400.30 **FORTRAN**

400.30.30 **Precompiler MCSTRUFO**

1 Introduction

The term MCSTRUFO, like FORTRAN, has several meanings, depending upo the context in which the term is used. MCSTRUFO has come to mean

- A Structured Language
- A Translator for that language
- A Macro-processor

MCSTRUFO is shorthand of **MaCro** oriented **STRU**ctured **FO**rtran.

The structured language is implemented as a set of **macros** which are used by the macro processor to translate the language into FORTRAN. The resulting FORTRAN program is then run like any other one. User-defined macros are easily added to the standard (language defining) set of macros so that the language is "open-ended" in the sense that extensions to the language may be made at any time by the user. Extensions have ranged from very simple ones like matrix multiplication to complex ones like those which define new data types.

The user need not concern himself with the method of implementation or the macro facility in order to take advantage of the structured language which is provided by the standard set of macros. The feature of this language include:

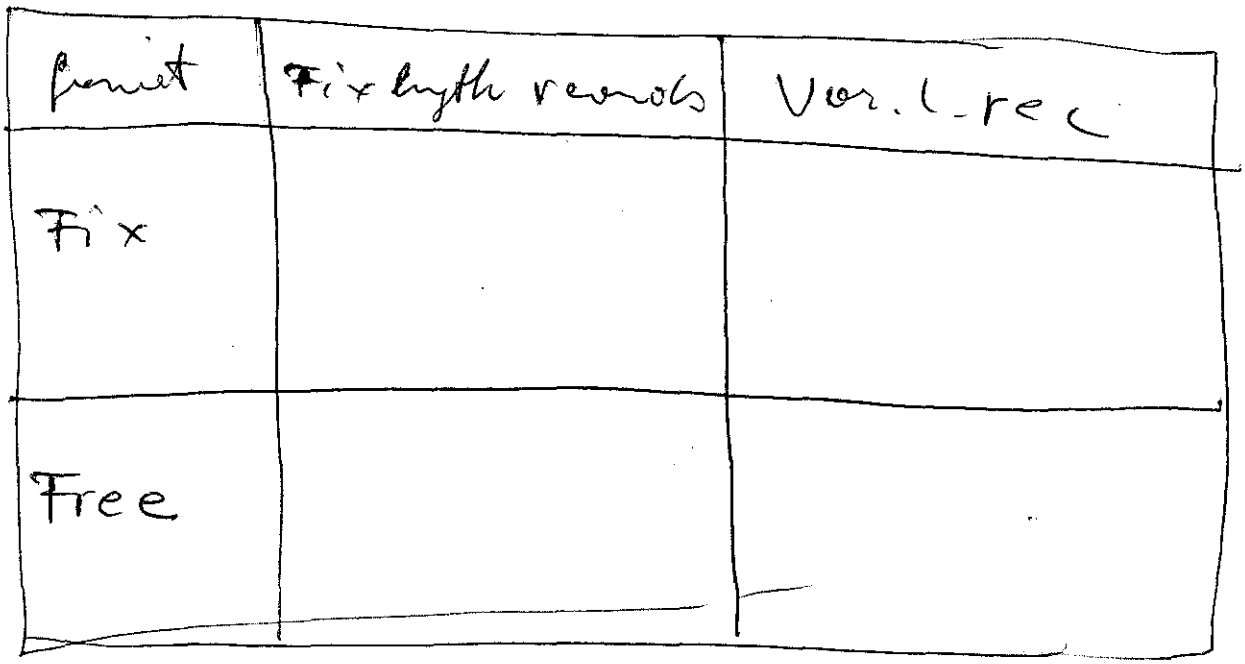
- Multiple statements per line terminated by semicolons
- Free-field (column and card boundaries may be ignored)
- Alphanumeric labels of arbitrary length
- Comments may be written in several forms
- Nested block structure
- Conditional statements which may be nested (IF, ELSE, and ELSEIF)
- Loops (repetitively executed blocks of statements) which test for termination at the beginning or end or both or neither (WHILE, UNTIL, FOR-BY-TO, LOOP and DO)
- EXIT (jump out) of any loop
- NEXT (go to NEXT iteration) of any loop
- Remote PROCEDURE (a local subroutine)
- Multiple assignment statements
- Conditional (alternate) compilation
- Abbreviations for simple I/O statements
- Interspersion of FORTRAN text with MCSTRUFO text
- graphic annotation of relational and logical operators
- assignment of non-numerics
- pseudo-GENERIC for precision encrease
- I/O features include
  - automatic indentation according to nesting level
  - LIST/NOLIST, SPACE, EJECT
  - INCLUDE from copylibrary
  - macro trace

The user may elect to override the standard set of macros and write sets which defines another, perhaps "problem oriented", language.

The MCSTRUFO processor is a FORTRAN program of approximately 1100 statements plus an assembler program of about 2000 lines. The macros to define the language require about 80 lines.

MCSTRUFO is the OBRZ implementation of **MORTRAN2**, which was developped by A. James Cook and L.J.Shustek from the Computation Research Group of Stanford Linear Acceleration Center, Stanford, California 94305. For details of our implementation see chapter 400.90.32.

The text You are reading was written originally by the authors of MORTRAN. We (OBRZ) only made some extensions and rearrangements to meet the requirements of our implementation.



2 Coding Rules

MCSTRUFO programs may be written with respect to columns as known from common FORTRAN (**fix format**) or in **free format.**

Only linecontents up to columns 72 is relevant to MCSTRUFO. The rest (columns 73 to 80) are only listed.

FIX format	FREE format
Columns 1...5 may only contain blanks or a pure numeric **label**. User defined labels should be in a range below 90000, because MCSTRUFO generates labels in the range 90000 ... 99999)	All columns 1...72 may contain statement text. A statement may be preceeded by a pure numeric label as in fix format.
Column 6 contains either a blank or the figure 0 to define this line as the beginning of a statement - or any other character to define this line as a **continuation line**.	
There may be more than 19 continuation lines as long as the relevant text (past macro expansion!) contains less than 1320 characters per statement (exclusive label).	Statements may contain as many as 1326 characters (including a preceeding generated label) - this ist counted past the macro expansion.
A **C** in column 1 indicates this line as a **comment**, which is not transferred to the processor.	
One line may contain more than one **statement** if they are separated by semicolons. Thus you can produce pseudo freeformat input when writing the first line of code starting at column 7, continue the code on the next lines with a nonblank character at column 6 and terminating each statement by a semicolon.	Statements are terminated by semicolons.

A **character string** comprising a hollerith field is enclosed in apostrophes (as in 'THIS IS HOLLERITH DATA'). If an embedded apostrophe is desired as a character within a quoted string, use a pair of apostrophes to represent each such embedded apostrophe (-> 'DON''T')

NOTE the **hollerith** form of strings (nH....) is **not** allowed and gives unwanted results, if there are blanks or apostrophes in the hollerithfield.

The characters with the hex representation FE and FF are not valid in strings, since they are used by the precompiler (FE) or compiler (FF).

In MCSTRUFO a **comment** may be inserted anywhere in the program except in strings. In macros only the form with " is allowed - but be carefully with the placement of it! There are three forms of comment

- the comment is enclosed in quotation marks as in "COMMENT *"
- the comment starts with the exclamation sign ! and extends to the end of the line.
- in FIX format there is (of course) the original FORTRAN comment with the character C in column 1.

In MCSTRUFO, an alphanumeric **label** is a character sequence of arbitrary length enclosed in colons (as in :TOMATOES:). The characters which comprise the sequence may be any combination of letters and digits. It may be used anywhere a FORTRAN statement label is allowed.

Multiple **blanks** (a sequence of two or more blanks) in a MCSTRUFO program are equivalent to a single blank except in quoted strings, where all blanks are preserved, and in macros (see there).

NOTE the character combinations << (begin block), >> (end block) and " (representation of apostrophes within strings) must not be divided into single characters at end of line or so!

NOTE If the **block-delimiter** << or >> occures more than once in a sequence, than they should be written with inserted blanks:

IF (a .eq. b) << ....;IF (q .and. p) <<....;>> >>

NOTE The **END-statement** must be written solely onto one line and should either be written with at least 5 trailing blanks or an immidiately following semicolon (to detect it).

NOTE The **direct-access** READ/WRITE (IBM-FORTRAN) should be written at the beginning of a line, so that the imbedded apostrophe - which is not a string-delimiter - may be detected:

```
fix fmt: 800  IF (A .EQ. B)  WRITE (7'IDN) LISTE;
  or          IF (A .EQ. B)
             *               WRITE (7'IDN) LISTE;

free fmt: :labll: IF (A .EQ. B) WRITE (7'IDN) LISTE;
  or              IF (WRITE .NE. 0)
                     WRITE (7'IDN) LISTE;
```

NOTE Any **IMPLICIT** statement or explicit specification statement must not contain refereces for **variable names** starting with $. For example the following is **not valid**:

```
IMPLICIT REAL*8 (O-Z,$);
REAL*8 FSIQN; INTEGER*2 $TEXT;
```

These constructs may cause conflicts with generated COMMONs since MCSTRUFO uses such variables and routine-names for special purposes.

3 Program structure

For structuring the source text MCSTRUFO offers

- statements can be grouped giving a block
- conditional statements for blocks, which may be nested (IF - ELSEIF - ELSE)
- remote PROCEDURE to define a local subroutine. It can be terminated by LEAVE
- many forms of iterations (UNTIL, WHILE)
- common loop with DO
- infinite LOOP terminable by EXIT
- FOR loop with arbitrary control expression
- EXIT from loops
- force NEXT iteration in loop

3.1 Statements

FORTRAN may be regarded as a subset of MCSTRUFO, since (with the minor exceptions of hollerith strings and the (IBM special) direct access READ/WRITE any valid FORTRAN statement becomes a valid MCSTRUFO statement when

- it is terminated by a semicolon, and
- continuation marks (if any) are deleted.

... and this is done automatically by the I/O-module according to the defined input-format.

3.2 Blocks

A MCSTRUFO **block** is a sequence of MCSTRUFO statements enclosed in the special characters << and >>, which we will call "brackets". The left bracket may be read "begin" and the right bracket may be read "end". This sequence must contain at least 2 statements.

```
Let      S1; S2; S3;...Sk;...Sn;                                          (3.21)
```

be a sequence of statements. The sequence becomes a block when it is enclosed in brackets

```
         << S1; S2; S3;...Sk;...Sn; >>                                    (3.22)
```

(The ellipses (...) are meta-symbols indicating arbitrary repetition. The brackets are not meta-symbols; they are delimiters in the MCSTRUFO language.)

Blocks may be nested. That is, any of the statements in a block may be replaced by a block. For example, in (3.22) we could replace Sk by a block and write

```
         << S1; S2; S3;...<< T1; T2; T3;...Tm; >>...Sn; >>                (3.23)
```

The block containing the sequence T1;...Tm; is completely contained, or nested, within the block containing the sequence S1;...Sn;. We will frequently write an ellipsis enclosed in brackets <<...>> to denote a block.

```
example     << X=Y; CALL SUB(A); B=1; >>
```

3.3 Conditional statements

The simplest form of a**conditional statement** in MCSTRUFO is written

```
IF  e <<...>>                                                    (3.31)
```

where e is an arbitrary logical expression, and the ellipsis enclosed in brackets denotes a block as described above. If e is true then the statements in the block are executed. If e is false, control is transferred to the first statement following the block. For example:

```
IF A .LT. B << C=D; E=F; >> G=H;
```

(notice that the parantheses around the logical expression are optional here - but not in the original FORTRAN logical IF)

If A is less than B then the statements C=D and E=F are executed after which G=H is executed. If A is not less than B control is transferred directly to the statement G=H.

Next in complexity is the**IF-ELSE** statement, which is written

```
IF e <<...>> ELSE <<...>>                                        (3.32)
```

If e is true then the statements in the first block are executed and control is transferred to the statement following the second block. I e is false then the statements in the second block are executed and control is transferred to the statement following the second block. consider

```
IF A .LT. B << C=D; E=F;>>
ELSE << G=H; I=J;>>

K=L;
```

If A is less than B the statements C=D and E=F are executed after which control is transferred to the statement K=L. If A is not less B the statements G=H and I=J are executed after which control is than transferred to the statement K=L.

Consider

```
IF A.EQ.B << X=Y;>>
```

Here, the block to be executed whenever A is equal to B consists of the single statement X=Y;. An alternate form acceptable in MCSTRUFO i the standard FORTRAN logical IF

```
)8 (A.EQ.B) X=Y;
```

IF-ELSE statements may be nested to any depth. Even so, the IF-ELSE is not really adequate (in terms of clarity) for some problems that arise. For example, consider the following "**case analysis**" problem:

3.3.1 Nested conditions

suppose we have

```
four logical expressions,        p, q, r, and s,
and  five blocks of statements, A, B, C, D, and E.
```

Now suppose that p. q, r, and s are to be tested sequentially. When the first TRUE expression is found we want to execute the statements in the corresponding block (p corresponds to A, q to B,

etc.) and the transfer control to the statement following block E. If none of them is true we want to execute block E. Using nested IF-ELSE statements we could write:

```
IF p <<A>>
   ELSE << IF q <<B>>
        ELSE << IF r <<C>>
             ELSE << IF s <<D>>
                  ELSE <<E>>
             >>
        >>
   >>
```

While this does what we want, it is awkward because each ELSE increases the level of nesting. MCSTRUFO offers the ELSEIF statement as an alternative:

```
IF     p <<A>>
ELSEIF q <<B>>
ELSEIF r <<C>>                                          (3.33)
ELSEIF s <<D>>
ELSE     <<E>>
```

Using **ELSEIF** instead of ELSE allows all the tests to be written at the same nest level.

In summary, an IF statement may be optionally followed by any number of ELSEIF clauses which in turn may be optionally followed by a single ELSE clause.

3.3.2 Reverse condition

The condition

```
        IF NOT expression << block >>;
```

is similar to the notation

```
        UNLESS expression << block >>;
```

3.4 Iterations

A MCSTRUFO loop is a block which is preceded by, and optionally followed by a "control phrase".

One such phrase is "**WHILE** e". One of the loops we may write with this phrase is

```
WHILE e <<...>>                                    (3.41)
```

The logical expression e is tested first. If e is true the block is executed and then control is returned to test e again. When e becomes false, control is transferred to the first statement following the block.

Another control phrase is "**LOOP**". If we wanted to test at the end of the loop instead of the beginning, we could write

```
LOOP <<...>> WHILE e ;                             (3.42)
```

In (3.42) the block is executed first. Then, if the logical expression e is true, the block is executed again. When e becomes false control is transferred to the statement following the loop (that is, the statement following the " WHILE e ;").

The logical converse of the WHILE loop is the **UNTIL** loop.

```
UNTIL e <<...>>                                    (3.43)
```

The logical expression e is tested first. If e is false the block is executed and then control is returned to test e again. When the logical expression becomes true, control is transferred to the first statement following the block. Similarly, the logical converse of (3.42) may be written by replacing the WHILE in (3.42) by UNTIL.

Tests for**loop termination** may be made at both ends of a loop. For example, if e and f are logical expressions

```
WHILE e <<...>> UNTIL f ;
WHILE e <<...>> WHILE f ;

UNTIL e <<...>> WHILE f ;
UNTIL e <<...>> UNTIL f ;
```

all test at both the beginning and the end. The above list is by no means exhaustive, but we must develop other "control phrases" in order to complete the discussion.

The iteration control phrases discussed above do not involve "control variables", that is, variables whose values are automatically changed for each execution of the loop. The following loop involves a control variable:

```
FOR v = e TO f BY g <<...>>                        (3.44)
```

where v is the control variable, and e, f, and g are arbitrary arithmethic expressions. The control variable v must be of type REAL or INTEGER or may be an array element (subscripted variable). The value of any of the arithmetic expressions may be positive or negative. Moereover the magnitudes as well as the signs of f and g may change during the execution of the loop.

The control variable v is set to the value of e and the test for loo termination (see below) is

made. If the test is passed then the block is executed, after which v is incremented by the value of g and control is returned to the test. Note that the block is never executee if the test fails the first time.

The "test" for the termination of a FOR loop refers to the logical expression

```
g * (v-f) .GT. 0                                    (3.45)
```

This test may in some few times may cause problems with overflow, if the values g and (v-f) are of great magnitude.

If the value of (3.45)is true then the test is said to have failed and control is transferred to the statement following the loop. Multiplication by g in (3.45) assures that loops in which the increment is (or becomes) negative will terminate properly.

The "FOR loop" (3.44) has two alternate forms

```
        FOR v = e BY g TO f <<...>>                 (3.46)
and     FOR v = e TO f      <<...>>                 (3.47)
```

In (3.47) no increment is given, so it is assumed to be one.

The iteration control phrase " DO I=J,K,N " also involves a control variable. In this case I,J,K, and N must all be of type INTEGER and may not be array elements or expressions. (These are the standard FORTRAN rules for DO loops). The following generates a standard FORTRAN**DO loop**:

```
DO I=J,K,N <<...>>                                  (3.48)
```

There is one exception to the rule that loops must be preceded by control phrases, namely the **compact DO-loop notation**

```
<<I=J,K,N;...>>                                     (3.15)
```

which generates a standard FORTRAN DO loop. This form permits compact notation for nests like

```
<<I=1,N1; <<J=1,N2; <<K=1,N3; A(I,J,K)=exp; >> >> >>.
```

The use of the compact DO-loop notation is controversial; some people feel that it obscures the loop control.

The MCSTRUFO FOR and DO loops apply only to blocks of statements, no to I-O lists. The usual FORTRAN implied DO should be used within READ or WRITE statements.

There remains one more type of loop to be discussed. This loop is sometimes referred to as the "**forever loop**". One writes the forever loop in MCSTRUFO by preceding a block with the phrase "LOOP".

```
      LOOP <<...>> REPEAT                           (3.49)
or simply LOOP <<...>>                              (3.4A)
```

The block is executed, and control is transferred back to the beginning of the loop. The optional phrase "REPEAT" in (3.49) is sometimes useful as a visual aid in locating the ends of deeply nested loops.

A reasonable question might be: "How do you get out of a forever loop? Or, for that matter, any of the loops?" One rather obvious way is to write

```
        GO TO :CHICAGO: ;
or      GO TO :THE HELL NUMBER 1:;
```

where the label :CHICAGO: is on some statement (or block) outside the loop. If a convenient label doesn't already exist, creating one for the sole purpose of jumping out of the loop can be annoying and distracting. For the case in which the jump is to the statement following the loop, the GO TO may be replaced by the MCSTRUFO **EXIT**; statement which is written

```
        EXIT;                                                    (3.4B)
or, with a conditional statement
        IF (e) EXIT;                                             (3.4C)
or      IF e <<...EXIT;>>                                        (3.4D)
```

In any MCSTRUFO loop, the occurrence of the statement "EXIT;" causes transfer of control to the first statement following the loop in which it occurs.

A companion to the "EXIT;" statement is the "NEXT;" statement, which is written

```
        NEXT;                                                    (3.4E)
or, with a conditional statement

        IF (e) NEXT;                                             (3.4F)
or      IF e <<...NEXT;>>                                        (3.4G)
```

The occurrence of a **NEXT**; statement (which is short-hand for "go to the **next iteration** of this loop") in any MCSTRUFO loop causes a trans fer of control to the beginning of the loop in which it occurs, incrementing the control variable (if any) before making the test for continuation in the loop.

In loops which test at both ends of the loop, only the test at the be ginning of the loop is made; tests at the end of the loop are made on ly when the end of the loop is reached. The tests of control variable in FOR and DO loops are considered to be at the beginning of the loop (For more detailed information refer to the flow-charts in chapter 9)

Any MCSTRUFO loop may be optionally preceded by a label. We will cal loops which are preceded by labels "labeled loops". Any EXIT or NEXT statement may be optionally followed by a label. Any labeled loop may contain one or more statements of the form

```
        EXIT :label: ;                                           (3.4H)
```

which transfers control to the first statement following the labeled loop. For example EXIT :ALPHA:; would transfer control to the statemnent following the loop which had been labeled :ALPHA:. This transfer of control takes place regardless of nesting, and thus provides a "multi-level" EXIT capability. The statement

```
        NEXT :label: ;                                           (3.4I)
```

transfers control in the manner described above for the NEXT statemenent

Suppose we have a nest of loops which search some arrays. The outer loop has been labeled :SEARCH:, and two of the inner loops have been labeled :COLUMN:, and :ROW:. Now we may write

```
        NEXT :ROW: ;
or      NEXT :COLUMN: ;
or      EXIT :SEARCH: ;
```

when the transfer involves more than one level of nesting, or

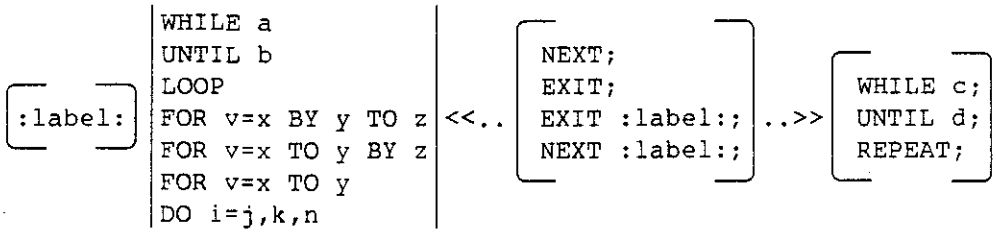
```
        NEXT;
or      EXIT;
```

when only one nest level is involved. Of course, the form

```
EXIT :label:;
```

may also be used to exit a single level if desired.

We can now summarize MCSTRUFO **loops** as follows:



where

a, b, c, d	are arbitrary logical expressions,
x, y, z	are arbitrary arithmetic expressions,
v	is a subscripted or non-subscripted variable of type INTEGER or REAL,
j, k, n	are non-subscripted INTEGER variables or INTEGER constants
i	is a non-subscripted INTEGER variable.
| |	indicates "choose one",
[ ]	indicates "optional"
...	indicates a (possibly null) sequence of statements.

3.5 Remote PROCEDURE

The construct **remote PROCEDURE** can be understood as a very extension of a FORTRAN statement function. It also may be seen as a local subroutine and is defined by a set of two statements:

```
EXECUTE :name of procedure:;                                    (3.50)
```

is used all times the procedure with the appropriate name is to be executet. The remote procedure itself is defined by

```
:name of prodedure: PROCEDURE <<                                (3.51)
      ....
      LEAVE :name of precure:;
      ....
>>
```

There may be multiple "calls" of the same procedure within one and the same program unit. Since in front of the procedure body no jumps are generated it is intendend to arrange procedures at the end of a program unit (prior to the end-statement). From this practice the name "remote" is derived.

Within the remote PROCEDURE all variables and definitions of the program unit are known (in contrary to a subroutine or function).

Using the same procedure name in different program units (main program, subroutine, function) within one run of the precompiler will cause errors for the compiler (undefined labels).

Example

```
            EXECUTE :NEXT CARD:;
            ...
            EXECUTE :NEXT CARD:;
            ....
:NEXT CARD: PROCEDURE <<
            READ (IN, 901,END=:EOF1:) ARRAY;
            VAL1=ARRAY(1);
            VAL2=ARRAY(2);
            LEAVE :NEXT CARD:;
    :EOF1: LEOF=.true. >>
```

(A) For formal parameters of subroutines or functions use l=0

→ SUBROUTINE BILD (H, W, TITLE)
STRING*0 TITLE

4 Additional data types

4.1 STRING data type

In MCSTRUFO there is no real **CHARACTER** data type as it is defined in FORTRAN77. The implemented **STRING** data type is a close simulation only.

In the following explanations string variables are denoted sx (with x beeing any character).

4.1.1 Declaration of strings

Avoid names of string variables being identical or part of subroutine names e.g.: avoid a string name TEXT, if there is a routine PLTEXT.

```
DEFINE STRING FUNCTIONS;

STRING*l s1,
         s2 (dimension),
         s3 /'inital'/,
         s4 (dimensions) /'initial'/;
```

The declaration "DEFINE STRING FUNCTIONS" defines the functions for string comparison as type LOGICAL. In the declartion of the strings the parts of the construct are:

l is the length declaration for the string. Do **not place** a blank between the * and the *l*. The macro will not work in this case. After the length declarator blanks are necessary.

dimension is one or at most **6** dimensioning numeric figures. (one dimension is used to simulate the type)

initial is any character sequence of lenght ≤*l*. A longer character sequence will cause compiler message "size", but works correctly. For dimensioned STRINGS only the first element will be initialized.

Example:

```
STRING*5 ALFA, BETA(3,4), GAMMA/'chars'/;
```

NOTE: For formal parameters a length declaration of 0 is used. The characters with hex representation FE and FF are not valid in strings since they are used by MCSTRUFO and compiler respectively.

4.1.2 Conversion to type STRING

If strings are used in **actual parameters** to subroutines or functions, You need the conversion function

```
STRING ('sequence of characters')
```

Example

```
CALL AXIS (x, y, 13.5, STRING('Note this') )
```

4.1.3 Assignment of strings

Of course it's only allowed to assign strings to variables or array's declared as strings. The simple assignment

```
s1 = s2
```

will not work. For the compiler the STRING data type is an array of dimension one - and there is no assignment of arrays in FORTRAN. So You need the funktions STRING and/or concatination with a NULL string to **assign strings**:

```
s2 = STRING ('sequence of characters')
s3 = s2 //
```

4.1.4 String functions

In the following ix denotes an integer variable. There are five functions usable in assignments:

concatination:

```
s4 = s2 // STRING('sequence of characters')
s5 = s2 // s4 // s2
```

length of string:

```
il= LEN (sx)
```

location of **substring** s2 within s1:

```
i2= INDEX (s1, s2)
```

integer equivalent of first **character** in string:

```
i5= ICHAR (sy)
```

character value of **integer**:

```
sx= CHAR (integer)
```

With the declaration DEFINE STRING FUNCTIONS the following functions are defined to be of type LOGICAL. They are used to **compare strings** in IF statements. The functions return .TRUE. for matching condition.

s1 ≥ s2	LGE (s1, s2)
s1 > s2	LGT (s1, s2)
s1 ≤ s2	LLE (s1, s2)
s1 < s2	LLT (s1, s2)
s1 == s2	LEQ (s1, s2)

To get the **implicit length** of a string, the notation L$-*name* may be used. This is useful eg in FORMAT statements or implied loops of WRITE:

```
STRING*17 S1, S2
 ....
OUTPUT S1, S1; (' S1= ', L$-S1 A1, ' S2= ', L$-S2 A1)
OUTPUT (S2(J), J= 1, L$-S2) ; (1X, 132A1)
```

5 Miscellaneous features

5.1 Multiple Assignment

It is sometimes useful to be able to assign the value of some expression or variable to several variables in a single statement. In MCSTRUFO one writes

```
/ p,q,r...,z / = e                                    (5.11)
```

where p,q,r,...,z are variables and e is an expression. The expression e is evaluated first and then assigned to the list of variables, proceeding from left to right. For example,

```
/ I, A(I,K), J / = SQRT(X/2.0);
```

produces the following FORTRAN statements:

```
I = SQRT(X/2.0)
A(I,K) = I
J = A(I,K)
```

It can be seen that some care must be taken to observe the order of assignments if type conversion is involved, or if (as in the example) the value of one variable affects assignment to another in the list.

This construct is not good for "rotation" of variables like Z ← Y ← X ← Z (it is controverse to that function).

(B) FTN66:
The length of a string-declaration is a minimum for separately compiled modules
→ declare it in common:
%'L$-name' = 'length+1'

5.2 I/O Abbreviations

Another instance in which the creation of a label can be annoying because it is used only once and contains no mnemonic information is the following:

```
READ (5,:NONSENSE:) i-o list;
:NONSENSE: FORMAT(format list);
```

In MCSTRUFO one may write

```
        INPUT i-o list; (format list);                    (5.21)
or      OUTPUT i-o list; (format list);                   (5.22)
```

whenever the input or output is to the standard FORTRAN input or output units (see "logical unit numbers").

5.3 Graphic representation of logical and relational operators

The common FORTRAN logical and relational operators may be shorthanded with some graphical characters or character sequences.

FORTRAN	MCSTRUFO	FORTRAN	MCSTRUFO
.AND.	&	.EQ.	= =
.OR.	|	.NE.	¬ = or < >
.NOT.	¬	.LE.	< =
		.LT.	<
		.GE.	> =
		.GT.	>

Since conflicts may arise betwenn the usage of the & (ampersand) as logical operator or designator of a branch label in actual argument lists (IBM extension), it should be written with surrounding blanks, if used as logical operator

```
IF (a & b) ....
```

but

```
CALL XYZ (argument, &900, &800)
```

It is to be mentioned that the use of branch labels in actual argument lists is not a good programming practice since the same effect is goaled by the use of a control variable and a computed GO TO following the call.

5.4 Assignment of binary, octal and hex values

Sometimes it is very awful to declare data for some pattern variables. Their meaning is a 'bit pattern' rather than a numeric value, so the source should reflect this. In MCSTRUFO one can write:

```
variable = B'binary value';                    (5.41)
variable = O'octal value';                     (5.42)
variable = X'hex value';                       (5.43)
variable = H'hex value';                       (5.44)
```

Mention that "variable" stands both for scalars or array-elements. The valid lengthes of the assigning strings depends on the length declaration of "variable".

If "variable" is declared as INTEGER, than the following is valid:

- The **binary value** in (5.41) may be coded of up to 32 1's or 0's. If there are less than 32 figures, the value is adjusted to the right.
- The **octal value** in (5.42) may be composed of up to 11 octal digits 0 ... 7, wherby the leftmost (if there are 11 digits) may only be 0 ... 3. If there are less than 10 digits, the value is adjusted to the right.
- In (5.43,4.44) the **hexadecimal value** may be coded of up to 8 hex digits 0 ... F. If there are less than 8 digits, the value is adjusted to the right.

In 4.41 ... 4.44 the characters B, O, X and H may be prefixed or suffixed with a length-declarator. (e.g. 8C'abcdefg' or C8'abcdefg'). The length of the following string is then derived from this declarator rather than from the string itself. Is the string itself to short to fullfil the length, then it is filled up (for C) at the right with blanks or (for B, O, X) at the left with binary zeroes. If the string is to long than it is truncated to the defined length.

The assignment	is interpreted as
X'999888777'	'999888777'
12X'1234567890'	'001234567890'
32B'10110011100011110000'	'00000000000010110011100011110000'
10O'777'	'0000000777'

5.5 Pseudo generic

The IBM FORTRAN IV H compiler knows the **generic** statement to allow automatic chossing function values depending on the type of arguments Most times this feature is used when **precision encrease** is needed. For this solely purpose MCSTRUFO offers a pseudo generic statement

```
FORCE D. P. FUNCTIONS;                                    (5.51)
```

This statement frees several macros, which convert single precision function calls to double precision ones and forces the corresponding arguments to double precision. Since macros are global to the whole sourcetext all programm units are affected by this mechanism . If one want to encrease precision in a simple manner he has to do:

- place the pseudo generic statement (5.51) in front of the sourcetext or at any convenient place
- insert an IMPLICIT statement into every routine (**one** of the following):
  - IMPLICIT REAL*8 (A-H, O-Z); (5.52)
  - IMPLICIT DOUBLE PRECISION (A-H, O-Z); (5.53)
- or one places appropriate DOUBLE PRECISION declarations into the routines, which also must contain DOUBLE PRECISION DOUBLE. (there will be a function with name DOUBLE).

To use these features, you must specify MACLEVL =1 (JCL procedure FORTRAN)

The generated macros than convert like

```
      COS (arg)
becomes
      DCOS (DOUBLE(arg))
```

5.6 ENCODE and DECODE

This very useful feature of some FORTRAN dialects is available in MCSTRUFO.

Data **encoding** means formatting of internal data to external representation (characters), wheras **decoding** means transformation from characters to (most times) binary values.

The statements are written as follows:

```
ENCODE (c, f, b ) list                                   (5.61)
DECODE (c, f, b ) list                                   (5.62)
```

c is an integer expression representing the number of characters (bytes) that are to be converted or that are to result from the conversion. This is analogous to the length of an external record

f is a format specifier (either a format label or an array containing the format declaration). If the format specifies more than one record errors occure.

b is the name of an array, array element or scalar. In the ENCODE statement this entity receives the characters. In the DECODE statement, it contains the characters that are to be translated to internal representation. B must me declared in a length equal or greater than the expression c. For example if c is 20 than b may be declared as INTEGER B(5) or LOGICAL*1 B(20).

Incorrect use of ENCODE or DECODE may cause the following **errors**:

- If the buffer b is specified to short, than FORTRAN **error 212** and 219 will occur (with unit-number 90).
- If the list is to long to fulfill the format and that is to be repeated, than the buffercontents is overlaid.

for FTN77 see routine NUMMER
WRITE ([unit=] char-var, [FMT=] format) list

5.7 PARAMETER statement

Global to whole module!

The MCSTRUFO **PARAMETER** statement is not totally similar to that of FORTRAN 77, since its scope is global to the total sourcetext rather than to one program unit. Its syntax is similar the FORTRAN 77.

```
PARAMETER pa = qa , .... , pz = qz ;                    (5.71)
```

Each symbolic name p (pa ... pz) is the name of a constant and becomes defined to the value of the constant q (qa ... qz). Once such a symbolic name is defined it may appear in that SOURCETEXT anywhere. The scope is more general than in FORTRAN 77; it extends to all following program units rather than only to that containing the PARAMETER statement.

Examples are

```
        PARAMETER  ALFA=27, SIZE=20, HEADER= 'string';
        ...
        DIMENSION ARRAY(SIZE)
        DO J= 1, SIZE  <<
           ARRAY (J)= ....
        >>
 901    FORMAT (T ALFA, HEADER, SIZE F5.2 )
        ...
        END
        FUNCTION ALFA (SIZE)
        ...
```

This will result in the following FORTRAN-statements

```
        ...
        DIMENSION ARRAY (20)
        DO 90010 J= 1, 20
        ARRAY(J)= ...
90010   CONTINUE
901     FORMAT (T  27, 'string', 20  F5.2  )
        ...
        END
        FUNCTION 27 (20)              ←
        ...
```

To minimize effects like that one noted with ← choose symbolic names carefully to avoid conflicts with normal fortran, e.g.:

- use lower case for symbolic names
- use symbolic names longer than 6 characters
- use special symbols to form symbolic names like #ALFA, @HEAP, IS_TO_GO etc.

5.8 Dummy arguments in calls of subroutines and functions

Sometimes the argument lists of calls are very long but one knows that only a few parameters are relevant to that case of execution. Than it is very convenient to write a short notation of the unused arguments. The called routine must be prepared to allow this special form, since the not written arguments are defined by a special value

```
CALL subroutine (,,,A)                                    (5.81)
CALL subroutine (, b, c,,,)                               (5.82)
Z= FUN ()
```

These examples are translated to FORTRAN as

```
CALL subroutine ($DUMMY, $DUMMY, $DUMMY, A)
CALL subroutine ($DUMMY, b, c, $DUMMY, $DUMMY, $DUMMY)
Z= FUN ($DUMMY)
```

and prior to the END statement **$DUMMY** is defined in a COMMON BLOCK with name **C$COMM1**. For INTEGERS there is a value I$DUMY in that COMMON BLOCK. The subroutine must test on this value if is not possible to know anything about the relevant parameters by other parameters or so.

6 User defined macros

6.1 String Replacement

A **macro definitions** is written in the following form:

```
%'pattern'='replacement'                                    (6.11)
```

Macro definitions are not statements and therefore need not be terminated by semicolons. (If you put one in it will be ignored.) Macro definitions are "free field" in the sense that you may write more than one definition on one line, or extend one definition to several lines (use continuation marker in this case).

The **pattern** and **replacement** parts of a macro definition are character strings in the sense described in section 2. Since embedded strings are permitted in macro definitions, the usual rules regarding the doubling of apostrophes apply.

The simplest kind of macro is one which contains neither parameters nor embedded strings. For example, one could write

```
%'ARRAYSIZE'='50'                                           (6.12)
```

after which all occurrences of the characters ARRAYSIZE in the program text would be replaced by 50. For example,

```
DIMENSION X(ARRAYSIZE); ... DO J=1,ARRAYSIZE <<...>>...
```

would produce the same FORTRAN program as if

```
DIMENSION X(50);... DO J=1,50 <<...>>...
```

had been written. This usage is completely similar to the PARAMETER definiton mentioned earlier.

Blanks are generally not significant when searching for occurrences of the pattern in the program text. For example, the macro

```
%'SIGMA(1)'='SIGMA1'
```

would match the program text

```
SIGMA     (1)
```

as well as

```
SIGMA(1)
```

In some cases it is desirable to require that one or more blanks be present in the program text in order that a match occur; this can be done by writing a single blank in the pattern part of the macro.

For example the macro

```
%'DUMP X;'='OUTPUT X; (F10.2);'
```

would match

```
DUMP X;
```

but not

```
Y = DUMPX;
```

Normally, the text generated by a macro is itself elegible for replacement by other macros, or

even by the the same macro that generated the text. A potential danger to keep in mind in writing macros is the "macro explosion" or infinite macro loop phenomenon.

Suppose that we define a macro like

```
%'EQ'='EQUIVALENCE'                                    (6.13)
```

to provide a convenient abbreviation. Assuming that the first instance of EQ encountered in the program text is not something like A.EQ.B whichwould be replaced by A.EQUIVALENCE.B), the first instance of EQ in the program text would be replaced by EQUIVALENCE the way we intended.

Then, since the processor "re-scans" all generated text, the EQ in EQUIVALENCE would be re-matched and replaced again resulting in EQUIVALENCEUIVALENCE, and again, resulting in EQUIVALENCEUIVALENCEUIVALENCE, and so on until we run out of space, get an error message, and stop. The simplest way to avoid a macro explosion is to enclose that portion of the replacement part not to be re-scanned in quotation marks ('). We could rewrite (6.13) as

```
%'EQ'='"EQUIVALENCE"'                                  (6.14)
```

and avoid the macro explosion; if we wrote

```
%'EQU'='"EQUIVALENCE"'                                 (6.15)
```

we could also avoid matching the relational operator .EQ. .

If we wrote %'AB'='CD' and %'CD'='AB' we would cause the processor to go into an infinite loop, not unlike inadvertent loops that are possible in any programming language. Unless you specifically want the replacement to be re-matched, it is a good idea to enclose the replacement part in quotation marks. Thus %'AB'='"CD"' and %'CD'='"AB"' would not cause this problem.

The characters appearing inside quoted strings are not elegible for macro replacement. For example, the statement

```
OUTPUT SETS; (' EQUIVALENCE SETS',10I5);
```

would be unaffected by (6.13).

6.2 Parameters in Macros

The pattern part of a macro definition may contain up to nine formal (or "dummy") parameters, each of which represents a variable length character string. The parameters are denoted by the symbol #. For example,

```
'EXAMPLE#PATTERN#DEFINITION'                                    (6.21)
```

contains two formal parameters. The formal parameters are "positional That is, the first formal parameter is the first # encountered (reading left to right), the second formal parameter is the second # encountered and so on. The corresponding actual parameters are detected and saved during the matching process. For example, in the string

```
EXAMPLE OF A PATTERN IN A MACRO DEFINITION                      (6.22)
```

(assuming (6.21) is the pattern to be matched), the first actual parameter is the string "OF A", and the second actual parameter is the string "IN A MACRO". The parameters are saved in a "holding buffer" until the match is completed.

After a macro is matched, it is "expanded". The expansion process consists of deleting the program text which matched the pattern part of the macro and substituting for it the replacement part of the macro.

The replacement part may contain an arbitrary number of occurrences of formal parameters of the form #i (i=1,2,...,9). During expansion, each formal parameter #i of the replacement part is replaced by the i-ths actual parameter. A given formal parameter may appear zero or more times in the replacement part. For example, the pattern part of the macro definition

```
%'PLUS #;' = '#1=#1+1;'                                         (6.23)
```

would match the program text

```
PLUS A(I,J,K);                                                  (6.24)
```

During the matching process the actual parameter A(I,J,K) is saved in the holding buffer. Upon completion of the matching process (that iswhen the semicolon in the program text matches the semicolon in the pattern part), the "expansion" of the macro takes place, during which the actual parameter A(I,J,K) replaces all occurrences of the corresponding formal parameter, producing

```
A(I,J,K)=A(I,J,K)+1;                                            (6.25)
```

Note that the single formal parameter #1 occurs twice in the replacement part and therefore the single actual parameter A(I,J,K) occurs twice in the resulting string.

The program text which may be substituted for the formal (dummy) parameter is arbitrary except for the following restrictions:

- It may not be text containing an end-of-statement semicolon. This restriction prevents "runaway" macros from consuming large parts of the program.
- Parentheses and brackets must be correctly matched (balanced). This facilitates the construction of macros which treat expressions as indivisible units.
- Quoted character strings are considered to be indivisible units. If the opening apostrophe of a character string is part of the actual parameter, then the entire string must be within the actual parameter.

6.3 Advanced Uses of macros

It is not necessary to understand this section in order to make effective use of user-defined macros. Much of the available power of the processor could be lost, however, if this section is not clearly understood. First, we define some terms: a string in the program text which matches the pattern part of a macro definition will be called an "instance" of the macro. For example, (6.24) is an instance of (6.23) which expands into (6.25). "No-rescan text" will mean that portion of the replacement part of a macro which has been enclosed in quotation marks. For example, all of the replacement part of (6.4B) is no-rescan text. "Squashed text" will refer to program text in which all multiple blanks have been replaced by a single blank, and from which all comments have been deleted.

The following is a brief description of the **matching process**: A line (card) is read, squashed, and put into a buffer called the "expand buffer", where beginning with the first character of the buffer is matched against each macro in turn. When an instance of a macro ifound, the macro is expanded, and the matching process begins again with the first character of the newly expanded text. If no instance of any macro is found, then the first character of the expand buffer is sent to an output buffer and the matching process begins again with the next character in the expand buffer. If the first character of the expand buffer is a quotation mark, then all the characters following the quotation mark, upt to the next quotation mark are sent to the output buffer. That is, all no-rescan text is sent immediately to the output buffer when it is encountered in the first position of the expand buffer.

A blank in the pattern part of a macro MUST be matched by a blank in the expand buffer for a successful match of the macro. If, on the other hand, the character in the pattern part of a macro is not a blank, a blank in the expand buffer is ignored. This convention allows the programmer to "force" a match to a blank in the incoming text or to ignore the blank. Since multiple blanks are squashed on inpu(exdept in quoted strings), failure to squash the blanks in the pattern and replacement part of macros (which are qu. strings) would mean that patterns containing multiple blanks could not be matched. Therefore, multiple blanks are squashed from the pattern and the replacement parts of macro definitions, except for embedded strings. Stated in terms of the input text: blanks in the input text are ignored unless a "forced" match" to a blank is being made, in which case a single blank in the pattern part of the macro effectively matches an arbitrary number of excess blanks in the input text.

Macro definitions are "stacked". That is, the most recently defined macro is the first candidate for matching. (Since the standard macros are read in first, the user's macros are matched before any of the standard set.)

Suppose that, for debugging purposes, we want to "dump" some variable (say X) every time it is assigned a value in the program. That is, every time any assignment is made to the variable X, we want to write on the FORTRAN output file something like

```
X ASSIGNED xxx
```

where xxx is the value which is being assigned to X in the program at execution time. We could do this by writing a macro like

```
%';X=#;'='";X"=#1; OUTPUT X; (''  X ASSIGNED '',E12.6);'    (6.31)
```

after which every assignment to X would cause X to be dumped. NOTE:

- The first character in the pattern part is a semicolon. This was done to prevent matching assignments like SUMX=expression.
- We have doubled the apostrophes which enclose the Hollerith data in the format-list part of the replacement.

- We have enclosed the assignment statement in the replacement part in quotation marks to prevent it being re-expanded by the same macro.

While (6.31) is a useful macro, it suffers from the following fault: If we want to dump another variable (say Y) we must write another macro which is identical to (6.10) except that the X's are replaced by Y's. This suggests that we could write a macro which would create macros to dump any variable in the program. This is in fact possible, as follows:

```
%';TRACE#;'='%'';#1=##;''='''";#1"=##1;
OUTPUT#1;(''''  #1 ASSIGNED '''',E12.6);'''                    (6.32)
```

Now, if the statement TRACE X; is put in the program, it will cause a macro like (6.31) to be generated. If we write TRACE Y; it will cause another macro to be generated which is like (6.10) except that the X's would be Y's. Careful examination will reveal that the replacement part of (6.32) is in fact the whole macro (6.31) with the following changes:

1 All occurrences of X have been replaced by #1.

2 We have doubled the apostrophes which delimit the pattern and replacement parts of (6.31) because they are now inside a quote string. The double apostrophes in (6.31) have been changed to quadruple apostrophes for the same reason.

3 It was necessary to generate a "#" in the replacement part. To do this we wrote "##" so that it could be distinguished from a formal parameter.

What we have illustrated here is a macro which generates macros. We leave to the reader's imagination the variety of uses to which this facility can be put.

6.4 Implementation of user defined macros

User defined macros should be tested in the format of the source input (default = fix format). To test them you place them in front of a testing program.

When the input is specified in FIX FORMAT than place the macro text only in columns 6 ... 72 and place the continuation marker in columns 6 for each continuation line of the macro (otherwise the precompiler will insert a ; in front of that lines).

```
1....6.............. input line columns ..........72

      %'TEST #' = 'OUPUT #1;(' ', 10G12.6)'
      %'LONG #, #, #' =
     +    'LONG MACRO WITH #1 AS FIRST, #2 AS SECOND
     + AND #3 AS THIRD PARAMETER'
```

7 Conditional compilation

It is often the case that several versions of a program are maintained that differ only in certain well-defined respects. One possible technique for merging many versions into a single program is to use MCSTRUFO macro calls in place of the code that varies, and to define set of macros that produce the different versions. The macro calls that are not being used for a particular version may be enclosed in commentsymbols. The production of a particular version then becomes a matter of removing the comment symbols from a subset of the macro calls, and enclosing another subset in comment symbols. In this way, the subset that is enclosed in comment symbols becomes a form of documentation

When the number of calls which must be altered in this way is not small, the macro technique becomes cumbersome. For these cases MCSTRUFO has a "conditional compilation" feature, which allows the generated code to be changed in many places by changing a small number of macro definitions.

The conditional compilation feature is a mechanism whereby a particular piece of program can be ignored (that is, treated as a comment) or not, depending on how a related macro has been defined. The "piece" can be as small as a single character, or as large as an entire program, and will hereafter be called a program "segment". To delimit a segment, the programmer decides on a unique string to begin the segment and another to end it. Any number of segments may be deli mited in this way, and whether or not the segments are compiled will depend on the macros that define the delimiters.

For example, suppose that certain segments are to be generated only in a "multiple-precision" version of a particular program, and other segments only for a "double-precision" version. One might use the strings "/MULTIPLE/" and "/ENDMULT/" to delimit the multiple-precision segments, and the strings "/DOUBLE/" and "/ENDDOUB/" to delimit the double-precision segments. A typical use might be:

```
/MULTIPLE/ DIMENSION A(10);    /ENDMULT/
/DOUBLE/  DOUBLE PRECISION A; /ENDDOUB/
```

To determine which segments are to be compiled, the strings serving as delimiters should be defined as macros before any of the segments are used. The macros for the start-of-segment strings should have either the keyword "GENERATE" or the keyword "NOGENERATE" as the replacement and the macro for the end-of-segment strings should have the keyword "ENDGENERATE" as the replacement text. For example, to specify the multiple-precision version of the program the following definitions should be used:

```
%'/MULTIPLE/'='GENERATE' %'/DOUBLE/'='NOGENERATE'
%'/ENDMULT/'='ENDGENERATE' %'/ENDDOUB/'='ENDGENERATE'
```

.keep To generate the double-precision version, the GENERATE and NOGENERATE keywords are interchanged.

Bear in mind that the strings to be used as delimiters will be macro patterns. They should be chosen so as to avoid unintentionally matching program text not involved in conditional compilation. Note that the replacement parts of both /ENDMULT/ and /ENDDOUB/ is the same keyword ENDGENERATE. The same delimiter (unique string) may be used as the terminating delimiter for all conditionally compiled segments. They were chosen to be different in the example in order to improve readability.

Conditionally-compiled segments may be nested within one another. This feature is particularly useful when the conditions are not mutually exclusive; whenever an outer segment is not being generated, all inner segments will also not be generated regardless of how the macros for the inner segments were defined.

ABSICHTLICH LEER GELASSEN

8 Runtime control

8.1 Control statements

MCSTRUFO control statements may appear anywhere within the program. Each control statement begins with a % in column ≥1 and should not contain embedded blanks or program text. Comment may be added after a blank following the relevant information. As an example, %SPACE 3 produces three blank lines in the output list.

SPACE # produces n (0 ... 9) **blank lines** on the listing. For #=0 an overprint occures for the preceeding and the following line If n is not a figure or > 9 than SPACE 1 is assumed.

EJECT advances the listing to a **new page**.

LIST (initial on) enables source listing.

NOLIST disables source listing

INCLUDE ... (name) From the **copylibrary** (see definitions of the JCL procedure FORTRAN) the library member with the name "name" is included. This is very convenient for introducing standard declarations, definitions of COMMON blocks or user macros.

FORTRAN switches the input to FORTRAN-mode (initial is MCSTRUFO). This statement may be abbreviated with "F" While in this mode, lines are transferred to output without any processing. Avoid switching to FORTRAN mode while macro are pending (e.g. within a block or a remote procedure), since than the rest of the expanded macro is genereated, when MCSTRUFO mode is resumed.
This control statement is **not valid in free format**.

MCSTRUFO Switch back to **MCSTRUFO** mode. (Initial mode). This statement may be abbreviated with "M"
This control statement is **not valid in free format**.

I# Indent the MCSTRUFO listing # places per nest level where # = 0,1,2,...,10 (Initally 3) (Leading blanks are automatically suppressed when n>0 so that "ragged" programs will be "straightened" by MCSTRUFO.) The picture of the listing depends on the source-format (fix-format or free-format).

A# Annotation mode switch. For #=1 the MCSTRUFO sourcetext is interleaved with the generated FORTRAN text. Default: 0

H# Defines the generation of output strings:

- **0** strings are generated as Hollerith ..H.....
- **1** strings are generated with apostrophs (default)

D# printing of **pattern part** and **replacement part** of macros:

- **0** turns off printing initiated by D1 (default).
- **1** prints pattern- and replacementpart of a macro, at the time it is stored.

T# **trace** of expansion process

- **0** turns off printing initiated by T1 or T2 (default).
- **1** trace only those macros having @MT in its replacement
- **2** trace all macros.
- **3** trace all macros + internal operators

4 trace all macros + internal operators + input to proces

ENDMEMBR This control statement may be inserted in a member of the copylibrary or macrolibrary to terminate the input processing prior to the real end of the member. Behind this control, explanations or other information may be placed, which is not relavant to MCSTRUFO. This control is not valid in normal source input (sequential).

8.2 Runtime options

The options of MCSTRUFO are defined by the **FORTRAN procedure** parameter **PREOPT** (options of the precompiler), which consists of 6 bytes:

`x.....` **format**-option of source-text (FMTTYP) and trace; right halfbyte is fmttyp:

- **1** fixed FORTRAN-format (default)
- **2** free format (macros), text is untouched

left halfbyte is trace-option, if LSOURCE >3 (this feature not yet used)

`.x....` **source listing** (LSOURC)

- **0** no source-listing is produced
- **1** source listing produced (default)
- **2** macro listing is produced
- **3** = 1 and 2

`..x...` **comment**-option (COMMNT)

- **0** comments are skipped (default)
- **1** comments are transferred directly to output

`...x..` **caps lock** option

- **0** no operation
- **1** **lower case output** chars are translated to upper case (default). **Attention**: this also happens to strings!

`....x.` **deck**-option

- **0** output to compiler-dataset (default)
- **1** output to **punch dataset** (for later use) In this case the precompiler will end with return code of 8 to avoid impossible compilation.

`.....x` **debugging level** (DBGLVL) see next section
x... any number from 0 to 9 (default: 0)

8.3 Debugging aid

MCSTRUFO offers a very convenient method for debugging programs. This feature allows special **comments** in the source text (only in fix format - sorry) to be converted to executable statements depending on the optionbyte 6 (see chapter 7.2).

This special comment is written as standard FORTRAN comment (starting with a C in column 1), but the comment text (starting at column 7) is an executable statement. This may be continued, if on the next line column 6 is not blank. The string **CDBG** is a constant.

```
CDBGn statement text starting at column 7
CDBGn* continue with column 7... (mark column 6)
```

n may be a number 0 ... 9, which denotes the "debugging level".

For explanation assume a program which contains the following debugging comments:

```
CDBG0 OUTPUT X1, X2, Z3; (3G10.3)                          (8.31)
      :
CDBG2 OUTPUT ALFA; (20G6.3);                               (8.32)
      :
CDBG1 OUTPUT I, ALFA(I); (I6, F10.3);                      (8.33)
      :
CDBG2 OUTPUT K, L, M, ALFA(17); (3I5, F10.3);              (8.34)
      :
```

Now when processing this program with optionbyte-6 =0 (default), only (8.31) is converted to executable (blank out columns 1...5). The remaining debugging-comments (8.32 ... 8.04) are still comments.

When processing the same program text with optionbyte-6 =1 than (8.31) and (8.33) are converted to executable, (8.32) and (8.34) are still comments.

When processing is done with optionbyte-6 =2 or greater than all statements (8.31 ... 8.04) are converted to executable form.

8.4 Error messages on output

These messages are printer on the output list of the precompiler.Due to the macro process (thich is pending some time) the massages may be "late" some lines. But their relation to the program text mey be found quite well.

The messages are arranged in alphabetical order for easy reference. The explanations for each error are arranged more or less in order of the most probable cause. The severity of the error is listed in ().

BLOCK(S) UNDERFLOW (8)

Excess end-of-block >> are defined (or to few block openings <<). The superfluous end-of-block are removed from source-text. Look at the indentation to find the block openings.

CONTROL VAL.WRONG -> DEFAULT (4)

A control card which required a number had invalid digits or a number that was outside the allowable range for the control card. For defaults see section 7.1

E N D STATEMENT MISSING (4)

At end of file on source input there was no END statement. It may be misspelled or not written in proper form ("END " or "END;"). A correct form is inserted.

ENDMEMBR-STATEMENT ONLY VALID DURING LIBRARY INPUT (8)

The reading of a library-member may be terminated prior to the end of the text with an %ENDMEMBR statement. Beyond this there may be explaining text or so. This control is not valid in normal source input (sequential).

EXPANSION BUFFER OVERFLOW (16)

Macro explosion. (see section 5.1) User defined macro error. User defined macro(s) expansion too large for buffer.

ILLEGAL CONTROL STATEMENT (8)

In freeformat the statements %F and %M are not valid - in this input format the processor is not switchable to "poor FORTRAN". Hence all program input - MCSTRUFO or pure FORTRAN - is threaded through it.

ILLEGAL MACRO PATTERN (4)

The pattern part of a macro definition may not be a null string.

ILLEGAL PARAMETER DIGIT (4)

The character following a # in the replacement part of a macro definition is not a digit from 1 to N, where N is the number of formal parameters in the pattern part of the macro.

INPUT BUFFER OVERFLOW (16)

Unclosed character string. Check ISN column to locate the beginning on the string. Statement too long. Wrong type of inputformat specified (freeformat defined for pure FORTRAN-input).

LABEL PATTERN OVERFLOW (8)

There are so much labels to be generated, so that the range 90000 to 99999 is overflowed. The labelgenerator starts again with 90000. This may only occur when processing very large programs.

LABEL STACK OVERFLOW (4)

User defined macro error Nesting level exceeds 50.

LABEL STACK UNDRFLOW (4)

An excess of right brackets or a missing left bracket. This message will usually be accompanied by a negative nesting level in the MORTRAN listing. Unlike the MISSING RIGHT BRACKET message, this message is issued when the extra right bracket is encountered.

LIBRARY I/O ERROR (MACROLIB OR COPYLIB) (12)

An input/output error happened on the specified library. Check the JCL-messages to identify the problem (library destroyed ...).

LIBRARY MEMBER xxxxxxxx NOT FOUND IN LIBRARY (12)

The name specified in the INCLUDE-statement don't exist in the copy library. It may be misspelled or written in lower case.

LIBRARY MEMBER xxxxxxxx TRIED TO INCLUDE ITSELF RECURSIVELY (12)

A chain of INCLUDES led to a library-member, which is also in the chain (and not terminated yet now). The chain of includes is listed to find the error. Redefine the module hierarchy to avoid this error. This error may occur also with the additional text

BY ITS ALIAS yyyyyyyy.

MACRO BUFFER OVERFLOW (16)

Missing right colon on alpha label. Too many labels, or excessively long labels. Too many user-defined macros for available space.

MACRO EXPANSION LOOP (8)

A macro explosion happens. After 500 expansions of the same macro the expansion process is terminated. Error in user macro.

MACRO STACK OVERFLOW (4)

User defined macro error Nesting level exceeds 50.

MACRO STACK UNDRFLOW (4)

An error in a user-defined macro.

MAXIMUM OF 8 NESTED INCLUDES POSSIBLE (12)

The chain of INCLUDES goes deeper than to level 8. Redefine the modul hierarchy to fix the problem.

MEMBER NAME NOT FOUND ON INCLUDE-CARD (8)

Error on control-statement. Either the member name is not enclosed in paranthesis or it is absent. The request is ignored.

MORE THAN 19 CONTINUATION LINES GENERATED FOR OUTPUT (8)

The FORTRAN compiler does accept only up to 19 continuation lines. The expansion of the sourcetext produces more than this. Restructure the sourcetext to avoid this problem (shorten the statements, ...)

OPEN BLOCK(S) -> CLOSED (4,8)

This message is issued at the end of a MCSTRUFO module. Check the nesting level (indentation) on the MCSTRUFO listing for error location.

OPEN COMMENT -> CLOSED (4)

The module ended whithout the closing quotation mark for a comment. Check the ISN at the left side of the listing to find the opening ".

OPEN STRING -> CLOSED (4)

The module ended without the closing apostrophe for a character string. Check the ISN at left side of listing to find the opening quote.

PARAMETER BUFFER OVERFLOW (16)

The sum of the lengths of the actual parameters exceeds the space available for them.

READ/WRT SYNTAX ERROR (8)

The open paranthesis is not found on the same line as the keyword READ or WRITE, or the closing paranthesis is not found on same line. This scan is done to substitute an apostrophe in direct access R/W- statements (because it is not a string delimiter). Rewrite the source code to fulfill this coding rule (see section 2).

UNDEFINED CONTROL STATEMENT (4)

The undefined control statement (for valid ones see section 8.1) is skipped.

UNABLE TO OPEN COPYLIBRARY (12)

This may happen, if in the FORTRAN-procedure the DD-statement for the copylibrary is nullified or specifies an inproper dataset. Look at the JCL-messages to identify the error.

SUMMARY MESSAGE

At the end of the MCSTRUFO processing the number of errors encountered and the highest severity code is printed. The **severity code** is raised to a minimum of 8, if there are more than 10 errors of level 4 - to avoid a suspect compilation of the generated code.

8.5 Messages in the JCL log file

These messages all have a severity code of 16. They may be not printed on the MCSTRUFO output, since the precompiler may be not yet operating. They are found in job log files. Sometimes they mey be accompained by system error messages.

8.5.1 Insufficient core available

There is to less storage defined to establisch MCSTRUFO. Increase REGION parameter or introduce one, if there is none in the JCL or the JCL procedure:

```
//       EXEC .....,REGION=512K
```

8.5.2 Unable to open xxxx

xxxx may be "print file", "macro library", "source dataset" or "punch file". One of these files should be used (due to runtime options or else), but there definition is not present or incorrect.

Correct the JCL. A correct definition of the files may be found in the JCL procedure or in the "programmers guide" HB 400.90.30.

If the punch file is not available, no output to the compiler is generated, but MCSTRUFO is working.

8.5.3 DD-translatelist wrong

At dynamic invocation a DD translat list to define the files is used. If errors are detected in this list, than the defaults specified in the program FOM#PRE1 are use. This may cause subsidiary errors.

9 Implementation of MCSTRUFO

9.1 OBRZ modifications to MORTRAN2

- all planned I/O is done by an assembler-routine FOM#PRE1, which gives several advantages in our environment:
  - fast, since more adequate to string I/O than FORTRAN
  - usage of system parmfield for options
  - introduction of INCLUDE control statement
- block delimiter is changed from <...> to <<...>> to allow < and > to become relational operators
- avoid interpretation of apostrophe in direct access read/write statement as string-delimiter (new function introduced to scan that).
- program input normally is in fix format like common FORTRAN rather than free as for macros (the termination semicolon is generated). This gives the possibility to detect some errors:
  - open strings (at end of statement)
  - unmatched parantheses (at end of statement)
  - unclosed "-comments (at end of module)
  - unclosed blocks (at end of module)
  - skip to new page at end of each program unit (module)
- free format is also available, but much of these tests are not possible in this case.
- by an option lower case characters are translated to upper case on output (**also in strings**).
- by an option the output may be punched rather than left to the FORTRAN-compiler
- additional form of comment (from ! to the end of the line)
- strings in output my be generated as nH..... or with apostrophes.
- On end-of-file the string '*EOF*' is appended to the source text, which can be used by the macros.
- macro trace-facility %Tx is expanded to allow trace of macro-operators and processor-input.
- new macro operator I# sets the internal counter to the value of the #-th argument (if numeric). Additional macro operators for the counter for * (multiplication), / (division) and \ (remainder of division).
- the character set is now defined in the BLOCK DATA rather than in the macro input. Also the fundamental macros are defined there.

9.2 Execution of MCSTRUFO

9.2.1 Explicit job control

```
//        EXEC PGM=FOM#PRE,
//             PARM=110101,                                RUNTIME OPTIONS
//             REGION=512K                                 IS ENOUGH
//STEPLIB   DD DISP=SHR,DSN=TWMON.STEPLIB.LOAD
//MACLIB    DD DISP=SHR,DSN=TWMON.PARMLIB(FOM#MC0)        MACROS LEVEL 0
//COPYLIB   DD DISP=SHR,...                               FOR INCLUDE
//FT93F001 DD SYSOUT=*                                    LIST
//FT07F001 DD SYSOUT=B,DEST=ANYLOCAL                      PUNCH / DS
//FT98F001 DD .....                                       TO COMPILER
//FT97F001 DD DDNAME=SYSIN                                INPUT TO MCSTRUFO
```

9.2.2 OBRZ JCL prodedure FORTRAN or FTNFILES

```
//        EXEC FORTRAN,                  defaults
//             PROCESS=C,                  T          PRECOMPILE & COMPILE
//             NAME=XYZ,                   NONAME     NAME OF PGM
//             PREOPT='xxxxxx',            110101     RUNTIME OPTIONS
//             COPYLIB='DATASET',          ''         FOR INCLUDE
//             MACLEVL=y                   0          COMMON MACROS
//FORSYSIN DD *   input to precompiler                 INLINE (OR DATASET)
```

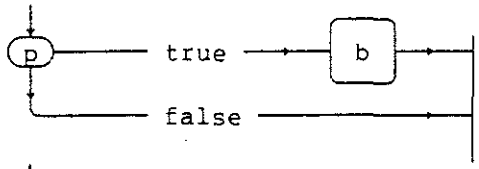
9.2.3 Dynamic call of the precompiler

When the precompiler is entered by use of LINK or CALL macro (assembler) a second parameter list giving a **DD translate list** may be defined. This specifies all or part of alternate DD names for the files of MCSTRUFO. This feature is documented in the "programmers guide" HB 400.90.30.

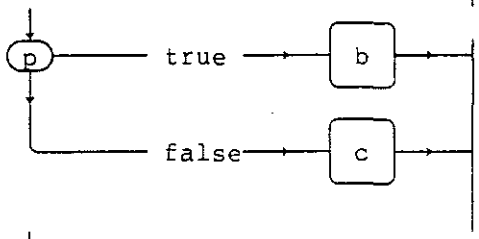
9.3 Flow charts and code generation

9.3.1 IF, ELSE, ELSEIF

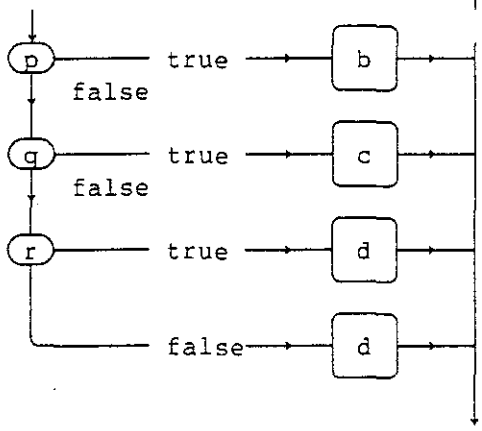
```
IF p << b >>
```




```
IF p << b >>
   ELSE << c >>
```

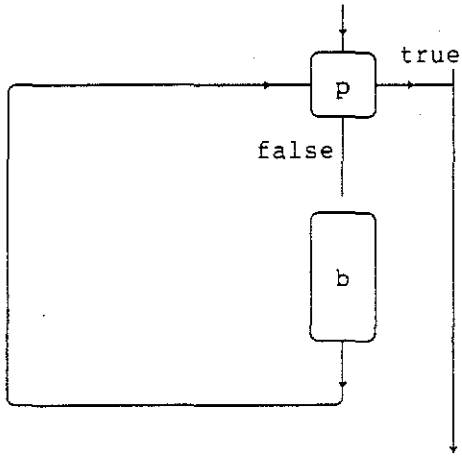



```
IF p << b >>
   ELSEIF  q << c >>
   ELSEIF  r << d >>
   ELSE      << e >>
```



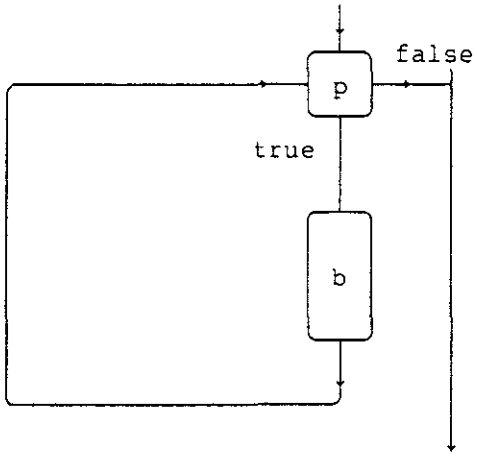
9.3.2 UNTIL

```
UNTIL  p <<  b   >>
```



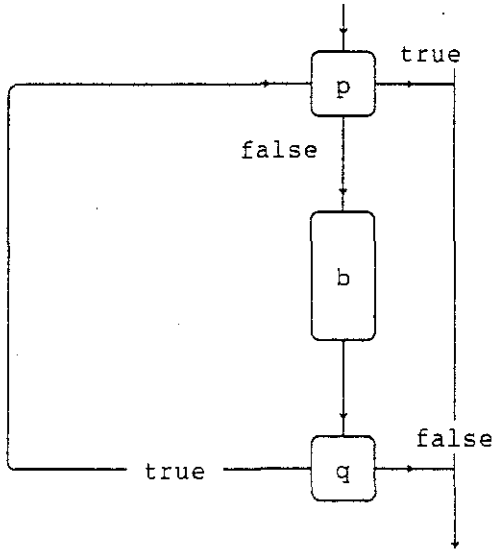
9.3.3 WHILE

```
WHILE  p <<  b   >>
```



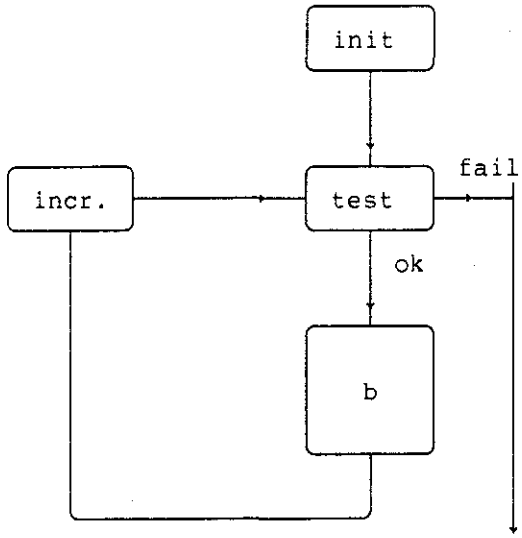
9.3.4 UNTIL ... WHILE

```
UNTIL p <<   b   >> WHILE q;
```



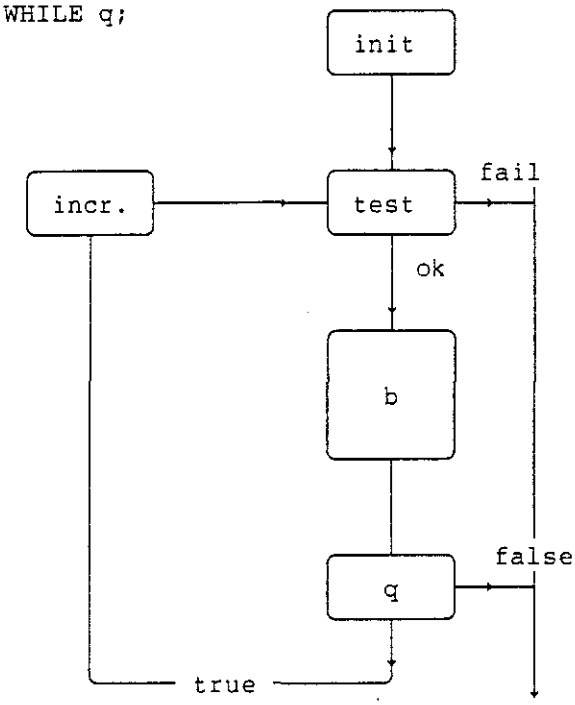
9.3.5 FOR

```
FOR v = e TO f BY g << b >>
```

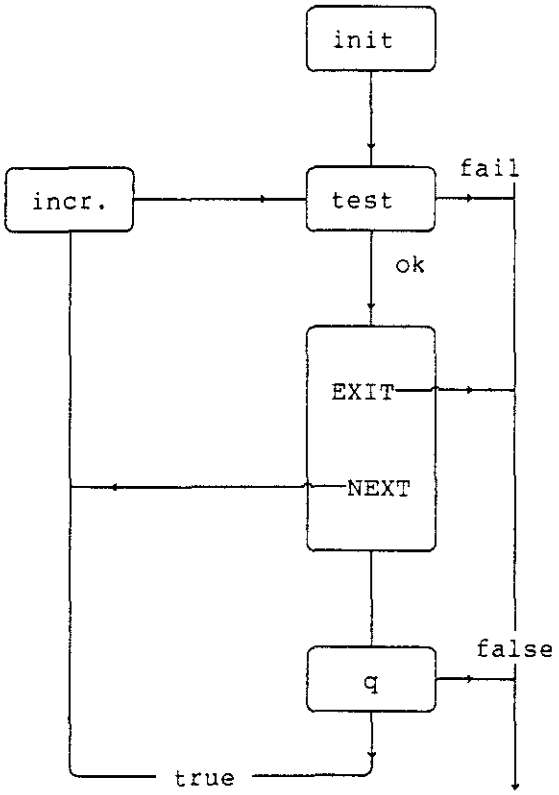


9.3.6 FOR

```
FOR v = e TO f BY g << b >> WHILE q;
```

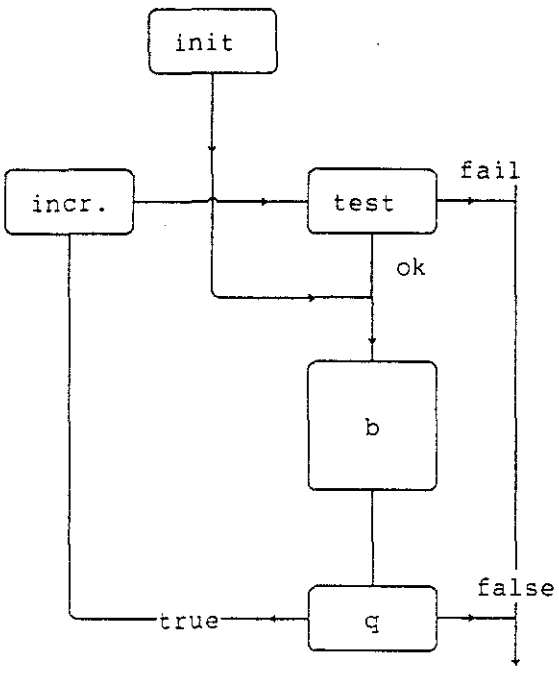


```
FOR v = e TO f BY g << .. EXIT; ... NEXT; >> WHILE q;
```

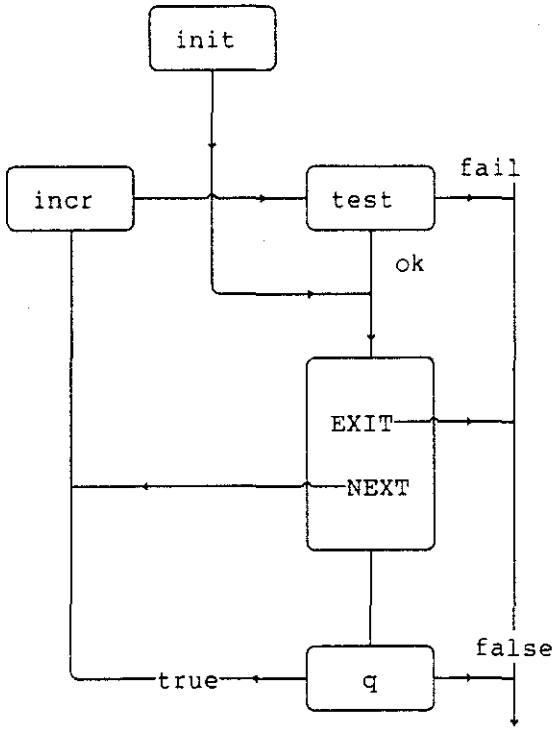


9.3.7 DO

```
DO I= J, K, N << b >> WHILE q
```



```
DO I= J, K, N << .. EXIT; .. NEXT; .. >> WHILE q;
```



9.3.8 Remote procedure

```
        EXECUTE :name:;
           :
        EXECUTE :name:;
           :
:name:  PROCEDURE <<
           part of procedure body
           LEAVE :name:
           part of procedure body
        >>
```

will produce

```
        ASSIGN alfa TO Idelta
        GO TO gamma
alfa    CONTINUE
        :
        ASSIGN beta TO Idelta
        GO TO gamma
beta    CONTINUE
        :
        :
        error-message and STOP 16 (when run into procedure)
gamma   CONTINUE
        part of body of procedure
        GO TO zeta
        part of body of procedure
zeta    GO TO Idelta (alfa,beta,gamma)
```

9.3.9 Non numeric assignment

```
alfa= beta'string'

where beta   nX, nB, nO or nil          will produce:

       NAMELIST /zeta/ alfa
       CALL NAMEL2 ('zeta', 'alfa(', Ileng, Iadres)
       CALL $MOVE  (alfa, 'beta■', 'string■', Iadres, Ileng)
```

The namelist assures that no invalid (too long) transfere is done. The mark ■ is actually a hex FE.

9.4 Runtime support for the precompiler

MCSTRUFO not only generates pure FORTRAN (with some IBM extensions like NAMELIST) but also uses runtime support - as a good precompiler does. This makes it possible to check things not available to the macro process. MCSTRUFO makes use of routines, functions and a COMMON block. It also uses special LUN's.

9.4.1 CHAR *integer*

This function checks the *integer* value to be in the range 0 .. 255 and defines a word as answer. The first byte of this word holds the charcter. The second byte is filled with the string terminating charcter. The remaining bytes are set to zero.

9.4.2 CONCT$ *string1, l-string1, string2, string3*

This routine concatenates string2 and string3 and assigns the result to string1. The maximum length of the result is l-string1.

9.4.3 COREIO *buffer, length*

This routine from SHARE transfers the control for the next following READ or WRITE to the *buffer* with *length* rather than to a real device buffer. This routine is used to simulate ENCODE and DECODE.

9.4.4 ERRTRA

This is the common trace back routine of the FORTRAN runtime system. In the OBRZ environment it is slightly modified to give a more clear presentation of the trace back.

9.4.5 Lexical compare functions LGE, LGT, LLE, LLT, LEQ

These functions are declared to be of type LOGICAL. They all use two arguments s1 and s2. The result of the function is TRUE for matching the **lexical comparison** of s2 and s1. This comparison is done in the **ASCII collating sequence** of the characters! For example LGE(s1, s2) is TRUE for s2 lexically greater or equal s1.

9.4.6 NAMEL1 *'namel', 'variable', length, address*

This routine gets attributes of the *variable* from the NAMELIST *namel. Length* is the maximum length in bytes for that variable or vector. *Address* gives an index within it, if 'variable' is indexed.

9.4.7 $MOVE *target, type, source, address, length*

This routine transfers the value of the source string to a target variable. Their maximum length in bytes is defined in *length*. The starting index (if taget has an idex) within the vector *target* is defined by *address*.

Type may be Binary, HeXadecimal or Octal and defines the kind of transformation to be applied to the source string. The type Character to transfere strings is not used here (see CONCT$). There may be an explicit length specification (numeric value) following the designating character.

9.4.8 $COMM1

C $RUN

This COMMON BLOCK is linked to the program at runtime and holds values for **undefined variables** with the names **$DUMMY** and **I$DUMY** as well as the value **I$ESTR** with the **end of string** character x'FE' in its first byte. There also is spare place in this COMMON.

9.4.9 Logical unit numbers

The macros INPUT and OUTPUT assume the **logical unit number**, 5 and 6 respectively. For the simulation of ENCODE and DECODE the LUN 90 is used.

10 Input and output description

10.1 Sample program and JCL

```
//  EXEC FORTRAN,
//       PROCESS=C,                      COMPILE ONLY
//       NAME=SIEVE,                     NAME OF PROGRAM
//       PREOPT=110111,                  .....1 TO PUNCH
//       COPYLIB='ST#K.LIB.CNTL',        FOR INCLUDE
//       FIRMA=S                         FIRST LEVEL OF LIBS
//FORSYSIN DD *
       "GENERATE PRIMES "
       "================"

       PARAMETER ARRAYSIZE= 100, PRIMESIZE= 3000;

       INTEGER SIEVE1;
       INTEGER PRIMES(PRIMESIZE);
       INTEGER COUNT;

       COUNT=SIEVE1(10000, PRIMES);
       OUTPUT COUNT; (I8,' PRIMES LESS THAN 10,000'//);
       OUTPUT (PRIMES(I),I=1,COUNT); (1X,10I10);
       STOP;
       END;
%INCLUDE ST#K.LIB.CNTL(SIEVE1)
```

10.2 Output description

The page header gives the following information (marked with arrows in the (narrowed) output listing below:

→ 1 the level of the used set of macros (default is 0) as definded by **MACLEVL**=x in the JCL procedure

→ 2 the options of the precompiler as defined by **PREOPT**=xxxxxx (default is 110100) in the JCL procedure.

→ 3 the name of the source module, defined by the JCL procedure parameter **NAME**

→ 4 date of compilation

→ 5 time of compilation

→ 6 page counter

→ 7 column ruler, corresponding to the input only if indentation is set to 0 by %I0.

→ 8 the (compiler)-**internal statement number** ISN shown by the precompiler may not be accurate due to pending macro-expansion. The deviation is seldomly more than +/- 1 to the correct figure.

The body of the listing contains normally:

→ 9 columns 73 to 77 of inputline is (when coming from SLM) the source line number.

→10 columns 78 to 80 of inputline show (when coming from SLM the updating information

→11 **nesting level** of **included members**, original source level is 0 which is not printed

→12 control-character % is printed instead of ISN for controls and in the first input line

→13 error-messages are preceded by * to show them better

→14 the + is printed in all cases of **messages** and **trace output**.

→15 The end of the page agian gives a columns ruler (see 7) and the page-number.

→16 Summary message at end of input. NOTE that a severity code of level 8 (min) is given when output punched, although there may be no errors of this level.

```
                          ↓1            ↓2            ↓3              ↓4    ↓5          ↓6
OBRZ FTN-PRECOMPILER (MCSTRUFO OCT830) ***** OPTIONS= 110111 ***** NAME= SIEVE     ***** 25.11.83 11:58 *****   0001
 ISN  ....*....1....*....2....*....3....*....4....*....5....*....6....*....7....*....8....*....9....*....0. IDENT UPD   INC
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% ←12     "GENERATE PRIMES"           ↑7                          00091 100
0003        "===========================                                        00092 000
                                                                   00093 000
↓8          PARAMETER ARRAYSIZE= 100, PRIMESIZE= 3000,                            9→00094 000 ←10
                                                                   00095 000
            INTEGER SIEVE1;                                          00096 000
            INTEGER PRIMES(PRIMESIZE);                                 00097 000
0004        INTEGER COUNT;                                            00098 000
0005                                                               00099 000
0006        COUNT=SIEVE1(10000, PRIMES);                                 00100 000
            OUTPUT COUNT, (I8,' PRIMES LESS THAN 10,000'//);               00110 000
0008        OUTPUT (PRIMES(I),I=1,COUNT), (1X,10I10);                    00120 000
0010        STOP,                                                   00130 000
0011        END ↓15                                                 00140 000
      ....*....1....*....2....*....3....*....4....*....5....*....6....*....7....*....8....*....9....*....0.          0001
```

```
OBRZ FTN-PRECOMPILER (MCSTRUFO OCT830) ***** OPTIONS= 110111 ***** NAME= SIEVE       ***** 25.11.83 11:58 *****   0002
 ISN  ...*...1...*...2...*...3...*...4...*...5...*...6...*...7...*...8...*...9...*...0. IDENT UPD    INC
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  %INCLUDE ST#KLIBCNTL(SIEVE1)                        00150 000
0003       INTEGER FUNCTION SIEVE1(MAX, PRIMES);   ! GENERATE PRIMES             00010 001 11→  01
           "ST#KLIBCNTL(SIEVE1) IS INCLUDES FROM ST#KLIBCNTL(SIEVE)"                00011 001    01
0004                                                                        00020 001    01
           "  USES ALGORITHM 310 (A1), PRIME GENERATOR 1,   "                      00030 001    01
           "  B A CHARTRES, CACM 10 (SEPT 1967), 569       "                     00040 001    01
                                                                        00050 001    01
           INTEGER MAX; INTEGER PRIMES(PRIMESIZE);                                00060 001    01
0005       INTEGER Q(ARRAYSIZE),DQ(ARRAYSIZE);                                  00070 001    01
0006       INTEGER I,SVSZ,PRMCNT,N;                                           00080 001    01
0007       LOGICAL PRIME;                                                    00090 001    01
0008                                                                        00100 001    01
0009         " INITIALIZE "                                                  00110 001    01
                                                                        00120 001    01
           PRIMES(1),SVSZ,PRMCNT=2; PRIMES(2)=3; Q(2)=9; DQ(2)=6;                00130 001    01
0012                                                                        00140 001    01
0013       :PRIME TEST:                    ! LOOK FOR PRIMES                   00150 001    01
           FOR N=5 TO MAX BY 2 <<                                             00160 001    01
0014         PRIME=TRUE;                                                     00170 001    01
0018         FOR I=2 TO SVSZ << " RUN THE NUMBER N THROUGH THE SIEVE "              00180 001    01
0019           IF N.EQ.Q(I) << " CAUGHT BY THE SIEVE? "                           00190 001    01
0023             PRIME=FALSE;                                                00200 001    01
0024             Q(I)=N+DQ(I);                                               00210 001    01
0025             IF I.EQ.SVSZ <<                                            00220 001    01
0026               SVSZ=SVSZ+1; Q(SVSZ)=PRIMES(SVSZ)**2;                         00230 001    01
0028               DQ(SVSZ)=2*PRIMES(SVSZ);                                     00240 001    01
0029               NEXT :PRIME TEST:;                                          00250 001    01
0031               >>                                                       00260 001    01
             >>                                                             00270 001    01
0032         >>                                                             00280 001    01
0033         IF (PRIME) << PRMCNT=PRMCNT+1; PRIMES(PRMCNT)=N; >>                  00290 001    01
0038         >>                                                             00300 001    01
0039       SIEVE1=PRMCNT;                                                    00310 001    01
0041       RETURN;                                                          00320 001    01
0042       END;                                                             00330 001    01
**** +←13,14
**** + NUMBER OF ERRORS ENCOUNTERED IS   0 HIGHEST SEVERITY IS  8   ←16
      ...*...1...*...2...*...3...*...4...*...5...*...6...*...7...*...8...*...9...*...0.        0002
```

10.3 Generated code

The sample program is transformed to the following code. Notice the information given in the first two lines - its a copy of the first page header.

The additional program segment BLOCK DATA for $COMM1 is added at linker time; so it is not shown here.

```
C OBRZ FTN-PRECOMPILER (MCSTRUFO OCT.83/0) ****** OPTIONS= 110111 ******OBRZ0
C ****** NAME=  S I E V E       ****** 25.11.83 11:58  ******    ******OBRZ0
      INTEGER SIEVE1                                                    OBRZ0
      INTEGER PRIMES(3000)                                              OBRZ0
      INTEGER COUNT                                                     OBRZ0
      COUNT=SIEVE1(10000, PRIMES)                                       OBRZ0
      WRITE(6,90010)COUNT                                               OBRZ0
90010 FORMAT (I8,' PRIMES LESS THAN 10,000'//)                          OBRZ0
      WRITE(6,90020)(PRIMES(I),I=1,COUNT)                               OBRZ0
90020 FORMAT (1X,10I10)                                                 OBRZ0
      STOP                                                              OBRZ0
90030 CONTINUE                                                          OBRZ0
      COMMON /$COMM1/ $DUMMY, I$DUMY, I$ESTR, $SPARE (7)                OBRZ0
      END                                                               OBRZ0
      INTEGER FUNCTION SIEVE1(MAX, PRIMES)                              OBRZ0
      INTEGER MAX                                                       OBRZ0
      INTEGER PRIMES(3000)                                              OBRZ0
      INTEGER Q(100),DQ(100)                                            OBRZ0
      INTEGER I,SVSZ,PRMCNT,N                                           OBRZ0
      LOGICAL PRIME                                                     OBRZ0
      PRIMES(1),SVSZ,PRMCNT/=2                                          OBRZ0
      PRIMES(2)=3                                                       OBRZ0
      Q(2)=9                                                            OBRZ0
      DQ(2)=6                                                           OBRZ0
90040 CONTINUE                                                          OBRZ0
90050 N=5                                                               OBRZ0
      GOTO 90053                                                        OBRZ0
90051 N=N+(2)                                                           OBRZ0
90053 IF((2)*((N)-(MAX)).GT.0)GOTO 90052                                OBRZ0
      PRIME=.TRUE.                                                      OBRZ0
90060 I=2                                                               OBRZ0
      GOTO 90063                                                        OBRZ0
90061 I=I+1                                                             OBRZ0
90063 IF((I).GT.(SVSZ))GOTO 90062                                       OBRZ0
      IF(N .NE. Q(I))GOTO 90081                                         OBRZ0
      PRIME=.FALSE.                                                     OBRZ0
      Q(I)=N+DQ(I)                                                      OBRZ0
      IF(I .NE. SVSZ)GOTO 90101                                         OBRZ0
      SVSZ=SVSZ+1                                                       OBRZ0
      Q(SVSZ)=PRIMES(SVSZ)**2                                           OBRZ0
      DQ(SVSZ)=2*PRIMES(SVSZ)                                           OBRZ0
      GOTO 90041                                                        OBRZ0
90101 CONTINUE                                                          OBRZ0
90081 CONTINUE                                                          OBRZ0
      GOTO 90061                                                        OBRZ0
90062 CONTINUE                                                          OBRZ0
      IF(.NOT.(PRIME))GOTO 90121                                        OBRZ0
      PRMCNT=PRMCNT+1                                                   OBRZ0
      PRIMES(PRMCNT)=N                                                  OBRZ0
90121 CONTINUE                                                          OBRZ0
      GOTO 90051                                                        OBRZ0
90052 CONTINUE                                                          OBRZ0
      SIEVE1=PRMCNT                                                     OBRZ0
      RETURN                                                            OBRZ0
90130 CONTINUE                                                          OBRZ0
```

```
COMMON /$COMM1/ $DUMMY, I$DUMY, I$ESTR, $SPARE (7)                OBRZ0  >
END                                                               OBRZ0  >
```

A

B

C

D

E

F

G

H

I