Version avec modifs comment c'est
(RL)

MORTRAN USER GUIDE- DD/US/18

22.03.1979

Rolf Nierhaus

Introduction.

Mortran is a very practical approach to "structured programming" especially for those who have a long history of Fortran programming or a large library of Fortran programs. This has to do with the relationship of Mortran to Fortran. Mortran, which seems to suggest "More than Fortran", was meant to be an extension of Fortran and got Fortran compatibility built-in at many levels. Unlike other "structured programming" languages, Mortran uses Fortran files and Fortran input-output, Fortran subroutine calls and Fortran common. Moreover there is a switch in the Mortran translator which allows the use of "in-line" Fortran.

The main advantages of Mortran over Fortran are that Mortran is a language for structured programming, like Pascal, PL/I, the Algols and BCPL. There are statement blocks and conditional and iteration statements which act on statement blocks. The conditional statements are the IF, the ELSE, the ELSEIF and the UNLESS. The iteration statements are the DO, the FOR, the WHILE, the UNTIL and the LOOP. Testing at the end of a loop is possible, and there are the NEXT and EXIT statements, which transfer control to the next iteration or out of the loop.

Other advantages of Mortran over Fortran are that Mortran is more "free-field", allows statements of any length and comments to be anywhere. There are multiple assignment statements and abbreviated input-output statements. Conditional compilation is available, as is switching between different input files.

One advantage which Mortran has not only over Fortran, but also over many other "structured programming" languages is its macro capability. The macro replacement feature is very often used for the global symbolic definition of Fortran constants, but it has much more sophisticated applications and leaves the language somewhat open-ended.

Mortran was designed and first described by James Cook of SLAC in 1973. Since then James Cook has written several translators for it. The MP2 processor, which was written at CERN in 1977, has been designed to be compatible with James Cook's Mortran2-processor.

MP2 however does not use the same set of macro-definitions to describe the Mortran language, and extensions of it have to be described by a different set of macro definitions.

The differences between the processor MP2 and the Mortran2-processor are:

1. MP2 is about 2 times faster.

2. The MP2 implementations on the CDC-machines are Update compatible, while the implementation on the IBM is Patchy compatible.

3. MP2 issues a message about the number of errors encountered for each subroutine separately.

From Fortran to Mortran.

Mortran is often called an extension of Fortran. This is however not strictly true, since Mortran is really an extension of modified Fortran. The modifications have nothing to do with the structured programming aspects of the Mortran language, but concern only the source card layout, which is more free field in Mortran.

They can be expressed in 4 rules, which are really fundamental, and you must learn them by heart before you can start programming in Mortran. These are the rules:

1. The Fortran convention of one statement per card with eventual continuations using a marker in column 6 is awkward. In Mortran statements can be any length and are completely independant of card boundaries. The price we have to pay for this is that each statement must be terminated by a semicolon.

2. Comments can be introduced into Fortran only by separate comment cards and are therefore rarely at the place where they would be most useful. In Mortran comments are enclosed in double quotes and can be anywhere, even in the middle of statements.

3. Though in Mortran you rarely need statement labels, there is a replacement for Fortran statement numbers. Fortran statement numbers are not free field and have no mnemonic value. In Mortran statement labels can be alphabetic or numeric, are not restricted in length and can be anywhere on the source card. However they must be enclosed in colons.

4. The last modification concerns Hollerith character strings: Instead of a count followed by the character H followed by the string, Mortran requires the character string enclosed in single quotes. This is obviously an advantage, since you don't have to count. (Single quotes within quoted strings are represented by two successive single quotes.)

Some Fortran Hollerith strings are accepted by the Mortran processor, and only strings which contain special characters which have special meaning in Mortran require the Mortran string format with single quotes. One such character is the space or blank, which Mortran tries to suppress subjected to certain conditions.

By applying these 4 rules you can change any Fortran program into a Mortran program.

The merit of the ELSEIF statement is that it allows many conditions to be tested consecutively by an initial IF statement followed by ELSEIF statements without increasing the nesting level. The last statement might or might not be an ELSE.

The UNLESS statement

UNLESS logex <block>

is the logical converse of the IF statement and means

IF .NOT.logex <block>

WHILE logex <block>

The logical expression logex is tested first, if it is true the block is executed and logex is tested again. As soon as logex is false control goes to the statement following the block.

UNTIL logex <block>

The UNTIL loop is the logical converse of the WHILE loop. Here block is executed if logex is false, and as logex becomes true, control passes to the statement following the block.

It is possible to specify additional testing at the end of a block, as in

```
WHILE logex1 <block> WHILE logex2
WHILE logex1 <block> UNTIL logex2
UNTIL logex1 <block> WHILE logex2
UNTIL logex1 <block> UNTIL logex2
```

If testing only at the end of a block is intended, an infinite loop can be specified.

```
LOOP <block> WHILE logex
LOOP <block> UNTIL logex
```

All iteration structures provide for the possibility to terminate the current iteration or to terminate the whole iteration loop, as it is possible in Fortran to jump to the CONTINUE statement indicating the end of the range or to jump out of the DO-loop. The control statements to accomplish this are NEXT and EXIT. Corresponding statements exist in other languages suitable for structured programming. They have however not always the same names. For example in BCPL the the Mortran NEXT is called LOOP and the Mortran EXIT is called BREAK.

If iteration blocks are nested, the statements NEXT and EXIT refer to the innermost block. It is however possible to exit from one of the outer blocks, or to start a new iteration of it. To do this the iteration block must be labelled, that is preceeded by an alphanumeric label enclosed in colons, for example

```
:LABEL: DO I=1,IMAX,2 < ... >
```

Then the statement EXIT:LABEL: specifies exit from the DO-block LABEL, and the statement NEXT:LABEL: specifies to begin the next iteration in the DO-block LABEL.

Spaces in the input text are suppressed according to a very simple rule: All spaces are suppressed except spaces within quoted strings and one single space between two alphanumeric characters. Macro definitions contain two quoted strings, and any spaces here will not be suppressed.

If the pattern of a macro definition begins with an alphanumeric character, it will only be matched if it occurs in the input text after a special character. In other words: matching of an alphanumeric string will never start in the middle of it, but only at its beginning.

Macro definitions are either global or local. They are global if the macro definition precedes the first Mortran statement which generates code in the Fortran file. Macros declared later are local, they lose their effect as soon as an end-statement is encountered, liberating the space they took up in the macro buffer for local macros defined in the next program unit.

If several macro definitions specify the same pattern, only the latest definition is used for macro expansion.

It is possible to override a global macro definition for only one program unit by specifying the same macro pattern in a local macro definition. As soon as an end-statement is encountered, the local macro is removed from the macro list and the global macro takes effect again.

The end of an actual parameter is signalled by a match between the character following the parameter in the input text and in the macro definition. However the empty string is only acceptable as a parameter if the next character is not alphanumeric. A space is not acceptable as a parameter, except if the preceeding character is the single quote. However it often happens that a parameter begins with a space.

It is possible to suppress all the rules to which parameters are normally subjected and to specify a parameter of length 1, or a one-character-parameter. This is indicated by two consecutive #-characters in the pattern part of a macro definition.

The sequence ### in the pattern part of a macro means a one character-parameter followed by a normal parameter, the sequence #### two one-character-parameters, etc.

In the replacement part of a macro however two consecutive #-characters have a completely different meaning: ## specifies the character # as replacement text.

#P **PARAMETER**
Load the first word of the first parameter into the accumulator. (Parameters are initially character strings, stored one character per word. If macros are nested, a character might be replaced by a binary number. This is for exemple the case during the translation of user defined statement labels. The binary number must be converted to decimal (operator #C) before it can be output.)

character ?

#N **NO RESCAN**
After the expansion of a macro, scanning for macro patterns continues at the beginning of the replacement text. If the replacement text however contains the operator #N (no rescan), the replacement text before #N will be skipped and scanning continues at the character after #N.

#N$ — no rescan ... operation : output of all the non-rescanned replacement and reset Replacement counter to 1 (X(RP))]

#B **OCTAL**
Convert the first parameter from octal to binary and load it into the accumulator.

#H **HEXADECIMAL**
Convert the first parameter from hexadecimal to binary and load it into the accumulator.

#Z **STRING**
Pack the first parameter, which is supposed to be a short string of 6-bit characters in internal representation and load it into the accumulator.

#E **END**
Check that all stacks are empty. If not write the error message "unclosed block". Reset all stack pointers. Write a message indicating the number of errors encountered. Reset the error counter. Reset the label register to 10000. Reset the macro list pointers to the values they had before the first Fortran output statement was generated.

#T **TITLE**
Transfer the first parameter up to an eventual left parenthesis or up to a maximum of 24 characters to a title buffer for printing in the header line on top of the next page.

#R Macro concatenation. &'#R macro' = 'rep' adds characters 'rep' to ~~left~~ right of previous macro replacement.

#K Macro-phage : kills latest patt. macro &'#K patt' : ...
[or delete latest 'rep' concatenated]

#* Génération de commentaires (2 champs : jusqu'au 1er ^ : répété sur continuation.)
Attention : il faut ;#*_comment_; (et double le niveau !)

Alternatively we could write a "macro generating macro" and add only a short statement requesting the trace of J. If we then needed the trace of again another variable practically no effort would be required anymore.

Let's look at an example of a suitable macro generating macro:

```
&';TRACE #;'=' &'';#1=##;''='';#1=##N##1;OUTPUT #1;
('''' #1 SET TO''''I10,
''''  BY EXECUTION OF THE STATEMENT #1 = ##1'''');'''
```

The pattern part of this macro is very simple, ';TRACE #;', and would for example be matched by the statement TRACE J with J as (first) parameter. To understand the replacement part, let's rewrite it with the beginning and ending string quotes removed. Then we must contract two consecutive string quotes within the string into one single quote:

```
 &';#1=##;'=';#1=##N##1;OUTPUT #1;('' #1 SET TO''I10,
''  BY EXECUTION OF THE STATEMENT #1 = ##1'');'
```

This string starts with a space, and this space is quite essential for the macro processor to identify the following &-character as the start of a macro definition. Since this string occurs as the replacement part of a macro, the meaning of #1 is: the first parameter, and the meaning of ## is: replacement by the character #. If we now do the replacement, assuming for example that the first actual parameter is J, we get

```
 &';J=#;'=';J=#N#1;OUTPUT J;('' J SET TO''I10,
''  BY EXECUTION OF THE STATEMENT  J = #1'');'
```

This is exactly the trace macro for J considered before.

So with one complicated macro generating macro and some short statements which match its simple pattern we can easily generate many additional macros of a certain complexity.

[B: il y a un bug qui empêche de terminer un paramètre par un < : '.. #<' => erreur ni nouvelle définition, bad character is <'

Mortran listing.

The Mortran listing is switched on with the control statement &L and switched off with the control statement &N.

The listing consists of a copy of the text read from the current input unit, possibly preceded by either of the characters * or S or by the nesting level count. The characters * or S are printed on the left margin of a line if a comment string or a quoted string is not closed on that line. The nesting level is printed only if it has changed compared to the nesting level of the previous line. The printed nesting level refers always to the beginning of the line.

Every page begins with a header line, which contains a 24 character field for the identification of the current text. The standard macro set contains four macros which will insert the program, subroutine of function name or possibly the text BLOCK DATA into this field.

The transfer mechanism fails if for example a subroutine statement is too long to be contained on a single line. The following text could be used to obtain a proper listing:

```
&N
&'SUBROUTINE#;'='SUBROUTINE#1#N;'
&'TITLE #;'='#T;&E;';
TITLE SUBROUTINE SUB;
&L
SUBROUTINE SUB(PARAM1,PARAM2,PARAM3,PARAM4,PARAM5,PARAM6,PARAM7,
PARAM8,PARAM9);
```

The first macro switches off a global macro which would transfer the subroutine name into the identification field. The second macro specifies in the replacement part the operation "title" #T which transfers the first parameter into the identification field and the control statement "eject page" &E. This is an example of a macro which contains a control statement in the replacement text. Both macros are local and lose their effect after the next end statement, except if subroutine SUB were the first program unit to be compiled.

During tracing no header line is printed.

If automatic indentation has been specified, columns 73 to 80 of the input line will be printed at the right margin of the output line. If the input line is longer than 80 characters, character 81 and all following characters will not be printed. The output line can receive 34 characters shifted out of column 72 during the indentation process. If any character is shifted beyond column 106 it will be lost.

&T trace:
Trace macro replacements.
If a macro pattern is recognized up to a first parameter, the pattern is traced from the macro buffer, where parameters are indicated by #. Then the parameters are traced, and the pattern is traced again, this time from the expansion buffer, that is with the actual parameters. The trace of the replacement part is restricted to what is written into the expansion buffer. In addition the generated Fortran statements are written onto the list file.

&T0 no trace:
Stop tracing macro replacements.

&Un (n=1,...,5,8,9) switch input unit:
Stack the logical unit number for the current input file and switch input to logical unit number n. If the stack is full an error message "input unit stack overflow" is printed.

&& can be used as an end of file indicator. Every time this control statement is read or a system end-of-file indicator is set, a logical unit number is unstacked from the input unit stack and input is switched to it.

&V file → sortie [sur unit 10]. Quel est le raccourci de &V ou le texte) la suite précédente est fermée et on continue sur la nouvelle.

For example the set

```
&'IBM-'='#G1'
&'CDC-'='#G2'
&'-END'='#G3'
```

would provide for the generation of code for the IBM, while the set

```
&'CDC-'='#G1'
&'IBM-'='#G2'
&'-END'='#G3'
```

would provide for the generation of code for the CDC. When there is no nesting, the string '#G1' can be replaced by the empty string ''.

For compatibility with the Mortran2-processor the following macros are contained in the standard set

```
&'GENERATE'='#G1'
&'NOGENERATE'='#G2'
&'ENDGENERATE'='#G3'
```

so that it possilbe to specify GENERATE, NOGENERATE or ENDGENERATE instead of #G1, #G2 and #G3.

How to run Mortran on the CDC 7600.

You can attach MP2 with the control card FIND,MTNB,MTNB,ID=PULIB. The set of standard macros can be attached with the control card FIND,TAPE1,MTNMAC,ID=PULIB. If you use only the standard macros and only one program file on INPUT, the following control cards are sufficient:

```
FIND,MTNB,MTNB,ID=PULIB.
FIND,TAPE1,MTNMAC,ID=PULIB.
MTNB.
FTN,I=TAPE7.
```

Alternatively you could use the control card macro MTN:

```
FIND,MTN,ID=PULIB.
MTN.
```

If you want to read the Mortran source from file COMPILE, specify: MTN,I.

Or if you want to read the Mortran source from file ZZZZ, specify: MTN,I=ZZZZ.

All other parameters on the MTN-card are transmitted to the Fortran compiler.

If you don't want to use the standard macros, of if you have more than two input files, or if you want the listing on another file than OUTPUT, etc. you must use your own control cards or your own control card macro. The different files used by MP2 appear on the program card in the following sequence: TAPE1, INPUT, OUTPUT, TAPE7, TAPE2, TAPE3 and TAPE4, where INPUT is equivalenced to TAPE5 and OUTPUT to TAPE6. The generated Fortran is on TAPE7.

Mortran is also available on MFB. Here the processor is called MTNB6,ID=PULIB and the size of its macro storage area is slightly reduced. The standard macro file is called MTNMAC6, and the following is a minimum set of control cards:

```
ATTACH,MTNB6,MTNB6,ID=PULIB.
ATTACH,TAPE1,MTNMAC6,ID=PULIB.
MTNB6.
FTN,I=TAPE7.
```

MACRO DEFINITION ERROR. ILLEGAL STACK NUMBER.
In the replacement text of a macro either of the operators stack #S or unstack #U is specified. However the characters S or U are not followed by any of the legal stack identifiers (digits 1 through 9).

STACK OVERFLOW.
The stacks are 50 words deep, and the error occurs for the first time when the stack pointer is at 45. After 5 warning messages the processor loses control in an unpredictable way.

STACK UNDERFLOW.
In the replacement text of a macro an operation is specified (#U) which tries to retrieve something from a stack which has not been put there. Since a right bracket is translated into unstack operations, this could for example occur, if there were a closing bracket without a corresponding opening bracket.

CONTROL CARD ERROR.
The character & specifies a macro definition if it is followed by a string quote. It specifies a control statement if it is followed by an alphabetic character. A control card error occurs if the character following the & is not a string quote and not alphabetic, of if it is alphabetic, but not one of the legal characters specifying a control action.

ILLEGAL INPUT UNIT. CONTROL CARD ERROR.
The character following the U in the control statement &U is not a digit.

ERROR IN STATEMENT LABEL.
Fortran statement numbers are transmitted to the MP2 output processor as five digit numeric strings followed by a semicolon. If a statement begins with a numeric character, it is interpreted as a statement number. An "error in statement label" occurs if the string is longer than five characters.

MACRO BUFFER OVERFLOW.
The size of the macro buffer is 7000 words on the IBM and the CDC 7600. It is 6000 words on the CDC front-ends. To define the Mortran language less than 1500 words are needed. Macro storage space for statement labels is released after each end-statement.

PARAMETER BUFFER OVERFLOW.
The total length of all parameters cannot exceed 700 words, including up to 18 pointer- and flag-words. Another 800 words is available for the storage of strings which are subjected to tests, but fail in the end to be parameters.

EXPANSION BUFFER OVERFLOW.
Input text is first stored in the upper half of the expansion buffer. If more than 2000 characters are read without an intervening semicolon, the expansion buffer overflows.

The macro processor MP2.

MP2 is a Fortran program of about 1000 source statements. It consists of a main program, a block data program unit and 9 subroutines. The main program is the macro processor. It calls upon a number of service subroutines, namely

an input processor: subroutine XIN,
an output processor: subroutine OUT,
3 interfaces to the I/O routines: subroutine READ, subroutine WRITE and subroutine NPAGE,
a control statement analyzer: subroutine CCA,
an error-message printer: subroutine ERROR,
and 2 tracing routines: subroutine TRACE, which traces patterns, parameters und replacements when macros are called, and DTRACE, which traces all macro definitions stored in the macro buffer.

The macro processor uses an internal character set of 63 characters. Subroutine READ converts into this character set, and subroutine WRITE converts back into the Fortran character representation, as do the tracing routines. Characters 1 through 27 are special characters, those which are used by MP2 come first. Characters 28 through 53 are the letters from A to Z and characters 54 through 63 are the digits from 0 to 9. 0 is not a character which can be obtained from input and is therefore widely used as flag-word.

These characters are defined in the data statement for OTAB in the block data program unit. This table is used for output conversion. Input conversion uses table ITAB, which is initialized by the ~~main~~ program. The input character set may comprise up to 256 characters, those not defined convert into internal character 23, the question mark.

Character 1 is the space or blank. This character is distinguished by its suppression according to certain rules.

Character 2 is the end of statement character (the semicolon).

Character 3 is the comment character (the double quote). The exclamation mark is converted into character 3 too.

Character 4 is the string character (the single quote).

Character 5 is the parameter and operator flag in macro definitions (#).

Character 6 is the first character in macro definitions and control statements (&).

Character 7 is the statement label character (the colon).

The input processor (subroutine XIN) transfers text from the input buffer to the expansion buffer. It recognizes the comment character, the string quote, the end of statement character and the blank. Text included between comment characters is not transferred at all. If a first comment character is immediately followed by a second comment character, one single comment character is transferred. Text included between string quotes is transferred unmodified regardless of blanks, end of statement characters and comment characters. All other text is transferred only until an end of statement character is encountered. A zero flag word is written behind it and subroutine XIN returns.

Blanks are suppressed if they are adjacent to special characters

will be available during expansion.

The first 9 locations of the parameter buffer P contain up to 9 pointers to actual parameters. Each parameter is terminated by a zero flag word. The number of parameters in the pattern currently under expansion is held in a separate location (NP).

If some piece of text does not match any pattern it is transferred to the beginning of the expansion buffer. The end of the transferred unmatched text is pointed to by an output pointer.

If then some piece of input text is matched by a macro pattern the macro replacement text is placed into the expansion buffer beginning at the output pointer and its end is pointed to by a replacement pointer.

If during expansion a NO-RESCAN operator is encountered the output pointer is set to the value of the replacement pointer and the replaced text is thus marked for output. Any other replacement text is moved in the expansion buffer to become adjacent to the still unscanned text. Scanning will then continue at the beginning of the moved replacement text. If during expansion the replacement text overwrites still unscanned text, this is a fatal error, named "macro expansion loop". This error can occur however if the replacement text is just too long.

For the translation of the Mortran language the following use is made of the different stacks:

Stack 1 contains a block identifier:
- 1 for an IF- or UNLESS-block,
- 2 for an ELSEIF-block,
- 3 for an ELSE-block,
- 4 for a WHILE- or UNTIL-block,
- 5 for a FOR- or DO-block and
- 6 for a block without a control statement.

Stacks 2 and 3 contain labels for the translation of conditional statements:
- stack 2 the ELSEIF-label and
- stack 3 the END-IF-label.

Stacks 4 and 5 contain labels used in iteration statement translation, namely
- stack 4 the NEXT-label and
- stack 5 the EXIT-label.

Stack 9 is used only for temporary storage, and stacks 6 through 8 are not used at all.

User defined labels are translated into macro defining macros.