`Msub` 1.2 manual

by Anders Munch

Last revised May 18, 1995

Contents

1 Introduction

The basic functionality of `Msub` is the same as your text editor's search-replace function. Of course it is unneededly complicated to quit your editor, run `Msub` and restart your editor, in order to do a simple search-replace. So why `Msub`? This is why:

- Multi-file operation. For when you need to change the same thing(s) in several files.
- Speed. `Msub` is several times faster than an editor.
- Expressive power. Regular expressions and context sensitivity allow you to pinpoint exactly what you are searching for.
- Versatility. You can do extensive text manipulation with `Msub` that would otherwise require you to write a program in C/BASIC/Awk/Perl. You can do such things as reformat records in an ascii file, reorder fields, change field separator, remove or duplicate parts of the text.
- Scripting. Changes that you do often could be automated with an `Msub` script. E.g. you could have a script to correct common spelling errors or typos, to replace abbreviations with the full form, just to name a few.
- Safety. Sometimes a search-replace operation changes something that you didn't expect to change nor wanted to. Few editors have undo commands that will undo a global substitution. `Msub` does.

Before you spend too much time reading this manual, consider registering first. That will get you the much nicer looking and more readable original (LaTeX) document. See about registering in section 11 below and in the accompanying file `REGISTER.DOC`.

2 How To Avoid Reading This Manual

If you know about script files and regular expressions, take a look at the example scripts `INTRO??.MS` and in no time you will be writing your own `Msub` scripts and getting the job done.

You still need to read specific sections of this manual, though:

- Section 4.1 and 4.2 on how to run `Msub`.
- Section 4.4 on how `Msub` regular expressions differ from the regular expressions you met elsewhere.

- New features, that are not (yet) covered by tutorial script, including abbreviations in 5.3 and file search directories 7.8.

When you have found out how useful `Msub` can be in your daily work, then register to get the printed manual and read the rest of it.

3 How To Read This Manual

Sections 4–8.4 below is the complete reference manual as well as an instructive tutorial with lots of examples.

Read it from linearly from top to bottom, or use it as a reference, skipping to specific sections of interest. It is intended for both.

If you are new to regular expressions you can read the introduction to regular expressions in section 10 below.

Be sure to look at the sample `Msub` scripts in the `EXAMPLES` subdirectory. (The `.MS` files.)

4 The Basics

4.1 Invoking `Msub`

`Msub` can be invoked in two ways. With a script file:

```
msub [-ask] [options] scriptfile [options] files...
```

– or what to search for can be given on the command line:

```
msub [-ask] -search=sth -replace=sth [options] files...
```

If the `files` are omitted, the scriptfile is just checked for syntactic correctness. You can use that to get the error and warning messages for a script without actually changing any data.

If `files` are a "`-`" character then `Msub` works as a filter: Input is read from the standard input and the modified output is written to the standard output.

The `-search/-replace` variant allows you to replace one thing only, and is basically provided as a shortcut for when you don't want to bother with creating a file. As a third possibility you can enter a script file directly from the keyboard, by using "`-`" as the script file name. `Msub` will then read the script file from the standard input, until you end it by typing `!end` or CTRL-Z on a line by itself. Quoting with ' and " is a little tricky on the command line, see section 9 for details.

`-search` and `-replace` are actually the options `searchfor` and `replacewith`, and everything that applies to options in general apply to them too: You can abbreviate them, use a space instead of "=", use "/" instead of "-" and rearrange the order:

```
msub [options] /repl sth /searchfor sth files...
```

A little terminology: *text files* or *input text* refer to the files you want something changed in. They are the ones simply termed "`files`" in the command lines above. The *input text* refers to the unchanged original contents of these files. (This is slightly misleading, as the *text files* may be binary files, and script files are text files, too.)

4.2 Query replace

By default `Msub` makes all the requested changes without asking. And if something is changed that shouldn't have been, you can recover the backup file.

The `-ask` option tells `Msub` to prompt you before a change. If you answer '`y`' for 'yes', this occurrence is changed. Answer '`n`' for 'no' and it is left unchanged.

Table 1 is a table over the replies and their meaning.

If you type a '`a`' for 'always' before the '`y`' or '`n`', then your answer will apply to *all* following changes, and `Msub` will stop prompting you completely (if you are

Figure 1: Query replies

You type	Msub does
'y'	Changes this occurrence
'n'	Skips this occurrence
'c'	Cancel (entire 'f'ile or 'a'll files)
'i'	Input an alternative replacement text (not implemented)
'f'	Prefix: All this file answer...
'a'	Prefix: Always, all files answer...
't'	Prefix: For this search-replace pair answer...

changing several files, then this will also apply to any subsequent files). I.e. after 'ay' all the rest of the changes will be done quietly, and after 'an' nothing more will be changed.

Typing 'f' for 'file' is similar to 'a', except that it only applies to the current input file.

Typing 't' for 'this' is also similar to 'a', but applies to this search-replace pair only. That is, every time something matches that very same search expression, then the answer ('y' or 'n') you give after 't' will apply, but no other search-replace pairs are affected.

A simple 'c' for 'cancel' will leave this change undone, just like 'n'. But 'c' is different when used in conjunction with 'a' or 'f': 'fc' will cancel all changes to the current file, even those that you have already answered 'y' to. And 'ac' will cancel not only the current file, but also any following files. That is, 'ac' will effectively abort Msub execution. Alas, this doesn't cancel any files *previous* to the current. If you want all your changes undone, it might be easier to answer 'an' and then recover the backup files using UNBAK (see section 7.6).

Hopefully, when you try this the prompt will be discretely colorized. If instead the text is riddled with strange codes, then you need to install the ANSI.SYS driver which comes with DOS. Add something like this to your CONFIG.SYS:

```
DEVICE=C:\DOS\ANSI.SYS
```

Finally there is a shell escape option (that doesn't show on the menu). Type '!' to shell to DOS, and then 'exit' to return to Msub. While shelling, you must not edit the current file, but any other input files and even current script file are fair game (script file changes here won't affect the current run).

4.3 Script file format

Script files are read a line at a time. Leading and trailing spaces are ignored. Lines that end with a "\" character are continued onto the next line.

Search-replace pairs

The basic script file format is very simple: Put the text you want to search for on one line (the *search expression*), and the text you want to replace it with on the next line (the *replacement text*). I call two such lines a *search-replace pair*.

If you want to search for more things, just write another search-replace pair, and put a blank line between them.

Options and directives

A line that begins with a "`!`" character is an option or directive. The options available in the script are the same as those that are available at the command line—except a few can be used only at the command line (`-search` and `-replace`) or only in a script (e.g. `!include`).

Options may simply be a single word. To avoid unnecessary messages:

```
!quiet
```

They may be qualified by a selector. To ignore case in search text:

```
!case=ignore
```

They may have string arguments. To read another script file:

```
!include=somefile.ext
```

They may have numeric arguments. To re-run the script 10 times:

```
!iterate=10
```

Instead of "=", you may use a space:

```
!case ignore
```

Option names may be abbreviated as much or as little as you like, as long as they do not become ambiguous. The first two characters of the option are always enough to avoid ambiguity. The same goes for selectors, they abbreviate all the way down to just the first character. The examples above abbreviated as much as possible:

```
!q

!ca=i

!it=10

!in=somefile.ext
```

Comments

Lines that begin with "#" are comment lines and are ignored. The comment must be on a line by itself (not following any option, regular expression or the alike).

4.4 Regular expression format

In this section I assume some familiarity with regular expressions, as they are found e.g. in the `grep` program. If you don't know regular expressions, there is an introduction in section 10 below.

`Msub` supports all the most common regular expression constructs:

expr`*` Repeat *expr* 0 or more times. (*closure*)

expr`+` Repeat *expr* 1 or more times. (*positive closure*)

expr? *expr* is optional (repeat 0 or 1 times)

$exp_1|exp_2|\ldots|exp_n$ Match any one of exp_{1-n}. (*union*)

(*expr*) Parentheses may be used freely.

$[c_1c_2\ldots c_n]$ Set: Matches any of the characters c_1–c_n. (*union*)

$[\ldots c_1 - c_2 \ldots]$ Set: Matches any of the characters which lie between c_1 and c_2 in ascii, including c_1 and c_2. E.g. `[A-Za-z]` matches any Latin letter.

[ˆ ...] Set: Matches any characters except those in the set. (To match any single character use `[^]`, but be ware: `[^]*` is 'dangerous'.)

digits The character with *decimal* ascii code *digits*. (C hackers read my lips: *No more octal.*)

c A literal character.

$expr_1$ $expr_2$ One thing after the other, $expr_1$ followed by $expr_2$. (*product*)

"..." or '...' Literal text: Otherwise special characters such as `*+?\` have no special meaning within quotes. To produce a literal ' character, enclose it within " quotes, and to produce a literal " character, enclose it within ' quotes.

$ The end-of-line character (CARRIAGE-RETURN/CR)

ˆ Then beginning-of-line character (LINEFEED/LF)

. Matches any single character except CR and LF (i.e. same as `[^$^]`).

This may look very familiar, but there are some points where the regular expressions of `Msub` differ from those of the unix `regex` package. These differences are mostly to favor safety: Remember, unlike `grep`, this program will modify your data.

The differences are:

Line separators $ and ^ are actual characters. They are simply shorthand for CR and LF, ascii 13 and ascii 10. You can use them in the middle of a regular expression, thus searching across line boundaries. A regular expression in `Msub` can easily match an entire file.

To match a plain dollar sign or caret, $ or ^ need quoting (unlike other programs where only a leading ^ and a trailing $ have special meaning).

Quoting Quoting, i.e. depriving special characters of any meta meaning, can be done with ' and ", as well as the traditional \ escape. Quoting works the same inside sets as outside. You are discouraged (warnings) from using any special characters unquoted, even if they don't have any meta-meaning. (Yet, that is! `Msub` syntax may be extended in the future.)

Note that `"Jill"*` repeats the whole word, same as `("Jill")*`; and unlike `Jill*`, where only the trailing "l" is repeated.

Ranges To get a "–" character in a set, you must quote it. Placing it next to the "[" or "]" doesn't work. The same goes for "^" and "$".

5 Beefing up the search text

5.1 Parallel and sequential substitution

When you write a series of search-replace pairs in a script, they are executed in parallel. This means that the inserted replacement text of one replacement is not scanned for further changes. Example:

```
"W. Henderson"
"William Henderson"

"William"
"Bill"
```

This will *not* change "W. Henderson" into "Bill Henderson".

If you want search-replace pairs to execute in sequence instead of in parallel, such that later search-replace pairs work on the result of the previous search-replace pairs, use the `!newpass` directive:

```
"W."
"William"

!newpass

"William"
"Bill"
```

This script has two *passes*. In the first pass, all "W."s are changed to "William", and the second pass "William" is changed to "Bill". This script *will* change "W." to "Bill".

Each pass may contain multiple search-replace pairs, that are replaced in parallel.

5.2 Letter case

All text is case sensitive by default. This can be changed with the `case` option. The option

```
!case=ignore
```

will cause `Msub` to ignore case in the search expressions. When you write `"and"` it will be interpreted the same as `[Aa][Nn][Dd]`, thus ignoring case. This only happens to plain, literal letters. Letters that are specified with a "\" escape sequence will still be case significant. E.g. `\65"nd"` will only match the word *And* if the "A" is uppercase. Also, case is always significant in sets. Another way to make sure the "A" is capital would be to write `[A]nd`.

5.3 Abbreviations

Common regular expressions have abbreviations. These abbreviations consist of a ':' followed by a single, lowercase letter, making up to 26 different abbreviations possible. E.g. `:d` is the abbreviation of a digit, as it has the value `[0-9]`. Anywhere you would have written `[0-9]`, you may just write `:d` instead.

The good part is that you can define your own abbreviations. Definitions are on the form "!:*c*=*definition text*". This makes ":*c*" short for "*definition text*". The definition text may itself contain other abbreviations. E.g. you could define :i to match a positive integer number with

```
!:i=:d+
```

(Read: An integer is one or more decimal digits.)

Some abbreviations come predefined. Here is a list:

<table>
<tr><th>Abbrev</th><th>Description</th><th>Definition</th></tr>
<tr><td>:d</td><td>Single digit</td><td>[0-9]</td></tr>
<tr><td>:i</td><td>Integer</td><td>:d+</td></tr>
<tr><td>:n</td><td>Number</td><td>('+'|'-')?:i('.':i)?((e|E)('+'|'-')?:i)?</td></tr>
<tr><td>:l</td><td>Letter</td><td>[a-zA-Z\128-\255_]</td></tr>
<tr><td>:t</td><td>Text (letters)</td><td>:l+</td></tr>
<tr><td>:w</td><td>Whitespace</td><td>[\9$^\11\12]+</td></tr>
<tr><td>:b</td><td>to Begin-line</td><td>^.*</td></tr>
<tr><td>:e</td><td>to End-line</td><td>.*$</td></tr>
<tr><td>:a</td><td>begin-file</td><td>\26^</td></tr>
<tr><td>:z</td><td>end-file</td><td>$\26</td></tr>
</table>

You may change (override) any of these definitions. Note that if you, say, changed the definition of `:d`, that would also effect the definition of `:i`, because the `:i` definition refers to `:d`. E.g. you could define

```
!:d=[0-9a-fA-F]
```

and `:i`, defined as `:d+`, would suddenly match hexadecimal numbers.

Abbreviations can be used within sets; i.e. if you had `!:d=0-9` then you could write a hexadecimal digit as `[:da-fA-F]`. If the definition itself is a set, it is treated specially: The set brackets are ignored. This means that `!:d=[0-9]` will work just as well with the hexadecimal digit expression above. This makes it easy to negate an abbreviation: `[^:d]` means any non-digit character. In a definition such as the definition for `:w`, the trailing operator (`*`, `+` or `?`, in this case `+`) is also ignored, and `[^:w]` is any non-whitespace character.

5.4 Context sensitive expressions

Suppose you want to replace "he" with "she", but you don't want to change the "he" if it is part of a word, such as "when" or "heart". Then you could write:

```
[^A-Za-z]/he/[^A-Za-z]
she
```

The parts before the first "/" and after the last "/" describe the *context*. `Msub` searches for the entire expression, but sets marks to remember where the "/" positions were. Only the part between the two "/" is replaced.

Even if you don't specify both preceding and trailing context, you still must write both "/" markers; even if one of the context parts is empty.

If the context is the same for all search-replace pairs, then you could specify it once and for all:

```
!precontext=[^A-Za-z]

!postcontext=[^A-Za-z]

he
she
```

This search-replace pair, and any others following it, will have the `!precontext` as preceding context and the `!postcontext` as trailing context. So this search-replace pair is actually identical to the one above with explicit context.

5.5 Equality test

Regular expressions are powerful, but not always powerful enough. The `\equal` condition is there to remedy that a little, by providing a means to compare two strings for equality.

Conditions are written just to the right of the search expression, without any intervening whitespace. Begin with a backslash, write the condition name *equal* and then the two arguments each enclosed in parentheses. Each argument may contain the same things that the replacement text consists of. Example:

```
~1:l+~1[:w]+~2:l+~2\equal(~1)(~2)
~1" (note: duplicate removed)"
```

This expression matches two words on the condition that the marked regions (ie. the two words) are identical. If they are, then everything proceeds as usual, and the net effect will be removing the duplicate and making a note of it. If the words are not identical, then the expression doesn't match.

The comparison is case sensitive. If you want case insensitive comparison, combine with the toupper or tolower function, like this:

```
\equal(\toupper(~1))(\toupper(~2))
```

Note that if the longest match for the regular expression doesn't satisfy the equality condition, then a shorter match may be chosen that does satisfy the condition. If the above example is run on "an anecdote", then although the entire string doesn't satisfy the condition, the prefix "an an" does, will match, and be replaced.

6 Beefing up the replacement text

The replacement text is not just a literal string. If you need it to be a literal string you can quote it, just as with the search expressions, using `'single quotes'` and `"double quotes"`. And the *ascii* form also works here, so you can use `\12` to produce the form-feed character.

6.1 Selections

You can in the replacement text refer to parts of the search text. Use the back-quote character ‘ to refer to the entire text that is being replaced:

```
(Jones|Jackson|Johnson)
"Mr. "`
```

This will effectively prefix these names with “Mr. ”. There may be more than one backquote character, thus duplicating the found text:

```
Bond
`". James "`"."
```

You can mark a selection of the search text (using that same backquote character), to indicate that only part of it should be copied to the replacement text:

```
My name is `Bond`
My name is `"." James `
```

More than one part of the search text can be copied from, using the additional markers `~0`,`~1`,...,`~9`:

```
My name is ~1James~1 ~2Bond~2"."
My name is ~2"." ~1 ~2"."
```

The examples above don't achieve much, as the search text selection is a piece of literal text. It is when the selection is some kind of wildcard, that these constructs are useful:

```
My name is ~1[A-Za-z]+~1 ~2[A-Za-z]+~2"."
My name is ~2"." ~1 ~2"."
```

Marks may not appear within parenthesis. Neither ~0–~9, backquote (‘) or the context marker “/” may appear within parenthesis. This is *not* legal:

```
(Sean ~1Connery~1|Roger ~1Moore~1|Timothy ~1Dalton~1)
James Bond '('~1')'
```

6.2 Converting letter case

Conversion to upper or lower case can be done by writing `\toupper(...)` or `\tolower(...)` in the replacement text. Instead of "...", put whatever it is that you want to convert to upper or lower case. Example:

```
"I am from "~1:l+~1
\toupper(~1)" is where I come from"
```

On the input "I am from Denmark" this will produce "DENMARK is where I come from".

For another example, see the PASBEAU.MS script, which (among other things) converts Pascal keywords to lower case.

7 Files

7.1 Extracting

There is a directive you can use to extract specified parts of your input text that you want to keep and throw everything else away:

```
!clip
```

This directive tells `Msub` to keep only the replacement text. This means that anything that is *not* matched by a search expression is thrown away.

You can use this in conjunction with the `!screen` option to create a `grep`-like effect. To print all `device` statements in your config.sys:

```
msub -clip -scr -case=i -se=^device.*$ -re=' \config.sys
```

7.2 Binary and ascii files

By default `Msub` considers input files to be ascii files, as specified by the option

```
!text
```

(Used to be named `!ascii`.) In ascii mode, before any substitutions take place, input is converted to an internal format. In the internal format it is easier to manage beginning and ending of lines, and it also makes it possible to search to or from the beginning or end of the entire file.

You can specify binary mode with

```
!binary
```

In binary mode, no such conversions take place.

The internal format looks like this:

```
CTRL-Z LF ...first line... CR
LF ...intermediate lines... CR
LF ...last line... CR CTRL-Z
```

7.3 Line separators

All lines begin with a line feed character, which you can search for using "^", and ends with a carriage return character, which you can search for using "$". To search across lines, you'll need both: `First line$^Following line`.

Line separator options:

```
!lineseparator=binary
```

No conversions take place.

```
!lineseparator=crlfwrite
```

The default. File is converted to internal format.

```
!lineseparator=lfwrite
```

File is converted to internal format, and at the end of processing, CR/LF (DOS style) line separators are changed to just LF (UNIX style).

```
!lineseparator=default
```

Same as `!lineseparator=crlfwrite`. In a UNIX version of `Msub`, this would be the same as `!lineseparator=lfwrite`.

7.4 Ctrl-Z

The file begins and ends with the DOS eof marker, CTRL-Z. This means you can search for something at the beginning of the file with `\26^First line text`. Or at the end of file with `Last line text$\26`.

The extraneous CTRL-Z characters are removed after your substitutions have taken place, and if the original file ended with CTRL-Z (within the last 10 characters) that ending is restored.

There are four different options for handling CTRL-Z:

```
!ctrlz=copy
```

The default, as described above.

```
!ctrlz=ignore
```

CTRL-Z is treated like any other character, and nothing is added at the beginning or end of file.

```
!ctrlz=remove
```

As `copy`, except doesn't restore original ending.

```
!ctrlz=add
```

As `copy`, except instead of restoring the original ending, a standard CTRL-Z file ending is added, which consists of a single CTRL-Z character on a line by itself.

7.5 Backup

Msub creates backup files. The backup files are named with a different extension from the original, and are placed in the same directory as the original.

There is no fixed backup file extension such as .BAK. (This is to avoid that files such as README.DOC and README.1ST get the same backup file name README.BAK.) Instead the backup file extension is created from the original extension by inserting a "~" character as the first character. E.g. SRCFILE.C is backed up to SRCFILE.~C. If the original extension is a full three letter extension, the middle character will be omitted. E.g. AUTOEXEC.BAT is backed up to AUTOEXEC.~BT.

This may seem confusing, but really all you need to know is that

```
del *.~*
```

will delete all backup files, and use the UNBAK utility (see below) to restore backup files.

You have an option to use conventional .BAK backup extension:

```
!backup=bak
```

You can turn off backup file creation completely:

```
!backup=no
```

7.6 Restoring backup files

Backup files can of course be restored manually using dos commands such as del, rename and copy.

However the UNBAK utility provides a much more convenient way to do this.

Suppose you just did this:

```
msub myscript file1.txt *.doc
```

If you want this undone, simply go

```
unbak file1.txt *.doc
```

The UNBAK program will, for each file named on the command line, locate the corresponding backup file, and swap these two files. So your backed up data is restored under the proper name, and the modified data is still there, only under the backup file name.

To undo an UNBAK command, just do it again with the same arguments, and your files will be swapped right back where they came from.

The UNBAK program knows about both Msub tilde-style backups as well as conventional .BAK backup files. If there is both a tilde-style backup file and a .BAK backup file, UNBAK doesn't choose between them but instead prints a diagnostic.

7.7 Options for updating

If you don't want to actually change a file but need to look at the changed contents, use

```
!update=screen
```

or simply

```
!screen
```

This will prevent any files from being changed, and instead print the changed text on the standard output. You could redirect this output if you wanted to, e.g. like this:

```
msub -screen myscript.ms original.fil > changed.fil
```

For safety, `Msub` never deletes the original file, until a new version has been written to disk. If you don't have enough space on your harddrive for this, you can (besides setting `!backup=no`) instruct `Msub` to overwrite the original directly, using

```
!update=overwrite
```

Finally, you can prevent `Msub` from changing any files at all:

```
!update=no
```

You probably never want to do this, as it doesn't seem to achieve much. (I use it for testing.)

7.8 Script file search directories

`Msub` searches for script files (that are named without a path component e.g. `MYSCRIPT.MS` but not `C:\MSSCRIPTS\MYSCRIPT`) in the following places:

- The current directory.
- The directory in which the `Msub` program executable (`MSUB.EXE`) resides.
- Each directory in the DOS search path, i.e. the `PATH` environment variable.
- An `!include`d file is also searched for in the directory of the script that it was included from. (include files are described in section 8.4 below)

8 Script structuring options

8.1 Iteration

The directive

```
!iterate
```

tells `Msub` to repeat the script. When the substitutions in the script have been performed, `Msub` restarts from the beginning of the text file—repeating for as long as there is something to do.

To limit the repetitions to at most `n` times, use the form

```
!iterate=n
```

where `n` is a positive integer.

To repeat only a specific section of the script, use the `!begin`/`!end` constructs described below.

8.2 Nested scripts

A *nested script* is a part of a script which is enclosed between directives `!begin` and `!end`.

The nested script is treated like a pass of its own: First the substitutions above the nested script are performed, then the substitutions of the nested script, and finally the substitutions below the script.

What sets a nested script apart from a pass is locality of options. All options written within a nested script pertain to the nested script only. When `!end` is reached, all options defined in the nested script cease to take effect. This applies to options for letter case and context, but not to options that are inherently global: `!backup`, `!update`, `!screen`, `!lineseparator`, `!ctrlz`, `!binary` and `!text`. Notably `!ask` is *not* on this list, and may vary locally.

A nested script may itself contains nested scripts.

An `!iterate` directive within a nested script repeats only the nested script.

`!end` alone (not preceded by `!begin`) can be used to signify end of input. Any text following `!end` is ignored.

8.3 Named subscripts

Named subscripts are defined much like nested scripts. They are opened with `!sub` *subroutinename* instead of `!begin`, and end with `!end` just like nested scripts.

They are not executed, however, just by stating the definition. Instead they are stored under the specified name (*subroutinename*), later to be used in replacement text.

To use (call) a named subscript you write its name preceded by a \ (backslash) and an *argument* enclosed in parentheses. The subscript will then, when inserting the replacement text with the call, first execute the named subscript on the text in the argument. Example:

```
!sub capwords

 ~1:l~1~2:l+~2
 \toupper(~1)\tolower(~2)

!end
```

This script capitalizes words, converting the first letter to uppercase the rest to lower. Having defined the script, you can now use it in replacement text:

```
"My name is "‘:l+([:w]+:l+)*‘"."
"Remember to capitalize names, like this: "\capwords(‘)"."
```

Named subscripts are parsed and checked for errors when defined, but not compiled (the time- and space-consuming part) unless actually used.

8.4 Include files

From within a script file, another script file can be included with

```
!include=filename
```

This works almost the same as inserting the text of *filename* at the point of the `!include`, except the file *filename* must be a well-formed script.

An included file is *not* a nested script. Sometimes you want the included file to act like a nested script, to avoid options in the included file to change options in the file it was included from. In this case just put the directives `!begin` and `!end` at the beginning and end of the include-file.

Including files nest, i.e. you may include another file from within a file that is being included.

9 Caveats and bugs

No bugs described here: Known bugs get fixed!!

But there are some things that you should be ware of; shortcomings of the `Msub` way of doing things.

Memory

`Msub` is very memory hungry. Firstly, two copies of any file being worked on are kept in memory concurrently. Secondly, the regular expressions are compiled internally to some big tables. If your script is very complex or your files are very big, `Msub` may run out of memory or become very slow (due to *virtual memory thrashing*).

If a script is simple enough, `Msub` will only need to store *one* copy of the input file. This is the case for scripts, that are not multi-pass (no `!newpass` and no `!begin`/`!end`), and that have the `!binary` flag set. A complex script can probably be split into several simple scripts that each have these properties.

Command-line quoting

To use spaces in a regular expression on the command line, you must enclose it within " quotes. However, you cannot enclose part of a coherent option in quotes. This doesn't work:

```
msub -search="one thing" -replace="sth else" somefile
```
wrong

Instead, enclose the entire option in quotes like this:

```
msub "-search=one thing" "-replace=sth else" somefile
```

or use space instead of the '=' character:'

```
msub -search "one thing" -replace "sth else" somefile
```

These quotes only affect reading the command line. They do not make the arguments into literal strings. To make special characters literal you need additional quoting, this time using the ' single quote or "\". As in

```
msub -search "'$2 + $2'" -re=\$4
```

Misplaced marks

You may get the warning
`Warning: ... marker may become misplaced (program limitation)`
(Instead of "..." it will say "`Last ~1`", "`First /`" or something like that.)

(Until you actually get that warning, there is no need to read this.)

This happens because Msub is implemented using a technique called finite-state automata, which is very fast but not quite powerful enough to handle the full Msub scripting language. When you write a script that Msub cannot handle, it prints this diagnostic but continues anyway.

The problem arises when the regular expressions on each side of a marker 'overlap', as in this script:

```
/a*/ab
""
```

If there were no restrictions on what could follow `a*`, then it could match the `a` in `ab`. When this script is run on the input text `aab`, this should be divisioned as `/a/ab`, but the `a*` is too 'greedy' and this is (incorrectly) divisioned `/aa/b`.

A more realistic example: Suppose you want to delete comments in a file, and the "%" character makes the rest of the line a comment. So you think: A comment is a "%" character followed by anything (`[^]` means any character) till the end of line, and you write the script:

```
/"%"[^]*/$
""
```

This doesn't work quite as you expected, because the `[^]*` parts gets too 'greedy': The line-end character $ gets accidentally included in the "/" selection. To correct this, write:

```
/"%"[^$]*/$
""
```

It is possible that you get this warning even though there are no problems. That is why it's just a warning.

Ambiguity: Begins-earliest and longest-match

If the exact same piece of text might be matched by two different regular expressions in the same pass, this is an ambiguity that isn't allowed; Msub stops with an error message.

If different, but overlapping pieces of the input text might be matched by different regular expressions in the same pass, this is not an error and the ambiguity is resolved in one of two ways:

- The first match found, scanning left-to-right, is always preferred. E.g. with the search expressions `text` and `extend`, if it says "textend" in your text, then `text` will be found but not `extend`.

- If the two matches start at the same character, but they are of different lengths, the longer one is chosen. This is the *longest-match* strategy.

  This goes not only for 'competing' regular expressions, but also 'within' a single regular expression. E.g. `[A-Za-z]+` will match an entire word, even though it could make do with just the first character. To make this explicit, you could write`/[A-Za-z]+/[^A-Za-z]`.

All this applies to the *entire* expression, including context (i.e. not just the part between "/" delimiters). Except a new match may start in the trailing context of the previous match. For example, if the input text is `abcde`, and the `b` has just been matched by `/b/c`, then the `c` can be matched by `/c/` but not by `b/c/`, nor will `/bc/` match.

"Unchanged" files may have changed

Even if `Msub` prints `(unchanged)` after a processed file name, the file is nevertheless rewritten. Although no substitutions have taken place, the `!lineseparator` and `!ctrlz` options are still in effect. With the default settings this means that any LF line endings will be changed to CR/LF line endings.

The date stamp is never changed with files that are '`(unchanged)`'.

File attributes are not conserved. Instead the backup file will have the original attributes. If your original file was read-only, this means that `Msub` will fail the second time around, unable to delete the old backup file.

Infinite loops

If a search expression matches the empty string, and there is no preceeding context, then `Msub` will keep matching the position until it crashes with an `Out of memory` message. (If the empty string is replaced with empty string, then it will loop until you cancel it with '`ctrl-break`'.) If the preceeding context is non-empty, there is no problem, and there is no problem using that to insert text:

```
yes//
" or no"
```

If an abbreviation refers to itself, directly or indirectly, then using it will cause `Msub` to enter an infinite loop, and keep on looping until `Msub` runs out of memory or stack space.

Finally a script file that `include`s itself, directly or indirectly, will loop until there are too many files open, and `Msub` quits saying `Unable to open include file`.

You!

Writing regular expressions is a kind of programming. One of the basic tenets of programming is that mistakes *are* made. Never trust a complex regular expression until you have actually seen it generate the desired output.

10 Regular expression primer

A regular expression is a way of describing set of text strings. Just like DOS wildcards describes a set of filenames using special characters "`*`" and "`?`". Full regular expressions, however, have different and more special characters.

The simplest regular expression is the literal string. E.g. the regular expression "`joe`" matches the string "`joe`".

The "`j`" in "`joe`" matches only a lowercase j. Suppose you need to match both lower- and uppercase j's. Then you can use the "`|`" character as in "`J|j`" to indicate that you don't care if it's one or the other. To do that in "`joe`" you need to use parenthesis like this: "`(J|j)oe`". That will match both "`joe`" and "`Joe`". Whereas "`J|joe`" would match "`J`" and "`joe`".

Next suppose you want to search for "`joe`", but you don't want "`Joel`" to show up. So you want to find "`joe`" only if the next character is a space or a punctuation character. That would be "`joe(' '|','|'.'|';')`". Notice the use of 'quotes' around special characters "`,.;`" just in case they have a special meaning in regular expressions (the way we have seen "`(`", "`)`" and "`|`" have).

There is a short form for this: "`joe[' ,.;']`". A list of characters between [brackets] means match any of these characters.

Instead of describing what character may follow "`joe`", it might be easier to say what may not. If the first character within brackets is "`^`", the bracketed expression matches not the characters within the brackets, but rather any character which is *not* within the brackets. Thus you write "`joe[^abcdefghijklmnopqrstuvwxyz]`" to match "`joe`" followed by anything but a lowercase character. You don't need to type all 26 characters, it can be abbreviated "`joe[^a-z]`". Of course you want to avoid uppercase characters too, which makes it "`joe[^a-zA-Z]`".

Similarly "`[0-9]`" will match any single digit. To match an entire (integer) number, use the "`+`" operator, like this: "`[0-9]+`". The "`+`" operator repeats one or more times that which comes just before it. Therefore "`[0-9]+`" matches one or more subsequent digits, i.e. any integer number. The operator works only on the last character or parenthesized/bracketed expression. This means that "`and +`" won't match repeated instances of the word and: the "`+`" only repeats the space that it immediately follows, and the expression will match the word "`and`" followed by one or more spaces.

The "`*`" operator is just like "`+`", except it repeats 0 or more times. That is it may also match the empty string, e.g. "`ab*c`" will match "`ac`", "`abc`", "`abbc`" and so on.

The "`?`" operator indicates that something may be omitted. E.g. "`a (useful)? operator`" will match both "`a operator`" and "`a useful operator`".

By now you should be ready for the terse descriptions back in section 4.4.

11 Registration

This program is shareware. Which means that if you put this program to good use without sending me a contribution, then you are basically ripping me off. Because I have put a lot of working hours into this.

You'll find the accurate wording of the copyright and licensing conditions in the accompanying file `REGISTER.DOC`. That file must be included if you in any way distribute `Msub` or this document.

Registration is $25 or the equivalent in any convertible currency. If you have Internet access you can save $5 off that price by having programs and documents sent by e-mail. Otherwise you will receive a diskette and a printed manual by surface mail.

11.1 "What's in it for me?"

Registration benefits:

- You get a copy of the latest version available.
- This copy does not have the 3 second startup delay that the shareware version does.
- Your registration covers all versions 1.x, and if you have an Internet address you will get an e-mail notice with each release.
- You can get the source code of `Msub`. You can recompile this on other platforms such as OS/2 or unix, provided you have a (reasonably standards-conforming) C++ compiler.
- Any money you send me will help me pay for an optimizing compiler. Which may double the speed of `Msub`.

11.2 How to register

See `REGISTER.DOC` for details.

11.3 Future enhancements

Here follows a list of coming improvements to `Msub`:

Wordprocessor documents A future version of `Msub` will understand the word-processor file formats of WordPerfect, Word and others, making it easy to manage nested codes and steer around them if necessary.

You can help me with this: If you have any information on a word processor file format, please send it to me. This is also the may to make sure that

`Msub` will support your favorite FooWord or BarWrite format: If I don't know the specifics of format, I can't support it. And unfortunately buying specs from the sources (WordPerfect Corporation and such) is way beyond the financial range of `Msub`. The others are too expensive, and `Msub` is too cheap!

... hm ... this list used to be a lot longer, but I seem to have implemented it all ☺

12 Closing argument

You can reach the author with questions and comments on the Internet as `juul@diku.dk`.

Enjoy ☺