is executed after changing the current directory in OS other than MS-DOS, the following is not applicable.

```
cd subdir
cc –c prog.c
```

The above must be written as follows.

```
cd subdir; cc –c prog.c
```

The kmmake itself processes the cd command and “;” since the command interpreter in MS-DOS and Windows (COMMAND.COM or CMD.EXE) does not support “;”. Therefore, the above description can be used in these OS.
For Windows NT, the same effect can be obtained when divided by “&” instead of “;”.

8.3.2 Inference rules

As described previously, the kmmake makes the dependent information, which is short, according to the inference rules. The built-in inference rules are described in the makedef file (storage location may vary depending on OS). It is also possible that the user can define desired inference rules. This section describes how to define the inference rules.

All inference rules have the following format.

```
.<suffix1>.<suffix2>:
        <command>
```

In the above inference rule, <command> is used to make files with an extension of .<suffix2> from those with an extension of .<suffix1>.

For example,

```
.c.obj:
        $(CC) $(CFLAGS) –c $<
```

the above has the meaning shown below.

To make *.obj files (with associated names) from *.c files, $(CC) $(CFLAGS) -c *.c is executed.
$< is a special macro which is developed into the source.

8.3.2.1 .SUFFIXES

If you put only the above inference rules in the `makefile`, the inference rules do not function correctly. To make the inference rules effective, it is absolutely necessary to define the target named `.SUFFIXES`.

In `. SUFFIXES`, the list of extensions of the source files to be searched for must be defined as source.

For example, the following is written.

```
.SUFFIXES: .y .c .obj
```

`.SUFFIXES` is already set in the standard setting file `makedef`. The extension list put in the `makefile` is added to the end of the `makedef`.

If this is not preferable (standard search order is to be replaced), null definition shown below is given and new definition is continued to clear the existing values and replace them with new ones.

```
.SUFFIXES:
.SUFFIXES: .asm  .c  .obj
```

8.3.2.2 Special macros

Several special macros can be used for commands in the inference rules.

$@	This develops the macro to the target name.
$*	This develops the macro to the target name, from which the extension is deleted.
$<	This develops the macro to the source (first source if multiple sources exist).
$?	This develops the macro to the list, which is newer than the target in the source.

The following special macros are local expansion functions provided only on the `kmmake`. However, it is recommended not to use such macros when taking the compatibility of the `makefile` into consideration.

$>	This develops the macro to that the extension is deleted from $<.
$#	
$^	This develops the macro to all source lists.
$&	This develops the macro to all sources from which the extension is deleted.
$:c	This develops nothing and uses next character “c” as a delimiter. This delimiter character is put between names defined by $?, $#, or $&. The

default delimiter is a space.

$ […] A character string enclosed by [ ] ({ }) is output to the file named make.i
$ {…} (if macro exists, it is developed) in order to replace the file name. This is convenient when using commands with the MS-DOS/Windows response file.

8.3.3.3 Changing the source directory with VPATH.

In some cases, it is desired to place the targets and sources in different directories. At this time, the special macro VPATH is used to identify the directory where the sources exist.

In the VPATH, directory path names containing the sources are divided by "**:**" as shown below.

```
VPATH = .:../lib:/proj/foo/common
```

When the inference rules function, it is checked whether or not the sources corresponding to the same directory as that containing the target exist (this is not always the current directory).
If the sources are not found, the inference rules start to sequentially check whether or not the sources exist in the directories from the left of the macro VPATH, if it is defined.

The default delimiter in the macro VPATH is "**:**". If a value is set in the VPATHDELIM macro, the start character of this macro is recognized as delimiter. If you use MS-DOS/Windows in which the path name contains "**:**", it is convenient to replace this character with "**;**".

```
VPATHDELIM = ;
VPATH = .;N:\LINSRC;..\COMMON
```

8.3.3.4 Starting up the kmmake

It is possible to give the following arguments when starting up the kmmake.

```
Kmmake [option] [macro definition] [target]
```

Multiple options, macro definitions, and targets can be written. However, the option must be written prior to the macro definition and target.

8.3.3.5 kmmake options

The following options are provided.

-d	(Debug)	This displays the debug information.
-f filename	(File)	This reads the instructions to the kmmake from the file with filename instead of the makefile.
-i	(Ignore)	This ignores the end code of the command.
-k	(Kontinue)	This executes as many commands as possible even though making of targets fails. (Only targets, which do not depend on failed targets, are executed.)
-n	(Notexec)	This only displays commands to be executed, and does not execute them.
-r	(not Read)	This does not read the initial setting file makedef.
-s	(Silent)	This executes the command without display.
-t	(Touch)	This updates only the time stamp of the target without execution of commands.

8.3.3.2 Macro definition at start up
The macro definition has the following format.

```
macro name = value
```

The meaning of the above macro definition is the same as that in the makefile. Additionally, all macros defined by arguments at start up are disregarded even though they are defined in the makefile.
That is, values specified by arguments always take precedence over others.

If the macro value contains special characters of the shell (command interpreter), such as <, >, and |, it is necessary to escape them using appropriate procedures. These procedures may vary depending on the type of OS. You may refer to the manual for the shell (command interpreter) of OS for further details.
The following is accepted in UNIX.

```
kmmake CFLAGS = '-O'
```

8.3.3.3 Target
Arguments, which are not option nor macro definition, are considered as they are specified for the target. Unless otherwise specified particularly, the kmmake attempts to make the first target in the makefile. If the target is specified by the command argument, the kmmake makes it.

For example, if the file `foo.c` exists in the current directory (`makedef` is set appropriately),

```
lcc86 foo.c -o foo.exe
```

is executed by

`kmmake foo.exe`.

At this time, it is accepted even though the `makefile` does not exist. (Of course, if the program contains multiple sources, the `makefile` is absolutely necessary.)

Multiple targets can be specified. In this case, targets are executed (made) from those specified earlier in sequential order.