

## 1. Introduction

This Application Note is issued to have a clear understanding of the contents of the makefile, generated for your project by EDE, and executed by the make utility mk51. It also gives you information on how to set up your own makefile.

## 2. Basic Operation and Invocation

Almost any application can be managed using one or more batch files to do the work of compiling, assembling, linking and locating. However, the use of a makefile can speed up the process, and often save a lot of errors. This is because the make process has a method for identifying files which have changed since the last time the application is built. This saves time, because only those files which are modified will be rebuilt during the make process. By the same token, the make process will rebuild all of the files which need to be rebuilt, and not only those files which we remembered had changed since the last time the application is built. This can prevent errors. In addition, the makefile can contain instructions for cleaning up intermediate files which have accumulated in the process of creating the end product.

The make process is steered by two makefiles, which contain all the necessary rules to build your project. One is the working makefile, which resides in your project directory. The other one represents the default-rules file and is called mk51.mk. This default-rules file can be found in the product's etc subdirectory. More on this later.

EDE will generate a file called project.mak, where project is referring to the actual name of your EDE project. To execute mk51, the `-f` option is used to specify the name of the makefile:

```
mk51 -f project.mak
```

The make utility mk51 may also be invoked without specifying the name of the makefile explicitly. For this, the name of the makefile must be "makefile" (without the extension .mak).

```
mk51    (is the same as: mk51 -f makefile)
```

For more information about all possible invocation options, please refer section 11.6 of the assembler manual named "MK51".

## 3. Filename extensions and suffixes

The make process is greatly simplified when the standard filename extensions are used which are recognized by the make utility. The default extensions that will be recognized by mk51 are:

.c					
.asm	.src	.obj			
.out	.abs	.hex	.sre	.o51	

The implicit make rules for these extensions are listed in the default-rules file called etc/mk51.mk.

The makefile element `.SUFFIXES:` (a so called "special target") defines a list of filename extensions used for selecting these implicit rules. Specifying this suffixes target with one or more filename extensions in your own makefile, adds these to the end of the already existing suffixes list. For example:

.SUFFIXES: .51 .sym

Any file whose name ends in one of these filename extensions will be handled by a default rule which is established in the default-rules file etc\mk51.mk. Files with other extensions can be handled by specific rules which need to be stipulated inside the working makefile from the project directory. For this reason, most often it is easier to use the filename extensions which are already listed in the default-rules file. See also the 'implicit rules' paragraph of section 11.6 of the assembler manual.

**Note:**

Specifying the suffixes target without any extension, will clear the suffixes list.

**4.1 Macros for Text Substitution**

Within a make file or default-rules file, text substitution macros can be defined in order to enhance readability of the makefile. It also helps to eliminate typographical errors and clutter within the file. An example of a macro definition is:

```
CCFLAGS = -MI -O1
```

A macro name is usually capatilized, in order to easily distinguish it from other makefile elements. The text which is substituted for the macro name is accessed within the makefile by enclosing it in parentheses and preceding it with a dollar sign. If a macro name is re-defined, it will expand to its last definition. For example, to expand the CCFLAGS macro, write:

```
$(CCFLAGS)
```

Macros defined in the default-rules file are also valid for the working makefile. However, a macro defined in the default-rules file, as follows:

```
CCFLAGS =
```

will be treated as a NULL string, unless it is redefined in the working makefile, whereupon it will then expand in the default-rules file to the new string assigned to it. This priciple allows the makefile to pass details (options etc.) to the default-rules file.

**4.2 Built-In Macros**

There are several built-in macros to represent objects on the target-dependency line (see paragraph 5):

\$*	The basename of the current target.
\$<	The name of the current dependency file.
\$@	The name of the current target.
\$?	The names of dependents which are younger than the target.
\$!	The names of all dependents.
\$	This macro translates to a dollar sign. Thus you can use "\$\$" in the makefile to represent a single "\$".

**Note:**

The \$< and \$\* macros are normally used for implicit rules. They may be unreliable when used within explicit target command lines. All macros may be suffixed with F to specify the Filename components (e.g. \${\*F}, \${@F}). Likewise, the macros \$\*, \$< and \$@ may be suffixed by D to specify the directory component.

**5.1 Targets and Dependencies**

A target is the name assigned to the result we want to achieve with the makefile. Sometimes this name identifies a file which must be created or updated, and sometimes the target is the name used to identify several targets which must be satisfied. Dependencies are those files which need to be tested to determine whether mk51 will create a newer target or not.

A target entry in the makefile has the following format:

```
target ... : [dependency ...] [; rule]
    [rule]
    ...
```

Any line which does not have leading white space (other than macro definitions) is a 'target' line. Target lines consist of one or more filenames (or macros which expand into same) called targets, followed by a colon (:). The ':' is followed by a list of dependent files. The dependency list may be terminated with a semicolon (;) which may be followed by a rule or shell command.

If a target is named in more than one target line, the dependencies are added to form the target's complete dependency list. The modification time of the target is compared to the modification times of each dependent file. If the target is older, one or more of the dependents have changed, so the target must be constructed. Of course, this checking is done recursively, so that all dependents of dependents of dependents of ... are up-to-date.

For example:

```
main.src:  main.c def.h
```

The target, the file named main.obj, will need to be rebuilt if either main.c and/or the include file def.h has changed since the last Make is executed.

Special allowance is made on MS-DOS for the colons which are needed to specify files on other drives, so for example, the following will work as intended:

```
c:foo.src : a:foo.c
```

**5.2 Rules and Shell Lines**

A line in a makefile that starts with a TAB or SPACE is a shell line or rule. This line is associated with the most recently preceding dependency line. A sequence of these may be associated with a single dependency line. When a target is out of date with respect to a dependent, the sequence of commands is executed. For example:

```
main.src:  main.c def.h
    cc51 main.c  -M1 -O1
```

In this example, all the elements are there to build main.src after identifying it needs to be updated. Please note that it is necessary to provide blank lines before and after a complete makefile element like this one.

If we combine the built-in macros and user defined macros to create the rule, it could look something like this:

```
CC          = CC51
MODEL       = -M1
CCFLAGS    = $(MODEL) -O1

main.src:   main.c def.h
            $(CC) main.c $(CCFLAGS)
```

Shell lines may have any combination of the following characters to the left of the command:

- @ will not echo the command line, except if -n is used.
- mk51 will ignore the exit code of the command, i.e. the ERRORLEVEL of MS-DOS. Without this, mk51 terminates when a non-zero exit code is returned.
- + mk51 will use a shell or COMMAND.COM to execute the command.

If the '+' is not attached to a shell line, but the command is a DOS command or if redirection is used (<, |, >), the shell line is passed to COMMAND.COM anyway. For UNIX, redirection, backquote ( ` ) parentheses and variables force the use of a shell.

mk51 can generate inline temporary files. If a line contains '<<WORD' then all subsequent lines up to a line starting with WORD, are placed in a temporary file. Next, '<<WORD' is replaced by the name of the temporary file.

Example:

```
link51 -f <<EOF
$(separate ",\n" $(match .obj $!)),
$(separate ",\n" $(match .lib $!))
to $@
$(LDFLAGS)
EOF
```

The four lines between the tags (EOF) are written to a temporary file (e.g. \tmp\mk2), and the command line is rewritten as link51 -f \tmp\mk2.

## 6.1 Built-In Functions

A function not only expands but also performs a certain operation. Functions syntactically look like macros but have embedded spaces in the macro name, e.g. '\$(match arg1 arg2 arg3)'. All functions are built-in and currently there are five of them: match, separate, protect, exist and nexist.

### match

The match function selects all arguments which match a certain filename extension:

```
$(match .obj prog.obj sub.obj mylib.lib)

will yield:

prog.obj sub.obj
```

Notice that the function rejects the non-matching mylib.lib in the argument list.

### separate

The separate function joins its arguments using a text string to define the separating characters. If the first argument is enclosed in double quotes then '\n' is interpreted as a newline character (CR-LF), '\t' is interpreted as a tab, '\nnn' is interpreted as an octal value (where nnn is an octal number) and spaces are taken literally. For example:

```
$(separate ",&\n" prog.obj sub.obj)

will result in:

prog.obj, &
sub.obj
```

**Note:** The ampersand character '&' is make's line continuation character.

Function arguments can also utilize the results provided by other built-in functions, like:

```
$(separate ",&\n" $(match .obj $!))

will result in:

file1.obj, &
file2.obj, &
file3.obj, &
etc.
```

The match function in this line will deliver all of the object files in the dependency list (\$!) to the separate function, which will separate each \*.obj file with comma, space, ampersand and new-line characters.

**protect**

The protect function adds one level of quoting. This function has one argument which can contain white space. If the argument contains any white space, single quotes, double quotes, or backslashes, it is enclosed in double quotes. In addition, any double quote or backslash is escaped with a backslash. Example:

```
echo $(protect I'll show you the "protect" function)

will yield

echo "I'll show you the \"protect\" function"
```

**exist**

The exist function expands to its second argument if the first argument is an existing file or directory. Example:

```
$(exist test.c cc51 test.c)

When the file test.c exists it will yield:

cc51 test.c
```

When the file test.c does not exist nothing is expanded.

**nexist**

The nexist function is the opposite of the exist function. It expands to its second argument if the first argument is not an existing file or directory. Example:

```
$(nexist test.src cc51 test.c)

When the file test.src does not exists it will yield:

cc51 test.c
```

**6.2 Conditional Processing**

Lines containing ifdef, ifndef, else or endif are used for conditional processing of the makefile. They are used in the following way:

- ifdef macroname
- if-lines
- else
- else-lines
- endif

The if-lines and else-lines may contain any number of lines or text of any kind, even other ifdef, ifndef, else and endif lines, or no lines at all. The else line may be omitted, along with the else-lines following it.

First the macroname after the if command is checked for definition. If the macro is defined then the if-lines are interpreted and the else-lines are discarded (if present). Otherwise the if-lines are discarded; and if there is an else line, the else-lines are interpreted; but if there is no else line, then no lines are interpreted. When using the ifndef line instead of ifndef, the macro is tested for not being defined. These conditional lines can be nested up to 6 levels deep.

## 7. Implicit Rules

Implicit rules are closely tied to the special target “.SUFFIXES:”. Each entry in the .SUFFIXES: list defines an extension to a filename which may be used to build another file with the same basename.

If a file (target) doesn’t have a specific *rule* associated with it on the next line, each entry in the .SUFFIXES: list is combined with the extension of the target, to get the name of an implicit target. If such a target exists, the implicit rules from the default-rules makefile (mk51.mk) are invoked to create the target. The following makefile uses implicit rules (from mk51.mk) to perform the job:

```
prog.out:      prog.obj sub.obj
prog.src:      prog.c inc.h
sub.src:       sub.c inc.h
```

Similar to that, if a file doesn’t have a specific *target* line, each entry in the .SUFFIXES: list is combined with the extension of the target, to get the name of an implicit target. If this target exists, the implicit rules from the default-rules makefile are invoked to create the target. Any dependents of the implicit target are ignored.

An implicit target is represented in the default rules file like this: **.c.src:**

The full description of a default rule to make an \*.src file from a dependent \*.c file could look like this:

```
.c.src:
$(CC) $(CCFLAGS) $(CPPFLAGS) $<
CC = cc51
CCFLAGS =
CPPFLAGS =
```

The built-in macro \$< represents the name of the current dependency file (\*.c file). Although macros CCFLAGS and CPPFLAGS are empty, any substitutions that are assigned to them in the working makefile, will be used.

## 8.1 Special Targets

**.DEFAULT:** The rule for this target is used to process a target when there is no other entry for it, and no implicit rule for building it. mk51 ignores all dependencies for this target.

**.DONE:**

This target and its dependencies are processed after all other targets are built.

**.IGNORE:**

Non-zero error codes returned from commands are ignored. Encountering this in a makefile is the same as specifying -i on the command line.

**.INIT:**

This target and its dependencies are processed before any other targets are processed.

**.SILENT:**

Commands are not echoed before executing them. Encountering this in a makefile is the same as specifying `-s` on the command line.

**.SUFFIXES:**

The suffixes list for selecting implicit rules. Specifying this target with dependents adds these to the end of the suffixes list. Specifying it with no dependents clears the list.

**.PRECIOUS:**

Dependency files mentioned for this target are not removed. Normally, `mk51` removes a target file if a command in its construction rule returned an error or when target construction is interrupted.

**clean:**

This target is used to invoke the DOS *delete* command on its arguments. In the default-rules file `mk51.mk`, the macro `RM` is used to substitute the `'del'` command. The `clean` target causes `make` to delete all files listed and to exit the `make` process. For example:

```
clean:
*.map
*.lst
*.obj
```

If your makefile is named `makefile`, you can invoke `make` on this file exclusively with: **`mk51 clean`**. This will remove all files which are listed after the special target `clean` in the makefile. If you execute `mk51` after this without `clean`, it will rebuild all the deleted files. This process can be performed at once, by using the next invocation: **`mk51 clean all`**. This will force `mk51` to remove the listed files, and rebuild them from scratch.

## 8.2 Special Macros

**MAKE**

This normally has the value `mk51`. Any line which invokes `MAKE` temporarily overrides the `-n` option, just for the duration of the one line. This allows nested invocations of `MAKE` to be tested with the `-n` option.

**MAKEFLAGS**

This macro has the set of options provided to `mk51` as its value. If this is set as an environment variable, the set of options is processed before any command line options. This macro may be explicitly passed to nested `mk51`'s, but it is also available to these invocations as an environment variable. The `-f` and `-d` flags are not recorded in this macro.

### PRODDIR

This macro expands the name of the directory where mk51 is installed without the last path component. The resulting directory name will be the root directory of the installed compiler package, unless of course if mk51 is installed somewhere else. This macro can be used to refer to files belonging to the product, for example a library source file.

Example:

```
START = $(PRODDIR)/lib/src/cstart.asm
```

When mk51 is installed in the directory /cc51/bin this line expands to:

```
START = /cc51/lib/src/cstart.asm
```

### SHELLCMD

This contains the default list of commands which are local to the SHELL. If a rule is an invocation of one of these commands, a SHELL is automatically spawned to handle it.