Lawrence Berkeley National Laboratory
Recent Work

Title
THE 'MHF' FORTRAN COMPILER

Permalink
https://escholarship.org/uc/item/7wd3f028

Author
Schofield, Clive F.

Publication Date
1973-10-01
THE 'MNF' FORTRAN COMPILER

Clive F. Schofield

October 1973

Prepared for the U.S. Atomic Energy Commission
under Contract W-7405-ENG-48

For Reference
Not to be taken from this room
DISCLAIMER

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.
An Introduction to the MNF FORTRAN Compiler

In April, 1973, a new FORTRAN compiler was installed at BKY. This step was taken since the new compiler offers very significant advantages over the RUN76 and, to a lesser extent, the FTN compilers. The new compiler was written at the University of Minnesota, and is called MNF (for Minnesota FORTRAN). MNF is used extensively at several installations — notably at the University of London, at Imperial College, and, of course, at Minnesota. MNF is fully supported by its authors and it is now running a substantial percentage of BKY Fortran jobs.

The advantages of MNF over RUN76 and FTN are as follows:

(A) The information content of diagnostics (error messages), source listing, and cross reference map of MNF is considerably better than that of RUN76 or FTN.

(B) Compilation speed of MNF is about twice that of RUN76, and 5 to 10 times that of FTN.

(C) Execution speed of MNF is nearly always better than that of RUN76, and is sometimes better even than FTN (OPT=2).

(D) MNF has execution-time debugging statements which are superior to those of FTN. RUN76 has no debugging facilities.

(E) MNF accepts many extensions to the FORTRAN language — such as expressions in IF statements, as well as format-free Input/Output. The MNF language is, as far as possible, a superset of both RUN76 and FTN.

The three major incompatibilities between the 3 compilers (RUN76, FTN, and MNF) are as follows:

(1) MNF and FTN require that specification statements (DIMENSION, type, COMMON, EXTERNAL, EQUIVALENCE, and DATA) precede the first executable statement. It is good programming practice to put such statements first, even though RUN76 does not insist upon it.

(2) The IF(EOF,...) test has various forms in the 3 compilers. I suggest that only the following forms be used, even though IF(EOF,...) is accepted by MNF:

IF(EOF(n))

IF(EOF(n).NE.0)GO TO yes

(3) FORTRAN II statements and intrinsic functions are accepted by neither MNF nor FTN. E.g., the following would have to be changed:

WRITE OUTPUT TAPE (use WRITE)
READ INPUT TAPE (use READ)
FORTRAN II SUBROUTINE, et al (use SUBROUTINE)
ABS,F,SIN,COS,F,XINTF,etc. (use ABS,F,SIN,COS,F,XINTF,etc.)

(The READ, PRINT, and PUNCH statements, and LOC function are accepted).

There are some further restrictions in the present version of MNF:

(a) OVERLAYs are not accepted by MNF, and fully relocatable binary code is not produced. The object code generated by MNF must be loaded into address RA+1000, so that although precompiled object files, COMPASS decks, and the like may be run with MNF programs, the process of doing this is less free than it is with RUN76 or FTN. MNF compiles a RUN76-compatible calling sequence, and a control card option exists to generate the FTN calling sequence.

(b) The LARGE statement of RUN76 is not accepted. However, the FTN 'ECS' (or TYPE ECS) statement is accepted with the restriction that a maximum of 131071 words (37777) of LCM may be used.

(c) Intermixed COMPASS subprograms are not accepted by MNF. COMPASS subprograms must be assembled by means of an explicit COMPASS control card.

MNF has a different formatted (coded) I/O library from RUN76 and FTN. The MNF library is both faster and more accurate than the other libraries on the BKY system, and is about the same size as the RUN76 library. Apart from being more efficient, the MNF library can handle format-free I/O and has clearer error messages than the existing libraries (the error numbers are the same as in RUN76). Exotic facilities, such as non-standard error recovery, are also available in the MNF library. Except for coded I/O, the BKYIIIO library may also be used.

Until the new BKY Fortran manual is completed, this document, in conjunction with the BKY HANDBOOK subset FORTRAN, provides a full description of how to use MNF, and of the differences (mainly extensions) between it and RUN76 and FTN. An up-to-date MNF bug list is in BKYNEWS.

The control cards needed for running MNF are:

1) Simple job - 7600

JOB CARD
MNF,
100.
7-8-9 card
FORTRAN Source Deck
7-8-9 card
Data (if any)
End-of-job card
The minimum field length on the job card is 50000 for a small job. Putting large arrays into COMMON will reduce the compilation core required by MNF. The same compiler and library is used on the 7600 and the 6000 machines - and the control cards needed are the same for each machine.

(2) **MNF with UPDATE - 7600**


```plaintext
JOBCARD
REQUEST(OLDPL,99999) [tape with FORTRAN programs ]
UPDATE(P=OLDPL,P) [MNF input on COMPILE ]
MNF(I=COMPILE)
LGO.
7-8-9 card
UPDATE cards
7-8-9 card
Data (if any)
End-of-job card
```

Note that there must be an end-of-record after the MNF decks and before the COMPASS decks, since MNF does not automatically call COMPASS.

(3) **MNF with COMPASS - 7600**


```plaintext
JOBCARD
MNF.
COMPASS.
LGO.
7-8-9 card
MNF program
7-8-9 card
COMPASS subprograms
7-8-9 card
Data (if any)
End-of-job card
```

As mentioned above, MNF has especially good debugging features and these are easily available to the user simply by specifying a "T" option on the control card -- as in MNF(T) or MNF.T. This option turns on the TRACE SUBSCRIPTS statement, which causes all array subscripts to be checked at execution time to see that they do not go outside the dimensions of the array. This is one of the most common programmer errors and is difficult to find when using RUN76. Several other traces are also activated by T. Some of them are:

(a) **TRACE FORMATIO** -- This checks that I/O list items are transmitted using an appropriate FORMAT conversion, e.g., that integers are not printed using F conversion. RUN76 and FTN simply print 0.0 this is done and they do not indicate the program error.

(b) **TRACE STATEMENT NUMBERS** -- This is an unusual and very valuable trace, which prints out (at the end of execution of the job) a list of statement numbers for each FORTRAN routine together with a count showing how many times the numbered statements were executed. The tables are self-explanatory and are very useful.

(c) **TRACE SUBPROGRAM CALLS** -- This is similar to (b), but prints a table of each routine name in the deck (including library functions) with a count of the number of times it was called.

Other traces are also turned on by "T", which will slow down execution of the program by about 50% because of the extra code needed to make all the checks. TRACE SUBSCRIPTS is the most expensive trace, since each array reference must be checked.

It will be seen from the above that these traces may be turned on and off by MNF statements as well as by "T". The statements are punched beginning in column 7 -- and can be turned off by a later statement saying (say) NOTRACE SUBSCRIPTS. A further trace example is TRACE A(1.0,10.7) which checks that values stored into A are within the range 1.0 to 10.7; if A assumes a value outside this range, then a message is printed during execution. Although use of the tracing statements will slow down execution, the degradation is much less severe than with FTN DEBUG mode, which will often slow down the execution of FTN jobs by factors of 10 or so. Note that MNF tracing is not the same as FTN DEBUG, although the facilities are similar.
The MNF control card has a free format which is very similar to those of RUN76 and FTN. The main MNF options are:

- **B=<file>** Compiled object code written to <file>. Note that, at present, this file must be loaded FIRST. Default is B=LOG.

- **D** Compile statements with CS in columns 1 and 2, and execute program event if it has fatal compilation errors. Execution stops when the first fatal error is reached. Default is not D, i.e., CS cards are treated as comments and execution is not started if fatal compilation errors occur.

- **E=<number>** (all through 5) <number> specifies the highest level of diagnostic error messages to be suppressed. E=0 means print all messages; E=5 means no messages at all.
  
  Level = 1 - COMMENT - Comments on inefficient programming practice.
  
  = 2 - NOTE - Non-standard (ANSI) FORTRAN used. This is useful if the program is to be published, or run on other machines.
  
  = 3 - CAUTION - E.g., "Go to next statement ignored". May indicate rather serious errors, so CAUTIONS should not be turned off without good reason.
  
  = 4 - WARNING - Usually serious errors.
  
  = 5 - FAI1 - Will abort job unless "D" is used.
  
  = 6 - DEADLY - Not enough core for compilation. Compilation continues, but compiled code is lost. Job will not execute. (E=6 is illegal).

Default is E=2. Notes and comments are not printed.

- **I=<file>** Input source program on <file>. Default I=INPUT.

- **L=<file>** Source listing on <file>. Default L=OUTPUT. L=0 (zero) for no list (implies R=0).

- **O** (letter O) Object listing on the same file as L listing. Default is not O.

- **P=<decimal number>** Maximum number of lines to be printed on the OUTPUT file during execution. Default is P=5000.

- **R=<number>** Cross reference map control. A map is printed after each routine.
  
  R = 0 - No map is printed.
  
  R = 3 - (Nearly) full map is printed.
  
  R = 7 - Full map is printed.

Default is R=3. This map is better than FTN(R=3) and infinitely better than the RUN76 maps.

**T** Execution time tracing control. Turns on the statements:

TRACE SUBSCRIPTS
  " STATEMENT NUMBERS
  " SUBPROGRAM CALLS
  " TRANSFERS
  " DOLOOPING
  " FORMATIO

Other traces are available by explicit statements. Default is not T.

Other options are also available on the MNF control card. The "default" control card is:

MNF(E=2,I=INPUT,L=OUTPUT,P=5000,B=LOG,R=3)

In the long term, MNF may replace RUN76. FTN will be retained and maintained since FTN(OPT=2) normally generates faster object code than MNF. In addition, FTN is supported by CDC, whereas RUN76 is not. RUN76 will be retained for the foreseeable future - but users are urged to give MNF a try, and also to use the "T" option which will help to debug their programs.

Clive Schofield, 5/118
# LIST OF APPENDICES

<table>
<thead>
<tr>
<th>Appendix</th>
<th>Pages</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>20</td>
<td>List of differences between RUN76, MNF, and FTN.</td>
</tr>
<tr>
<td>B</td>
<td>38</td>
<td>Optimization of Fortran Programs.</td>
</tr>
<tr>
<td>C</td>
<td>15</td>
<td>Extra facilities allowed by MNF.</td>
</tr>
<tr>
<td>D</td>
<td>12</td>
<td>The MNF TRACE statements.</td>
</tr>
<tr>
<td>E</td>
<td>21</td>
<td>The MNF Format-free I/O statements, and other extensions to I/O.</td>
</tr>
<tr>
<td>F</td>
<td>6</td>
<td>MNF Listings.</td>
</tr>
<tr>
<td>G</td>
<td>2</td>
<td>MNF Object Code.</td>
</tr>
<tr>
<td>H</td>
<td>3</td>
<td>FORTRAN statement order</td>
</tr>
<tr>
<td>I</td>
<td>6</td>
<td>The RUN76 and MNF System routine.</td>
</tr>
<tr>
<td>J</td>
<td>6</td>
<td>MNF Fortran Statement List.</td>
</tr>
<tr>
<td>K</td>
<td>14</td>
<td>Fortran Library Functions and Subroutines.</td>
</tr>
</tbody>
</table>

Acknowledgement is made to the University of Minnesota for permission to reproduce much of the information in these Appendices.
Differences between RUN76, FTN, and MNF

This section attempts to describe the syntactic and semantical differences between the three BKY Fortran compilers. The word 'syntactic' refers to the form of the Fortran statements themselves and thus refers to the compilation phase; whereas semantic refers to what the statements mean - or what they do when the program is executed. All Fortran compilers differ from each other - to some extent this is obvious since each 'compiler' is in reality a distinct computer program. With the three compilers here discussed the programs which comprise them are radically different. Whilst all three compilers are written in 'machine language' (COMPASS), their lengths are, approximately:

RUN76 30000 statements (these are not proportional to the
MNF 65000 " field length required for compilation).
FTN 120000 "

Most users will know that the reliability (i.e. freedom from bugs) of a program is in some inverse ratio to its length - and during the early lives of MNF and FTN this was surely the case. In the early days, MNF was never as unreliable as FTN, and both were less reliable than RUN. However, these compilers have been in use for several years now, and each has performed some millions of compilations. This has enabled the compiler writers to correct nearly all of the errors, and RUN76, MNF, and FTN are now considered to be relatively free from errors. If your program goes wrong (in execution), then it is not likely to be due to a compiler bug - although there is probably no compiler that is entirely free from error.
The fact of compilers being different programs has another, more significant, impact upon the Fortran user, and it is this to which this section is really addressed. In short, the differences between the compilers means that a program which works (or appears to work) correctly on MNF may fail to compile, or (more rarely) give different answers using FTN or RUN76 (permute the names as you will). This is becoming a more serious problem at BKY as more users are transferring their RUN76 programs to FTN and MNF, and are finding that some of the more non-standard features of RUN76 do not work using the other 2 compilers. The point of this section is to help users avoid those programming practices which will cause problems when running their programs on different compilers.

Note that only the 3 BKY compilers are discussed - e.g. small mention is made of masking expressions which work at BKY, but will fail even to compile using IBM compilers. If it is important that your program can run outside BKY then you should program in standard FORTRAN, which is described in American National Standards Institute Report X3.9, 1966 - New York. This report was not designed for the ordinary Fortran user however, and the MNF(E=0) option is very useful for pointing out non-standard Fortran statements. Note messages are printed for each non-standard statement when E=0 is used. In addition, the BKY Fortran manual, which is based on the CDC FTN Version 4 manual, will help in this regard. It points out the CDC features which are non standard; and which may, therefore, not work on other machines. Nearly all manufacturers of large and medium sized computers have compilers
which accept programs written in Standard Fortran.

The table below indicates those features which will work on at least one of the BKY compilers, but may fail on the other two.

In the FTN column, the phrase 'Not yet' means that the facility is available under FTN Version 4 (7600 Version 2), but not under FTN Version 3 (7600 Version 1). *** on the left of the page indicates an important difference between the compilers.
SECTION I: Specification Statements

<table>
<thead>
<tr>
<th>Feature</th>
<th>MNF</th>
<th>FTN</th>
<th>RUN76</th>
</tr>
</thead>
<tbody>
<tr>
<td>*** Precedence Rules (order required for specification statements, etc.)</td>
<td>Following order required: (1) IMPLICIT (2) Type, COMMON, EQUIVALENCE, EXTERNAL and DATA (3) Statement functions (4) Executable statements (FORMAT and NAMELIST anywhere)</td>
<td>Following order required: (1) Type, COMMON, EXTERNAL (2) EQUIVALENCE (3) DATA (4) Statement functions (5) Executable statements (FORMAT and NAMELIST anywhere)</td>
<td>Arrays and variables must be declared before they are used. Statement functions should come before executable statements.</td>
</tr>
<tr>
<td>*** IMPLICIT statement</td>
<td>Allowed</td>
<td>Not yet</td>
<td>No</td>
</tr>
<tr>
<td>IBM/360 style type statements. E.g. REAL*8</td>
<td>No. Note that REAL<em>8 means DOUBLE on /360 thus giving about 15 digits of accuracy. CDC single precision is nearly as accurate (14.6 digits) as IBM REAL</em>8.</td>
<td>As MNF</td>
<td>As MNF</td>
</tr>
<tr>
<td>*** ECS type statement</td>
<td>Allowed (maximum of 131071 words—otherwise same as FTN)</td>
<td>Allowed (to reference LCM)</td>
<td>No</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------------------------------------------------------</td>
<td>----------------------------</td>
<td>----</td>
</tr>
<tr>
<td>*** LARGE type statement</td>
<td>No.</td>
<td>No.</td>
<td>Allowed (to reference LCM)</td>
</tr>
<tr>
<td>FORTRAN II column one designators B, D, F, I</td>
<td>No.</td>
<td>No.</td>
<td>Allowed (in part -- e.g. F in column one means EXTERNAL)</td>
</tr>
<tr>
<td>EQUIVALENCE storage problem? (See note at the end of this chapter)</td>
<td>Yes — unless F option used on MNF card.</td>
<td>Yes, in general (EQUIVALENCE itself is correct).</td>
<td>Yes.</td>
</tr>
<tr>
<td>Number of subscripts allowed in EQUIVALENCE statements.</td>
<td>3</td>
<td>As MNF</td>
<td>One only</td>
</tr>
</tbody>
</table>
### SECTION II: Constants, and Expressions

<table>
<thead>
<tr>
<th>Feature</th>
<th>MNF</th>
<th>FTN</th>
<th>RUN76</th>
</tr>
</thead>
<tbody>
<tr>
<td>Onnn form of octal constants</td>
<td>No</td>
<td>No</td>
<td>Yes (nnn B form is also allowed and should be used).</td>
</tr>
<tr>
<td>Internal forms of constants .TRUE., and .FALSE. (this has implications for 1 and 2 branch Arithmetic IF statements)</td>
<td>True = negative</td>
<td>True = negative</td>
<td>True = non zero</td>
</tr>
<tr>
<td></td>
<td>False = positive</td>
<td>False = +0</td>
<td>False = +0</td>
</tr>
<tr>
<td>Type assumed for octal (B) and Hollerith (H) constants</td>
<td>Same as other operand in +,-,*,/**, or = operations. Otherwise type Integer is assumed</td>
<td>Same as other operand in +,-,*,/**, or = operations. Otherwise type Integer is assumed</td>
<td>See Handbook chapter FORTRAN</td>
</tr>
<tr>
<td>Mixture of type logical and arithmetic types in expressions</td>
<td>No</td>
<td>No</td>
<td>Allowed (results are not usually meaningful)</td>
</tr>
<tr>
<td>Integer and Real mixtures. E.g. evaluation of X=I/J+R</td>
<td>X=FLOAT(I/J)+R</td>
<td>As MNF</td>
<td>X=FLOAT(I)/FLOAT(J)+R</td>
</tr>
<tr>
<td>Evaluation of X=I/J/R</td>
<td>X=FLOAT(I/J)/R</td>
<td>X=FLOAT(I)/FLOAT(J)/R</td>
<td>As FTN</td>
</tr>
<tr>
<td>Evaluation of $A^{**B^{**C}}$</td>
<td>Not allowed</td>
<td>$(A^{**B})^{**C}$</td>
<td>As FTN</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-------------</td>
<td>------------------</td>
<td>--------</td>
</tr>
<tr>
<td>Type combinations allowed in $X^{**Y}$</td>
<td>X may be complex only if $Y$ is Integer otherwise no restriction</td>
<td>As MNF</td>
<td>Following restrictions X may be complex only if $Y$ is Integer. $Y$ may not be complex. X may be integer only if $Y$ is integer.</td>
</tr>
<tr>
<td>***Form of subscripts</td>
<td>Any arithmetic expression</td>
<td>As MNF</td>
<td>Any arithmetic expression not containing ** nor parentheses. So subscripted subscripts are not allowed</td>
</tr>
<tr>
<td>***Reserved words (names that cannot be used)</td>
<td>FORMAT used as an array name is the only reserved word</td>
<td>As MNF</td>
<td>Many - e.g. CALL, END, FORMAT, etc. (use of some of these words causes RUN76 to loop).</td>
</tr>
<tr>
<td>Maximum number of subscripts</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Meaning of array name written without subscripts</td>
<td>Means entire array in I/O and DATA statements, and in COMMON or type statements. Means first element of array in argument lists</td>
<td>As MNF</td>
<td>As MNF</td>
</tr>
</tbody>
</table>
| Relational expressions: effect of complex .op. complex | Imaginary parts ignored | Both real and imaginary parts significant if .op. is .EQ. or .NE. | As MNF  
|------------------------------------------------------|-------------------------|-------------------------------------------------------------|--------  
| Use of octal constants as subscripts, DO parameters, or as dimensions. | Allowed | As MNF. | No.  
| *** Primed text constants allowed as in #constant# | Yes - they terminate with a zero byte (12 bits), if used in CALL or I/O List. | Not yet. | No  
| *Constants*, and H,R, and L constants allowed | Yes | As MNF | As MNF  

### SECTION III: Programs and Subprograms

<table>
<thead>
<tr>
<th>Feature</th>
<th>MNF</th>
<th>FTN</th>
<th>RUN76</th>
</tr>
</thead>
<tbody>
<tr>
<td>File environment assumed if PROGRAM statement omitted</td>
<td>PROGRAM ST. (INPUT, OUTPUT, TAPE5 = INPUT, TAPE6 = OUTPUT)</td>
<td>PROGRAM START. (INPUT,OUTPUT)</td>
<td>PROGRAM (INPUT, OUTPUT) (blank name)</td>
</tr>
<tr>
<td>Buffer lengths in PROGRAM statement (including 20 words for F.E.T.)</td>
<td>Decimal or octal B. Default lengths are 2020B on 6600 422B on 7600</td>
<td>As MNF</td>
<td>Octal (B or not) Default Buffer lengths are 2022B on 6600 424B on 7600</td>
</tr>
<tr>
<td>Special PROGRAM statements. E.g. FORTRAN II PROGRAM</td>
<td>No</td>
<td>No</td>
<td>Allowed (but not recommended)</td>
</tr>
<tr>
<td>*** Variable RETURN statement</td>
<td>No</td>
<td>Allowed, with special CALL statement.</td>
<td>No</td>
</tr>
<tr>
<td>Special subprogram header statements. E.g. FORTRAN VI FUNCTION</td>
<td>No</td>
<td>No</td>
<td>Allowed (but not recommended)</td>
</tr>
<tr>
<td>*** FORTRAN II style functions allowed</td>
<td>No</td>
<td>No</td>
<td>Yes (should be avoided in new programs)</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>----</td>
<td>----</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>SHIFT intrinsic function</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>*** MASK intrinsic function</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Random number generators</td>
<td>RANF, and RANGET, RANSET available</td>
<td>RANGET, RANSET used</td>
<td>RANF used</td>
</tr>
<tr>
<td>Statement numbers allowed in argument lists. E.g. CALL X(10S)</td>
<td>No (use ASSIGNed label variables in CALL).</td>
<td>No. Use variable RETURN instead</td>
<td>Allowed (ASSIGNed label variables also allowed in CALL).</td>
</tr>
<tr>
<td>Duplicate actual argument; and identical actual argument with COMMON problem? (See note at end of this chapter)</td>
<td>Yes (unless F option used on MNF card)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>*** RETURN allowed in main program</td>
<td>Yes</td>
<td>Not yet</td>
<td>Yes</td>
</tr>
<tr>
<td>*** Execution of END statement</td>
<td>Acts as STOP (as RETURN if MNF(C) or MNF(Y))</td>
<td>Acts as RETURN</td>
<td>Acts as STOP</td>
</tr>
<tr>
<td>FORTRAN II</td>
<td>FORTRAN IV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABSF</td>
<td>ABS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XABSF</td>
<td>IABS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTF</td>
<td>AINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XINTF</td>
<td>INT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MODF</td>
<td>AMOD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMODF</td>
<td>MOD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAXOF</td>
<td>AMAXO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAX1F</td>
<td>AMAX1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMAXOP</td>
<td>MAXO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMAX1F</td>
<td>MAX1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIN1F</td>
<td>MINO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMIN1F</td>
<td>MIN1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLOATF</td>
<td>FLOAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XFIXF</td>
<td>IFIX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGNF</td>
<td>SIGN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XSIGNF</td>
<td>ISIGN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIMF</td>
<td>DIM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XDIMF</td>
<td>IDIM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SINF</td>
<td>SIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COSF</td>
<td>COS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TANF</td>
<td>TAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASINF</td>
<td>ASIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACSF</td>
<td>ACOS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATANF</td>
<td>ATAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TANHF</td>
<td>TANH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXPF</td>
<td>EXP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOGF</td>
<td>ALOG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQRTP</td>
<td>SQRT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XLOCF</td>
<td>LOCP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RUN76 allows Fortran II and Fortran IV library and intrinsic function names, but FTN and MNF allow only the Fortran IV names. Therefore, the Fortran IV names are recommended, and Fortran II names should be changed according to the following list:
SECTION IV: Control statements

<table>
<thead>
<tr>
<th>Feature</th>
<th>MNF</th>
<th>FTN</th>
<th>RUN76</th>
</tr>
</thead>
<tbody>
<tr>
<td>*** Form of Assigned GO TO n, (e₁, e₂...)</td>
<td>Parenthesized list of labels may be omitted [except on MNF(T)]</td>
<td>Parenthesized list of labels required</td>
<td>As MNF.</td>
</tr>
<tr>
<td>Action if computed GO TO variable is out of range: GOTO (e₁,e₂,...,eₙ),m</td>
<td>GO TO e₁ if n ≤ l</td>
<td>GO TO e₁ if n ≤ l</td>
<td>Fatal execution error if m &lt; l or m &gt; n</td>
</tr>
<tr>
<td>One Branch Arithmetic IF allowed, e.g. IF(I+2)GO TO 10</td>
<td>No</td>
<td>Yes (may not be compatible with RUN76)</td>
<td>Yes</td>
</tr>
<tr>
<td>*** Two branch arithmetic IF statement allowed, e.g. IF(I+2)10,20</td>
<td>No</td>
<td>Yes (may not be compatible with RUN76)</td>
<td>Yes</td>
</tr>
<tr>
<td>*** Expression allowed as computed GO TO control</td>
<td>Yes</td>
<td>Not yet</td>
<td>No</td>
</tr>
<tr>
<td>FOR statement (reverse DO loop)</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>*** Expressions allowed as DO parameters</td>
<td>Yes (integer expressions)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>***** IF (logical) GO TO n allowed as terminal statement of DO loop. *****</td>
<td><strong>Yes</strong></td>
<td><strong>No</strong></td>
<td><strong>Yes</strong></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Type of n in STOP n and PAUSE n</td>
<td>Decimal</td>
<td>Octal</td>
<td>Octal</td>
</tr>
<tr>
<td>STOP # string # and PAUSE # string #</td>
<td>No. Up to 5 octal digits may appear instead of # string #</td>
<td>Not yet.</td>
<td>As MNF</td>
</tr>
<tr>
<td>FORTRAN II fault condition statements: IF (DIVIDE CHECK) n₁, n₂</td>
<td>No. Use GO TO n₂</td>
<td>As MNF</td>
<td>Allowed, but meaningless on CDC 6/7000.</td>
</tr>
<tr>
<td>IF (ACCUMULATOR OVERFLOW) n₁, n₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IF (QUOTIENT OVERFLOW) n₁, n₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FORTRAN II sense switches/sense lights SENSE LIGHT i</td>
<td>No. Use: CALL SLITE (i)</td>
<td>As MNF</td>
<td>FORTRAN II statements allowed, but so are the MNF calls, which should be used.</td>
</tr>
<tr>
<td>IF (SENSE LIGHT i) n₁, n₂</td>
<td>{ CALL SLITET (i, j) }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IF (SENSE SWITCH i) n₁, n₂</td>
<td>{ CALL SWITCH(i, j) }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OVERFL and DVCK library routines</td>
<td>No - use TRACE ARITHMETIC (see Appendix D)</td>
<td>No</td>
<td>Not working correctly</td>
</tr>
</tbody>
</table>
SECTION V: Input/Output

<table>
<thead>
<tr>
<th>Feature</th>
<th>MNF</th>
<th>FTN</th>
<th>RUN76</th>
</tr>
</thead>
<tbody>
<tr>
<td>*** FORTRAN II I/O statements allowed: READ TAPE</td>
<td>No. Use: READ(u)</td>
<td>As MNF</td>
<td>Allowed. Modern forms preferred (as in MNF).</td>
</tr>
<tr>
<td>WRITE TAPE</td>
<td>WRITE(u)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ INPUT TAPE</td>
<td>READ(u,fmt)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE OUTPUT TAPE</td>
<td>WRITE(u,fmt)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*** END and ERR parameters in READ statement</td>
<td>Yes (but not quite compatible with IBM/360 forms).</td>
<td>Not yet</td>
<td>No</td>
</tr>
<tr>
<td>*** Format free (list directed) I/O statements</td>
<td>Yes</td>
<td>Not yet</td>
<td>No</td>
</tr>
<tr>
<td>*** Expressions in output lists E.g. PRINT 10, X+Y</td>
<td>Yes</td>
<td>Not yet</td>
<td>No</td>
</tr>
<tr>
<td>*** Processing of FORMAT statements by the library. (See Handbook chapter FORTRAN for details).</td>
<td>As Standard Fortran (IBM). The A option on the MNF card causes compatibility with RUN76.</td>
<td>As Standard Fortran.</td>
<td>As RUN manual unless A option used on RUN76 card in which case as FTN/MNF (nearly).</td>
</tr>
<tr>
<td>*** FORMAT statement analysis during compilation</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>*** T specification allowed in FORMAT</td>
<td>Yes</td>
<td>Yes</td>
<td>No (only in BKYIO)</td>
</tr>
<tr>
<td>*** # string # conversion allowed in FORMAT (compatible with IBM)</td>
<td>Yes</td>
<td>Yes</td>
<td>No (only in BKYIO)</td>
</tr>
<tr>
<td><em>string</em> conversion allowed in FORMAT) (not compatible with IBM)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Action if <em>(or #) met in input field. E.g. READ 10 10 FORMAT (</em>,<em>...</em>) with data <em>A</em></td>
<td>Blanks substituted for all <em>(or #)</em></td>
<td>As MNF</td>
<td>Store up to first * and set rest of field to blanks</td>
</tr>
<tr>
<td>*** Default P scale factor for E and D output</td>
<td>OP.(e.g: .1E+01) (1P if MNF(A) is specified E.g: 1.0E+00)</td>
<td>OP.(e.g: .1E+01)</td>
<td>1P.(e.g: 1.0E+00)</td>
</tr>
<tr>
<td>*** IF(EOF,n) a,b IF(EOF(n)) a,b,a IF(ENDFILE n) a,b etc...</td>
<td>IF (EOF(n)) a,b,c is recommended for n</td>
<td>As MNF</td>
<td>As MNF</td>
</tr>
<tr>
<td>Condition</td>
<td>Description</td>
<td>Allowed</td>
<td>No-use</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>---------</td>
<td>--------</td>
</tr>
<tr>
<td>IF(UNIT, i) n₁, n₂, n₃, n₄ (used with BUFFER I/O)</td>
<td>Allowed. So is UNIT function (as next column)</td>
<td>No-use IF(UNIT(i)) n₂, n₃, n₄</td>
<td>Allowed</td>
</tr>
<tr>
<td>IF(IOCHECK, i) n₁, n₂</td>
<td>No. Use IF(IOCHECK(i)) n₁, n₂</td>
<td>As MNF</td>
<td>Allowed</td>
</tr>
<tr>
<td>Coded (formatted) I/O library routines used.</td>
<td>MNF library only</td>
<td>FTN library or BKYIO</td>
<td>RUN library or BKYIO</td>
</tr>
<tr>
<td>*** Maximum length of file names (declared on PROGRAM card)</td>
<td>7 characters</td>
<td>6 characters (in FTN Version 1.0)</td>
<td>As MNF</td>
</tr>
<tr>
<td>Dn, Gn, En, Fn allowed in FORMAT</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>-O printed using Fl.O conversion</td>
<td>Prints blank</td>
<td>Prints *</td>
<td>As FTN</td>
</tr>
<tr>
<td>Real number printed using I conversion</td>
<td>Prints X</td>
<td>As MNF</td>
<td>Prints R</td>
</tr>
<tr>
<td>Text constant used for F. or MAT, as in: WRITE (6, '#(I3)#')</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
### SECTION VI: Miscellaneous Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>MNF</th>
<th>FTN</th>
<th>RUN76</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library routines used (except formatted I/O)</td>
<td>RUN library except for formatted I/O routines. FTN library if MNF(c) used. BKYIO routines may also be used.</td>
<td>FTN library, or BKYIO</td>
<td>RUN library, or BKYIO</td>
</tr>
<tr>
<td>*** LIST and NOLIST statements</td>
<td>Allowed. May appear anywhere. (Also CODE and NOCODE for switching object listing)</td>
<td>No.</td>
<td>Allowed before subprogram header card only</td>
</tr>
<tr>
<td>New page in source listing</td>
<td>PAGE statement</td>
<td>No</td>
<td>Period in Column 1</td>
</tr>
<tr>
<td>*** Implied DO loop in DATA statement</td>
<td>Allowed to depth 3</td>
<td>As MNF</td>
<td>Allowed to depth 1 only. Also only one subscript may be specified</td>
</tr>
<tr>
<td>Maximum number of continuation cards</td>
<td>Between 9 and 35. only less than 20 for very long expressions</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>$ as statement separator</td>
<td>Allowed anywhere</td>
<td>Not allowed after DATA, FORMAT, or END statements</td>
<td>As FTN</td>
</tr>
<tr>
<td>*** Intermixed COMPASS routines</td>
<td>No (use explicit COMPASS card).</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Rounded arithmetic used in execution (see Handbook chapter FORTRAN)</td>
<td>Yes (no option)</td>
<td>No (yes if ROUND option on FTN card)</td>
<td>No (yes if R option on RUN76 card)</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>***Calling sequence (see Handbook subset CPU)</td>
<td>RUN76 style (in Bl, etc.) unless MNF(C) used in which case FTN style (in Al)</td>
<td>FTN style (in Al) only</td>
<td>RUN76 style (in Bl, etc) only.</td>
</tr>
</tbody>
</table>
EQUIVALENCE/COMMON/Argument Interaction

This can lead to what has been described as the Forced Store Problem. This problem arises in all of the BKY FORTRAN compilers - because they all (to varying extents) optimise generated code.

The MNF compiler resolves this problem (in all of its aspects) if an F option is present on the MNF control card; it cannot be resolved when using the RUN76 or FTN compilers. However the F option should only be used when really necessary, since its presence may slow down execution of the program quite considerably. The problem is best illustrated by examples:

(a) EQUIVALENCE(A,B)

(1) B = 1.0+A
(2) C = 1.0+A

Statement (1) increases the value of A by 1.0 (since A = B). Therefore statement (2) should be the same as C = 1.0+B.

However, because of optimisation, the compilers may have put A into a machine register whilst compiling statement (1) and when they come to statement (2) they may remember that A is already in that register - and not bother to reload it. If this happens then C will finish up with the wrong value. It will be seen that this problem only arises when one value (i.e. one computer cell) is known, or referenced, by more than one symbolic name. This situation can arise other than by using EQUIVALENCE, as in:-
(b) CALL SAM(A,A)
    :
    END
SUBROUTINE SAM(A,B)
(1) B = 1.0 + A
(2) C = 1.0 + A

and in
(c) COMMON B
    CALL X(B)
    :
    END
SUBROUTINE X(A)
COMMON B
(1) B = 1.0 + A
(2) C = 1.0 + A

In both of these examples the comments under (a) apply.

The EQUIVALENCE problem described in (a) is not present in FTN, but (b) and (c) may (and probably will) give wrong answers under FTN. All three examples may go wrong on RUN76 and MNF - unless the MNF(F) option is used, in which case the correct answers for all three will be given under MNF.

There is no way of avoiding the problem when using RUN76 - except by sensible programming. E.g. in all three examples, statement(s) (2) could be written

(2) C = 1.0 + B (using B instead of A)

and this would solve the problem: i.e. the name that was most recently to the left of the = sign is used. Sometimes the problem may be solved by putting a statement number on statement (2) but this is not infallible.

Most programs will run correctly without the F option, but if it is used and the results (answers) differ from the normal mode, then the program is using a cell which is known by more than one name.
OPTIMIZATION OF FORTRAN PROGRAMS

The key to optimization in Fortran is to understand the problem to be solved and to find the best possible methods (or algorithms) to solve that problem. Coding the best method without attempting to be efficient will almost always be better than writing the most efficient code using a poor method. Best algorithms will not be discussed here; instead it is assumed that the programmer has found and translated the best method into Fortran and wishes to improve the efficiency of the program by optimizing the code itself. An assumption is also made that the programmer does not want to use assembly language since although optimum code can be written in assembly language, it cannot be run on other manufacturers' machines and it is hard to write and debug for large programs. Furthermore, it is assumed that the program is worth optimizing. A short program to be run a few times is probably not a candidate.

To begin improving efficiency, first find the most-used repeated code: innermost loops or subprograms, and do the best possible Fortran programming there. The rest of the program may usually be ignored because most of the computer time spent on programs (other than I/O operations) is in this repeated code.

Finding the innermost loops is often quite easy - just a glance at the program by drawing in the loop levels will pin-point them or else use TRACE STATEMENT NUMBERS. In some cases, the loops are obscured by the fact that a library routine may contain them. This is particularly evident when the user supplies a FUNCTION or SUBROUTINE for integration or differential equation solving routines. There may be millions of FUNCTION references or SUBROUTINE calls from such routines so the user must carefully optimize his supplied subprogram to minimize computer time.

There are two levels of Fortran optimization: general types which apply to all compilers, and individual types which are tailored to suit a particular compiler or computer (and which may be very poor for some other compiler).

NOTE: Whilst this Appendix was taken from the MNF Reference Manual, most of its contents apply to RUN76 and FTN programs as well.
or computer). MNF users can take advantage of both types so we begin with a list of general optimization hints.

1. In DO, FOR or IF loops, move expressions that do not depend on the current index variable (or variables that are not changed in the loop) outside of the loop because such expressions or variables can be considered as constants for the loop. MNF does not automatically do this although some compilers do so (e.g. Fortran Extended with OPT=2, IBM Fortran H). For most compilers, however, when the user takes the time to do this type of optimization himself it is perhaps the most worthwhile one of all those discussed in this Appendix. Thus,

\[
\begin{align*}
\text{DO 10 J} & = 1, N \\
\text{DO 10 I} & = 1, N \\
10 & \text{ V(I) = VJ(I) + 0.5 * H * VK(J)} \\
\text{is better written as} & \\
\text{DO 10 J} & = 1, N \\
\text{TEMP} & = 0.5 * H * VK(J) \\
\text{DO 10 I} & = 1, N \\
10 & \text{ V(I) = VJ(I) + TEMP}
\end{align*}
\]

is better written as

\[
\begin{align*}
\text{DO 10 J} & = 1, N \\
\text{DO 10 I} & = 1, N \\
10 & \text{ V(I) = VJ(I) + 0.5 * H * VK(J)} \\
\text{is better written as} & \\
\text{DO 10 J} & = 1, N \\
\text{TEMP} & = 0.5 * H * VK(J) \\
\text{DO 10 I} & = 1, N \\
10 & \text{ V(I) = VJ(I) + TEMP}
\end{align*}
\]

and

\[
\begin{align*}
5 & \text{ F = SIN(Y) * XN ** 2 + 2. * COS(XN) + 2. * SIN(Y)} \\
\text{FP} & = 2. * \text{SIN(Y) * XN - 2. * SIN(XN)} \\
\text{XNP1} & = \text{XN - F / FP} \\
\text{T} & = \text{XNP1 - XN} \\
\text{XN} & = \text{XNP1} \\
\text{IF (ABS(T / XN) .GT. 1.0E-6) GO TO 5}
\end{align*}
\]

is better written as

\[
\begin{align*}
5 & \text{ F = Z * XN ** 2 + 2. * COS(XN) + TWOZ} \\
\text{FP} & = \text{TWOZ * XN - 2. * SIN(XN)} \\
\text{XNP1} & = \text{XN - F / SIN(XN)} \\
\text{T} & = \text{XNP1 - XN} \\
\text{XN} & = \text{XNP1} \\
\text{IF (ABS(T / XN) .GT. 1.0E-6) GO TO 5}
\end{align*}
\]
In a similar case, a loop containing a decision that does not depend on the loop control variable should be broken up into two or more separate loops. For example,

```
LOGICAL SHTCH
DO 20 I = 1, N
  IF (SHTCH) GO TO 15
  T(I) = A(I) * B(I)
  GO TO 20
15  T(I) = A(I) * C(I)
20 CONTINUE
  T(N) = 0.5 * T(N)
```

should be written as

```
LOGICAL SWTCH
  IF (SWTCH) GO TO 15
  DO 10 I = 1, N
10  T(I) = A(I) * B(I)
  GO TO 25
15  DO 20 I = 1, N
20  T(I) = A(I) * C(I)
25  T(N) = 0.5 * T(N)
```

2. Remove unnecessary statement numbers which are not being used. Heed the comment message "STATEMENT NUMBER n NEVER USED" because extra code is needed whenever a statement number is encountered and many compiler optimizations may have to stop there. For example, when MNF encounters a statement number on a statement inside a loop, it must save away indexing registers and can no longer remember operands in active registers for later use. (The exception is when the statement number terminates the loop and there have been no transfer statements in the loop.)

3. Whenever possible, place variables and arrays needed by subprograms in COMMON statements rather than in the formal parameter list. This minimizes the number of storage addresses that need to be transmitted and substituted for the formal parameter references every time the subprogram is executed. For example,
SUBROUTINE SAM (X, Y, Z, N)
   Y = Y + X ** N - X / N
   Z = Y + FLOAT (3 * N)
RETURN
END

can be optimized as
SUBROUTINE SAM
   COMMON /PL/ X, Y, Z, N
   Y = Y + X ** N - X / N
   Z = Y + FLOAT (3 * N)
RETURN
END

However, arrays in COMMON have a fixed size that is determined at compilation time so arrays with differing sizes which should use variable dimensioning cannot use COMMON. Another disadvantage of COMMON is that it makes subprograms less general -- the specific order and size of the COMMON list is pre-determined -- but in a production program to be optimized it is probably worth the sacrifice.

A further problem can arise when the subprogram is called or referenced in many places with different actual parameters. At each use, the values in COMMON must be changed to correspond to the current actual parameters and this would be inefficient if large arrays were involved. For this reason, the code:

```
DIMENSION A(50, 50), B(50, 50), C(50, 50), D(50, 50), E(50, 50),
1   F(50, 50), G(50, 50)
CALL MTXADD (A, B, C, 50)
CALL MTXADD (C, D, E, 50)
CALL MTXADD (E, F, G, 50)
```

```
SUBROUTINE MTXADD (X, Y, Z, N)
DIMENSION X(N, N), Y(N, N), Z(N, N)
DO 1 J = 1, N
   DO 1 I = 1, N
   1   Z(I, J) = X(I, J) + Y(I, J)
RETURN
END
```
should not be changed so that the formal parameters X, Y, Z and N reside in COMMON.

4. Use local variables in subprograms when a simple variable formal parameter would be needed many times within the subprogram. This minimizes the number of words needed for prestoration every time the subprogram is executed. Thus, heed the comment message "ARGUMENT name SHOULD BE SET TO LOCAL VARIABLE FOR FASTER SUBPROGRAM SETUP." For example,

```fortran
SUBROUTINE BOB (I, J, KK)
   DIMENSION KK(10)
   JIM = I * J + KK(1)
   IF (J .GT. KK(1)) GO TO 3
   IF (JIM .GT. J) GO TO 2
   DO 1 K = 1, 10
      1 JIM = JIM + KK(K) * K / I
   2 JIM = JIM / I - J ** I
   3 JIM = JIM / KK(10) - I
   RETURN
END
```

uses I enough times to cause the above comment message to be given. The SUBROUTINE could be re-written using I as a local variable and changing the formal parameter name to II.

```fortran
SUBROUTINE BOB (II, J, KK) *
   DIMENSION KK(10)
   I = II *
   JIM = I * J + KK(1)
   IF (J .GT. KK(1)) GO TO 3
   IF (JIM .GT. J) GO TO 2
   DO 1 K = 1, 10
      1 JIM = JIM + KK(K) * K / I
   2 JIM = JIM / I - J ** I
   3 II = JIM / KK(10) - I *
   RETURN
END
```

The asterisks mark the changed statements.
Note that a local variable may need to be stored into the formal parameter just before returning to the calling or referencing program unit. This situation can be observed during optimization by looking at the cross-reference map (see Appendix F) which notes when a variable is stored. In the first version of the SUBROUTINE above, I is both used and stored within it so the variable II in the second version must be stored into before returning from the SUBROUTINE.

In general, formal parameter arrays should not be stored into local arrays since except for very small arrays, this storing will take longer than the corresponding restoration code.

5. Use the DATA statement to prestore constants rather than using executable statements for initialization. This is particularly important for subprograms which are executed many times since both execution time and storage are saved. However, executable statements must be used for initialization if certain variables or arrays must be initialized every time the subprogram is entered. For example,

```
SUBROUTINE JOE (X, Y, A, B, N)
  DIMENSION A(N), B(N), TEMP(100), KEY(6)
  DO 1 I = 1, 100
  1 TEMP(I) = 0.0
  PI = 3.14159265358979
  DO 2 I = 1, 4
  2 KEY(I) = 3
  KEY(5) = 7
  KEY(6) = 9
  .
  .
  .
```

Here, array TEMP must be initialized during every call of JOE but PI and array KEY need only be initialized once so an optimized version is
SUBROUTINE JOE (X, Y, A, B, N)
DIMENSION A(N), B(N), TEMP(100), KEY(6)
DATA PI / 3.14159265358979 / , KEY / 4 * 3, 7, 9 /
DO 1 I = 1, 100
1 TEMP(I) = 0.0

6. If it is possible to use a single-dimension array instead of a two- or three-dimension array without many extra calculations for subscripts, then do so. If extra calculations are needed the compiler may be better able to do them.

However, the compiler cannot optimize subscript calculations when a loop contains complicated decisions, SUBROUTINE calls, FUNCTION references, I/O statements, extended DO ranges or non-standard subscripts. In this case, the user can optimize himself by using single-dimension arrays and calculating the subscript explicitly. For example,

SUBROUTINE JOE (X, Y, N, M, SAM)
COMMON A(100, 50), B(100, 50), C(100, 50)
CALL SAM (A, B, C, N, M, 0, 0)
X = 0.0
Y = 0.0
DO 10 J = 1, M
DO 10 I = 1, N
CALL SAM (A, B, C, N, M, I, J)
X = X + A(I, J) + B(I, J) / C(I, J)
10 Y = Y - 0.5 * A(I, J) * B(I, J)
RETURN
END
can be better written as
SUBROUTINE JOE (X, Y, N, M, SAM)
COMMON A(5000), B(5000), C(5000)  *
CALL SAM (A, B, C, N, M, 0, 0)
X = 0.0
Y = 0.0
IDIMA = -100  *
DO 10 J = 1, M
   IDIMA = IDIMA + 100  *
DO 10 I = 1, N
   IJ = IDIMA + I  *
   CALL SAM (A, B, C, N, M, I, J)
   X = X + A(IJ) + B(IJ) / C(IJ)  *
10 Y = Y - 0.5 * A(IJ) * B(IJ)  *
RETURN
END

where * denotes new or changed statements.

In a related optimization, it is best not to combine into one multi-dimensional array what could be separate arrays with fewer dimensions. For example, A(20, 2) might be split up as B(20) and C(20).

7. An array with 2 or 3 dimensions which must be completely set to zero (or to any other constant) should be considered a single-dimension array for that operation. For example,

    DIMENSION A(100, 50, 3)
    DO 1 I = 1, 3
    DO 1 J = 1, 50
    DO 1 K = 1, 100
    1 A(K, J, I) = 0.0

is better written as

    DIMENSION A(100, 50, 3)
    DO 1 I = 1, 15000
    1 A(I) = 0.0

The reason is that the subscript calculation for A(K, J, I) is more complicated than for A(I). Note that the Fortran compiler used must allow fewer subscripts than were declared in the DIMENSION, COMMON or type statement (MNDF does allow this).
8. If a subroutine or function is only used by one subprogram which is the most-often used subprogram, incorporate it into the calling or referencing program unit. Note that although this optimizes the code generated, it reduces the modularity and may make the program harder to follow or to debug. This can be alleviated by using more comments to explain the program's operation. Example:

```fortran
CALL RAND (ZERO)
DO 1 K = M, 20
CALL RAND(R)
IF (R .GT. 0.81) N(K) = 1
1 CONTINUE
.
.
SUBROUTINE RAND (SEED)
DOUBLE PRECISION XN
DATA XN / 2147483647.DO /
IF (SEED .EQ. 0.0) SEED = 1.0
SEED = DMOD(DBLE(SEED) * 1220703125.DO, XN) / XN
RETURN
END
```

is better written as

```fortran
DOUBLE PRECISION XN
C XN IS 2 ** 31 - 1
DATA XN / 2147483647.DO /
R = 1.0
DO 1 K = M, 20
C 1220703125 IS 5 ** 13
R = DMOD(DBLE(R) * 1220703125.DO, XN) / XN
IF (R .GT. 0.81) N(K) = 1
1 CONTINUE
.
.
.
```
(The random number generator RAND will work on any machine with a word-size of at least 32 bits and unrounded DOUBLE PRECISION arithmetic; however faster generators should be written in assembly language.)

9. Use Horner's Rule to evaluate polynomials so as not to evaluate $X ** I$ for each term. Thus $P(X) = \sum_{i=0}^{n} a_i x^i$ should be evaluated as $P(X) = ((... (a_n x + a_{n-1}) x + ...)) x + a_0$. For example, with coefficients $A(I), I = 1, ..., N+1$ where $A(1) = a_0, ..., A(N+1) = a_n$,

```
POLY = A(1)
NP1 = N + 1
DO 1 I = 2, NP1
  1 POLY = POLY + A(I) * X ** I
```
is better written as

```
POLY = A(N + 1)
FOR 1 I = N, 1
  1 POLY = POLY * X + A(I)
```
or be reversing the order of the coefficients where we have $B(I) = A(N + 2 - I), I = 1, ..., N + 1$, we can write in standard Fortran,

```
POLY = B(1)
NP1 = N + 1
DO 1 I = 2, NP1
  1 POLY = POLY * X + B(I)
```

If the order of the polynomial ($n$) is not too large, one may want to evaluate it without loops. For example $A * X ** 4 + B * X ** 3 + C * X ** 2 + D * X + E$ should be written as $(((A * X + B) * X + C) * X + D) * X + E$.

10. If a subexpression having several constants is needed in an expression inside heavily repeated code, evaluate the result elsewhere (on paper or with another program) and use that result in the calculation. For example,

```
Y = 3.0 + ALOG10(2.0) - X
```
is better written as (with comments added for clarity)

```
C 3.30102 99956 6398 = 3.0 + ALOG10(2.0)
Y = 3.30102 99956 6398 - X
```

and

```
Z = 2.0 * 3.14159265358979 * ANGLE
```
is better written as

```
C 6.28318 53071 7958 = 2.0 * PI
A = 6.28318 53071 7958 * ANGLE
```
Note that some compilers (such as MNF, section I below) can evaluate constant expressions themselves during compilation. However, constants involving library functions or ** (exponentiation) to other than integer powers usually cannot be evaluated during compilation so the user should do so.

11. A ** (REAL integer constant) where A is REAL or DOUBLE PRECISION should be written as A ** (INTEGER constant). For example, A ** 4.0 should be A ** 4 (because A ** 4 can be done by a series of multiplications while A ** 4.0 must use logarithm and exponential routines).

In a similar case, use SQRT(X) instead of X ** 0.5 and X * SQRT(X) instead of X ** 1.5 because the SQRT function is usually faster than the logarithm and exponential functions which are used to evaluate REAL ** REAL.

Complicated tolerance checks involving SQRT can sometimes be better written using X ** 2. For example,

\[
\text{IF (SQRT(X) .GE. Y) GO TO 10}
\]

is better written as

\[
\text{IF (X .GE. Y ** 2) GO TO 10}
\]
(assuming Y ** 2 does not overflow) and

\[
\begin{align*}
D &= 0.0 \\
\text{DO 100 I = 1, N} \\
100 &\quad D = \text{AMAX1 (D, SQRT (X(I) ** 2 + Y(I) ** 2))}
\end{align*}
\]
can be better written as

\[
\begin{align*}
D &= 0.0 \\
\text{DO 100 I = 1, N} \\
100 &\quad D = \text{AMAX1 (D, X(I) ** 2 + Y(I) ** 2)} \\
\quad D &= \text{SQRT(D)}
\end{align*}
\]
since only one evaluation of SQRT is needed.

12. A constant of the form k * 10 ** n should be written in the form kEn. For example, 3.7 * 10 ** (-3) should be 3.7E-3.

13. (-1) ** N should be rewritten to take advantage of the fact that the result is either +1.0 or -1.0. For example,
```
DO 1 N = 1, 15
SIGNE = (-1.0) ** N
.
.
should be
SIGNE = 1.0
DO 1 N = 1, 15
SIGNE = -SIGNE
.
.
In another example that refers back to section 9, \( \sum_{i=1}^{N+1} (-1)^{i-1} A_i X^{i-1} \)
or (what is equivalent) \( \sum_{i=1}^{N} (-1)^{i-1} B_{N+2-i} X^{i-1} \), can best be coded using
a double Horner step in statement 200:

```
POLY = B(1)
IF (N - 1) 400, 300, 100
100 X2 = X * X
DO 200 I = 2, N, 2
200 POLY = POLY * X2 - B(I) * X + B(I + 1)
300 IF (N .NE. 2 * (N / 2)) POLY = B(N + 1) - POLY * X
400 ... 
```

Note that this much more efficient of time (but not of space or clarity)
than the following:

```
POLY = A(1)
IF (N .EQ. 0) GO TO 500
NP1 = N + 1
DO 400 I = 2, NP1
400 POLY = (-1.0) ** (I - 1) * A(I) * X ** (I - 1)
500 ... 
```

A maxim of optimization is that one can usually trade space for faster
execution time.

14. Many computer decisions can be avoided during searches to find a
value if the code is properly written. Thus, to find which element in
an array is the same as a given value, one can write
DO 10 I = 1, N
  IF (XA .EQ. A(I)) GO TO 20
10 CONTINUE
  I = N + 1
20...

If I is N + 1 at statement 20, XA is not in the array. This requires two decisions per pass through the loop: one for the IF and one for the DO answering the question "is I equal to N?" A better method requiring only one decision per loop is

A(N + 1) = XA
  I = 0
10 I = I + 1
  IF (XA .NE. A(I)) GO TO 10
20...

Unless the machine fails, the loop is guaranteed to find the value in array A which is equal to XA, since the last element A(N + 1) is set to XA initially. This is a case where the normal Fortran DO statement is not always best.

Another example is the use of a table lookup instead of a search. This often occurs when converting from one type of code to another, e.g. display code to BCD code. The loop

DO 10 I = 1, 64
  IF (IX .EQ. ICHAR(I)) GO TO 20
10 CONTINUE
  JX = JCHAR(I)
20...

can be re-written as a single statement by taking advantage of the value of IX itself to index into the table JCHAR as

JX = JCHAR(IX)

if the JCHAR table is built up using the sequence of IX.

In still another example, the number of decisions is almost cut in half at the expense of more code; two operations are done per loop instead of one.
For example,
\[
\begin{align*}
S &= 0.0 \\
DO 10 \ I = 1, N \\
10 \ S &= S + A(I)
\end{align*}
\]
is better written as
\[
\begin{align*}
S &= 0.0 \\
IF (N \ LE. 1) \ GO \ TO \ 20 \\
DO 10 \ I = 1, N, 2 \\
10 \ S &= S + A(I) + A(I + 1) \\
20 \ IF (N \ NE. 2 * (N / 2)) \\
1 \ S &= S + A(N)
\end{align*}
\]
However, see section A below about further cautions.

15. Minimize the number of multiplications by use of the distributive law: \( A \times C + B \times C = (A + B) \times C \) and \( A \times C - B \times C = (A - B) \times C \). MNF does not detect this case, and other compilers that do may miss it if there are different subexpressions in between the combinable ones.

16. Whenever possible, use library routines provided by the computer installation rather than write your own. In general, installation-provided routines are of good quality. However, they sometimes cannot be run on other machines so this must be taken into consideration. Only if a programmer is very knowledgeable in an area and if no library program does what he wants should he attempt to write his own routine. Testing and debugging one's own routine can be much more expensive than obtaining someone else's version that does the same as well or better. The only good thing about writing your own routine is that it is very educational.

17. In nested loops, attempt to make the innermost loop have operands that are in adjacent words whenever possible. For example,
\[
\begin{align*}
&\text{DIMENSION A(16,16), B(16,16)} \\
&DO 10 \ I = 1, 16 \\
&DO 10 \ J = 1, 16 \\
&10 \ A(I, J) = A(I, J) + B(I, J) \times T
\end{align*}
\]
should be re-written by reversing the order of the I and J loops as

```fortran
DIMENSION A(16,16), B(16,16)
DO 10 J = 1, 16
DO 10 I = 1, 16
10 A(I, J) = A(I, J) + B(I, J) * T
```
since the first form uses elements that are 16 words apart while the second form uses adjacent elements. This prevents possible memory bank conflicts in some machines with interleaved memory banks and simplifies subscript incrementing in many compilers. On certain machines such a change can make an enormous difference in efficiency (see section Q, below).

In a similar case, keep references to a particular variable or array element in as few Fortran statements as possible and attempt to keep them within a single block of statements. (A block consists of individual statements terminated by transfer statements, labeled statements, I/O statements, or statements having subprogram references.) Such usage aids optimization for many compilers and minimizes the need for loading from and storing to memory on machines with multiple high-speed registers since values in active registers can be remembered.

18. Be careful when using non-standard mixed-mode arithmetic because this may cause inefficient code to be generated or loss of optimization inside loops. For example

**MNF allows**

```fortran
REAL A(10,20), B(10,20), D(10,20), E(10,20), F(10,20)
INTEGER X(10,20)
DO 10 J = 1, 20
  DO 10 I = 1, 10
    A(I, J) = X(I, J) + B(I, J)
    D(I, J) = X(I, J) * E(I, J) + F(I, J)
10
```

**standard Fortran requires**

```fortran
REAL A(10,20), B(10,20), D(10,20), E(10,20), F(10,20)
INTEGER X(10,20)
DO 10 J = 1, 20
  DO 10 I = 1, 10
    A(I, J) = FLOAT(X(I, J)) + B(I, J)
    D(I, J) = FLOAT(X(I, J)) * E(I, J) + F(I, J)
10
```

The non-standard mixed-mode arithmetic that MNF allows hides that fact that there must be type-conversions every time through the loop. The innermost code can thus be better written as

```fortran
REAL XIJ
...
XIJ = X(I, J)
A(I, J) = XIJ + B(I, J)
10 D(I, J) = XIJ * E(I, J) + F(I, J)
```
As another example,

\[ X = I \times Z - (I - 1) / T + 6 \times I \]

can be better written as

\[ V = I \]

\[ X = V \times (Z + 6.0) - (V - 1.0) / T \]

However, for mixed-mode arithmetic may allow more efficient code to be generated if it allows a DO or FOR loop to be expressed in one statement rather than as a multi-statement loop (see section A below).

19. In circumstances where it is possible, replace a multiplication by an addition, or an exponentiation to an integer power by a multiplication inside a loop. (The latter can appear when applying Horner's rule, section 9, above.) At the same time, replace tests to end a loop by an equivalent test based on another variable in the loop. As an example, the function \( f(x,n,m) \) represented by the double sum

\[
\sum_{r=1}^{n} \sum_{s=1}^{m} \frac{1}{(4s-x)^{2r}}
\]

might at first glance be coded as

FUNCTION F(X,N,M)

\[ F = 0.0 \]

DO 10 IR = 1, N

DO 10 IS = 1, M

10 \[ F = F + 1.0 / (4 \times IS - X)^{2 \times IR} \]

RETURN

END

However, by reversing the order of the summations so that

\[
f(x,n,m) = \sum_{s=1}^{m} \sum_{r=1}^{n} \frac{1}{(4s-x)^{2r}}
\]

and by using the replacements mentioned above, we can code the function as
FUNCTION F(X,N,M)
F = 0.0
XM = 4.0 * FLOAT(M) - X + 1.0E - 10 (1.0E - 10 added to insure loop completion)
S = 4.0 - X
OS2 = 1.0 / (S * S) (division done here instead of inside loop; see section J, below)
PROD = 1.0
DO 2 IR = 1, N
   PROD = PROD * OS2 (exponentiation replaced by multiplication)
END DO
2 F = F + PROD
S = S + 4.0 (multiplication replaced by addition)
IF (S .LT. XM) GO TO 1 (DO test on IS replaced by IF test on S)
RETURN
END

Another example of addition replacing multiplication is the loop in section 6, above, where a hidden multiplication in a subscript calculation is replaced by an explicit addition.

There is one case, however, where exponentiation to an integer power should not be replaced by multiplications: when the exponentiation only occurs once. The compiler usually sets up better code to do the exponentiation. Thus, X ** 4 is as (X * X) ** 2 which only requires 2 multiplications. In another example,

DO 30 I = 1, N
   PROD(I) = 1.0
   DO 30 J = 1, N
   30 PROD(I) = PROD(I) * 2.3 * B(J)

should be re-written as

TEMP = 2.3 ** N
DO 30 I = 1, N
   PROD(I) = TEMP
   DO 30 J = 1, N
   30 PROD(I) = PROD(I) * B(J)
Now, some optimization hints that are machine or compiler oriented.

A. Whenever possible, use DO or FOR loops that consist of only a single arithmetic-replacement statement (a CONTINUE statement not being counted as a statement here) because these are often optimized by the compiler to use index registers for holding loop controls, variable dimensions and needed constant increments. In order for such a loop to be optimized,

   a) the DO or FOR increment must be constant (e.g. DO 10 I = 2, N, 2)
   b) only INTEGER or REAL constants, variables or arrays can be used
   c) there can be no FUNCTION or statement function references
   d) each individual subscript must be in the form INTEGER variable ± constant, INTEGER variable, or constant (e.g. A(I+1) or B(J) or C(17))
   e) the innermost DO or FOR loop control variable can only appear once in any array subscript (e.g. DO 20 I = 1, N $ A(I,I) = I - 1 would prevent optimization)
   f) there must be no more than a total of four occurrences of either multiple subscripts (two- or three-dimension) depending on the innermost loop control variable or that variable appearing in the second or third subscript of a variable-dimension array (e.g. DO 30 I = 1, N 30 A(I, J) = B(J, I)).
   g) if the F-option is used on the MNF control card (Appendix A), there must be no variables or arrays in the loop appearing in an EQUIVALENCE statement.
   h) if the left-hand-side of the equal sign is an INTEGER variable, it must not appear in a subscript on the right-hand-side (e.g. DO 70 I = 1, N 70 L = I + A(L) would prevent optimization).

As a complete example, the form

   F = 1.
   DO 20 I = 2, M
   X = I
   20 F = F * X

is better written in the (non-standard) form:

   DO 20 I = 2, N
   20 F = F * I

In addition to using index registers, up to two variables or constants (elements which do not change within the loop) can be pre-loaded into active registers before the loop begins if all the above and following
conditions are met:

i) The only parenthesis groups that can appear are subscripts (e.g. DO 50 I = 1, L
50 X(I) = X(I) + B).

ii) There must be no exponentiation (***) operators.

iii) There can be at most four operands in the expression on the right-hand-side of the equal sign (or five if one of them is the same as the left-hand-side variable or array element, e.g. DO 60 K = M, I
60 II(K) = II(K) + JJ(K) / K / J).

iv) If the element is a constant, it cannot appear after a divide operator (/) (e.g. in DO 70 N = 3, 7
70 P(N) = P(N) / 6.9, 6.9 will not be pre-loaded).

v) If the element is a simple variable, it cannot be the left-hand-side variable, the loop control variable, or the loop terminal variable (e.g. in DO 30 I = 1, N, 3
30 A(I) = A(I) * N, N will not be pre-loaded).

vi) If the element is a subscripted variable, it must be singly subscripted and the subscript must not be the loop control variable; furthermore, the variable must not be the left-hand-side variable (e.g. in DO 40 L = 1, M
40 P(L) = P(L) + P(N) * Q(N), Q(N) will be pre-loaded but P(N) will not be pre-loaded).

Thus, in the loop

DO 100 I = 1, N
100 A(I) = B(I) * X

X is pre-loaded once before the loop begins.

A further optimization is made when the loop is a simple summation (Σ) or product (Π); the sum or product is not actually stored until the loop is completed if the above conditions a) through h) and the following hold:

1) The only parenthesis groups that can appear are subscripts.
2) There must be no exponentiation (***) operators.
3) If the left-hand-side of the equal sign is an array element, the subscripts must not depend on the innermost loop control variable and every appearance of the array on the right-hand-side must contain the identical subscripts as the left-hand-side.
Examples are:

```
DO 10 I = 1, N
  A(I) = 0.
DO 10 J = 1, L
10 A(I) = A(I) + B(I, J) * C(I, J)
   POLY = A(I)
DO 20 J = 2, N
20 POLY = POLY * X + A(J)
```

Because MNF does not optimize beyond one statement in a DO or FOR loop (although some compilers do so)*, making two one-statement loops into a single two-statement loop should be avoided. For example,

```
DO 1 I = 1, N
  1 B(I) = S + A(I)
  T = 0.0
DO 2 I = 1, N
  2 T = T + B(I)
```

should not be re-written as

```
  T = 0.0
DO 3 I = 1, N
  3 B(I) = S + A(I)
  T = T + B(I)
```

because the second case is not optimized whereas the first case is.

B. Attempt to optimize branch statements by choosing the fastest statements and putting decisions in order of decreasing likelihood of occurrence. The fastest branch statement compiled by MNF is the assigned GO TO, followed by the logical IF, arithmetic IF, and the computed GO TO statements. Thus when practical, replace an arithmetic IF by a logical IF since it is faster, may eliminate saving index registers or storing away control values, and allows the compiler to use values remaining in operating registers when needed for later computations. This replacement is especially good when 2 of the 3 branches of the arithmetic IF are the same because extra statement numbers often can also be removed.

* FTN (OPT=2) does optimize long DO loops.
For example,

```
DO 20 J = 1, K
    A(J) = B(J) * C(J)
    IF (J - 15) 10, 20, 10
10 A(J) = A(J) + 7.5
20 CONTINUE
```

is better written as

```
DO 20 J = 1, K
    A(J) = B(J) * C(J)
    IF (J .NE. 15) A(J) = A(J) + 7.5
20 CONTINUE
```

Similarly, replace computed GO TO statements having 3 or fewer statement numbers with arithmetic or logical IF's (e.g. GO TO (10,20,30),N should be IF (N-2) 10,20,30).

When a series of decisions must be made, they should be put in the order that makes the most likely relations take the first branches. For example, if one is examining data for college age students, the statements

```
IF (AGE .GT. 100) GO TO 100
IF (AGE .GT. 75) GO TO 75
IF (AGE .GT. 50) GO TO 50
IF (AGE .GT. 35) GO TO 35
IF (AGE .GT. 25) GO TO 25
IF (AGE .GT. 15) GO TO 15
```

would require 6 decisions on the average whereas the statements

```
IF (AGE .LE. 15) GO TO 5
IF (AGE .LE. 25) GO TO 15
IF (AGE .LE. 35) GO TO 25
IF (AGE .LE. 50) GO TO 35
IF (AGE .LE. 75) GO TO 50
IF (AGE .LE. 100) GO TO 75
```

require only 2 decisions on the average.

Logical IF statements containing multiple .OR. decisions or multiple .AND. decisions should be written in such a way as to make the most likely branch be taken first because of the way such statements are compiled. Thus,

```
IF (A .GT. B .OR. I .EQ. N .OR. P .LE. Q) GO TO 10
```
is compiled as if it had been written as

```fortran
IF (A .GT. B) GO TO 10
IF (I .EQ. N) GO TO 10
IF (P .LE. Q) GO TO 10
```

so that whichever relation is most likely to be true should be placed first in the statement (e.g. A .GT. B). Similarly,

```fortran
IF (C .GE. D .AND. F .LT. G .AND. K .NE. L) GO TO 30
```

is compiled as if it had been written as

```fortran
IF (C .LT. D) GO TO 40
IF (F .GE. G) GO TO 40
IF (K .NE. L) GO TO 30
```

```fortran
40 ... 
```

so that whichever of the original relations is most likely to be false should be placed first in the statement (e.g. C .GE. D) in order to skip over the less likely relational decisions. However, the programmer should keep in mind that MNF compiles parenthesis groups before looking at the .OR. and .AND. relations themselves. In particular, when a function evaluation is involved in a relation which is not likely to occur, it should be split off in a separate statement. Thus

```fortran
IF (X .GT. Y .OR. K .EQ. M .OR. P .LE. FCT(T)) GO TO 60
```

is compiled as if it has been written as

```fortran
TEMP = FCN(T)
IF (X .GT. Y) GO TO 60
IF (K .EQ. M) GO TO 60
IF (P .LE. TEMP) GO TO 60
```

so the original statement could better be written as

```fortran
IF (X .GT. Y .OR. K .EQ. M) GO TO 60
IF (P .LE. FCT(T)) GO TO 60
```

in order to minimize the number of unneeded evaluations of FCT(T). In the case where there are mixed .AND. and .OR. relations or the unary .NOT. in a logical IF, the complete logical expression within the IF statement must be evaluated before any decision can be made. This can be wasteful of time when one relation is more likely to be true or false than the others and such a relation should be split off and placed first in a separate IF statement.
SUBROUTINE calls, FUNCTION references and input/output operations require saving and restoring registers and control values, and cause the compiler to lose values in operating registers, so they should be done before a series of IF decisions if possible. Thus,

```
DO 10 I = 1, 1000
  IF (MOD(I, 10) .EQ. 0) GO TO 10
  L(I) = J(I)
  Y(I) = SIN(X(I)) * COS(X(I)) / FLOAT(J(I))
10 CONTINUE
```

should be re-written as

```
DO 10 I = 1, 1000
  TEMP = SIN(X(I)) * COS(X(I))
  IF (MOD(I, 10) .EQ. 0) GO TO 10
  L(I) = J(I)
  Y(I) = TEMP / FLOAT(J(I))
10 CONTINUE
```

Note that when the IF's almost always branch around the code, this should not be done. An example where SUBROUTINE calls and FUNCTION references should not be moved (since the branch is taken 986 times) is:

```
DO 10 I = 1, 1000
  IF (MOD(I, 10) .NE. 0 .OR. MOD(I, 7) .NE. 0) GO TO 10
  L(I) = J(I)
  Y(I) = SIN(X(I)) * COS(X(I)) / FLOAT(J(I))
10 CONTINUE
```

When 3-branch arithmetic IF statements are used, moving statements around can often improve decision-making. The reason is that the compiler looks at the statement following the IF and when its statement number is the same as one of the branches of the IF, different instructions are usually generated. Thus

```
IF (X) 10, 20, 30
10 A = B
```

is compiled as an equal-to-zero jump to 20 and a greater-than-zero jump to 30 (this form is best when $X < 0$ occurs least often),
IF (X) 10, 20, 30
20 C = D
is compiled as a less-than-zero jump to 10 and a non-zero jump to 30
(this form is best when X = 0 occurs least often),
IF (X) 10, 20, 30
30 E = F
is compiled as an equal-to-zero jump to 20 and less-than-zero jump to
10 (this form is best when X > 0 occurs least often) and
IF (X) 10, 20, 30
40 G = H
is compiled as an equal-to-zero jump to 20, a greater-than-zero jump to
30 and an unconditional jump to 10.

C. Decisions on the CDC 6600 are costly so they should be avoided whenever
possible. The compiler attempts to generate code with the minimum number
of decisions but the programmer can attempt to minimize them, also. One
method is to use the ABS, MIN, MAX, and SIGN type of function whenever
possible (i.e. IABS, ABS, DABS, MINO, MIN1, AMINQ, AMIN1, DMINS, DMIN1
MAXO, MAX1, AMAXO, AMAX1, DMAXO, DMAX1, ISIGN, SIGN, DSIGN). These have
been carefully written for MNF in such a way that no actual computer
decision is made. For example,
XMIN = 1.0E + 10
DO 10 I = 1, N
IF (XMIN .LT. X(I)) XMIN = X(I)
10 CONTINUE
is better written as
XMIN = 1.0E + 10
DO 10 I = 1, N
10 XMIN = AMIN1(XMIN, X(I))

D. Attempt to take advantage of "permanent" subscripting for two- or
three-dimensional arrays -- in certain cases, MNF will calculate a sub-
script only once in a DO or FOR loop when the subscript or similar sub-
scripts are used in one or more statements several times in the loop.
At the time the subscript or similar subscripts are found later in the
loop, the previously calculated "permanent" value is remembered and is not
recalculated.
A permanent subscript will be generated and saved by the compiler if

a) at least one of the variables in the original subscript is the innermost DO or FOR control and

b) all other parts of the original subscript are of the form +constant or variable+constant and that variable is any DO or FOR control in the nest or is a formal parameter used as a variable dimension.

(MNF requires that DO or FOR controls and variable dimensions not change in a loop and in a subprogram respectively.)

In addition, there must be no transfer statements appearing within the loop before the original subscript reference is seen, and the original subscript reference must not be in the dependent statement portion of a logical IF. Thus, moving a proper subscript reference before the first transfer statement in a loop can cause a permanent subscript to be generated.

In order for a later subscript to be similar to the original so as to be able to use a generated permanent value, when the array has constant dimensions each part can only differ from the original by a constant (e.g. for DIMENSION A(10, 10), B(10, 10), if the permanent subscript A(I, J) is generated it can be used for A(I + 1, J - 1) and for B(I - 1, J + 1)). However, when the array is variable dimensioned, if

a) the first dimension of a two-dimensional array is variable or

b) the first dimension of a three-dimensional array is variable

then only the first part may differ from the original by a constant to be considered similar (e.g. for DIMENSION A(I, 10), if A(K + 3, 7) is generated, it can be used for A(K - 1, 7) or for DIMENSION A(N, N, N), if A(I + 16, N, 36) is generated, it can be used for A(I, N, 36)).

If the first dimension of a three-dimensional array is constant but the second dimension is variable, then either the first or second part but not the third part may differ from the original to be considered similar (e.g. for DIMENSION A(6, L, K), if A(I - 1, J - 1, N) is generated, it can be used for A(I, J + 1, N)).
Example:

SUBROUTINE PEAK (D, E, M, L, CC, MM)

DIMENSION B(10, 10), C(20, 5, 6), D(M, 13), E(6, L, 9)

NML = N - 1

DO 200 I = 2, N
   DO 200 J = I, NML
      B(I - 1, J + 1) = D(J - 1, I)
      DO 100 K = I, J
         C(I, J, K) = CC
         C(I + 1, J + 1, K - 1) = C(I - 1, J + 1, K + 1)
         TEMP = E(K + 1, L, 7)
         IF (TEMP .EQ. 0.0) GO TO 100
         E(K + 1, L, 7) = E(K - 1, L, 7)
         E(K - 1, L, 7) = TEMP
   100 CONTINUE

200 D(J, I) = B(I, J) * C(I + 1, J + 1, MM - 1)
   DO 300 J = 1, N
   DO 300 I = 1, J
      C(I + 3, J, MM) = B(I, J)
      C(I, J + 1, MM - 2) = 1.0 + B(J, I + 1)
      BB = B(I, I)
      IF (BB .GT. CC) Z = D(I, J + 1)
      X = D(I, J) * D(I, J + 3)
      IF (BB .GT. CC) GO TO 300
      CC = CC + E(6, L, 6)
      X = X + E(6, L - 3, 6)
   300 CONTINUE

RETURN
END

It is interesting to note that if the seemingly useless statement
DO 300 MM = MM, MM is inserted before the statement DO 300 J = 1, N then
a permanent subscript will actually be generated for C(I + 3, J, MM) and
it will be used for C(I, J + 1, MM - 2).
E. Constants in the first subscript in a subscript reference of the form expression+constant or -constant are lumped with the address of the array by the compiler. In the case of an array having constant DIMENSION declarations, constants appearing last in all subscripts allow inclusion of them into the array address thereby simplifying the subscript calculation. However, the compiler will not do this if the subscript is constant+expression with the constant coming before the expression. Example:

\[
\text{DIMENSION } D(6, 5) \\
Y = D(K + 3, J + 3)
\]

The subscript is calculated as \((D + 14) + 6 \times J + K\) (see 2.6.1).

In the case of the variable dimensioned arrays, constants can be lumped with the address of the array if they appear properly (expression+constant) up through the first variable dimension of the array. Thus, if the first dimension is variable, only the first part can be lumped; if the second dimension is variable and the first is constant, both the first and second constant parts of the subscript can be lumped; and if only the third dimension is variable, all three parts of the subscript can have their constant portions lumped with the array address.

F. In a single statement, subscripts of the same array having constant DIMENSION declarations or of different arrays having the same constant DIMENSION declarations are only evaluated once by the compiler if they all use the same variables within the subscripts. A further requirement is that the subscripts each be in the form: variable+constant, variable, or constant. Thus in

\[
\text{DIMENSION } A(10, 20) \\
A(I, J) = A(I + 1, J - 1)
\]

only one subscript evaluation is made since the compiler knows that the two elements are 9 words apart. Another example is

\[
\text{DIMENSION } B(15, 5), C(15, 5) \\
A(I + 1, J) = B(I, J) - C(I, J)
\]

G. The compiler remembers result values previously stored which remain in active registers. For example, instead of
If one constant is to be stored in several variables, writing the statements consecutively is equivalent to the (non-standard) form of multiple replacement (see 4.5). It causes only one load or set command plus necessary store commands to be generated when the type of the constant is the same type as the variable. Thus

\[ \begin{align*} A &= 0 \\ B &= 0 \end{align*} \]

is better written as

\[ \begin{align*} A &= 0.0 \\ B &= 0.0 \end{align*} \]

Also, group variables of the same type together. Thus

\[ \begin{align*} C &= 1.0 \\ N &= 1 \\ D &= 1.0 \\ M &= 1 \end{align*} \]

is better written as

\[ \begin{align*} C &= 1.0 \\ D &= 1.0 \\ N &= 1 \\ M &= 1 \end{align*} \]

H. Identical subexpressions appearing within parentheses in a single statement are remembered by the compiler and evaluated only once. For example

\[ Z = -A + B - (A - B) / C \]

should be written as

\[ Z = (B - A) + (B - A) / C \]

to take advantage of this. A further requirement is that in the single statement if one subexpression is immediately preceded by a name (i.e. that of a FUNCTION or an array), all other identical subexpressions must be
preceded by the same name in order to be remembered. This allows complete
FUNCTION or array references to be calculated once and re-used in a statement.
However, this requirement means that extra parentheses are sometimes needed.
Thus,

\[ Q = \frac{1.0}{\sqrt{X}} \sin \left(\frac{1.0}{\sqrt{X}}\right) \]

must be re-written as

\[ Q = \left(\frac{1.0}{\sqrt{X}}\right) \sin \left(\frac{1.0}{\sqrt{X}}\right) \]

to take advantage of remembering the identical subexpression \( \frac{1.0}{\sqrt{X}} \).
Since subexpression remembering is only done within a single statement by
the compiler, the programmer must sometimes do the required work himself.
Thus

\[
\begin{align*}
\text{IF (G .GT. (B + C) * D) GO TO 10} & \quad \text{and} \\
D &= (B + C) * D \\
\text{should be re-written as} & \quad \text{should be re-written as} \\
\text{TEMP} &= (B + C) * D \\
\text{IF (G .GT. TEMP) GO TO 10} & \quad \text{CT} = \cos(\text{PHI}) \\
B &= \text{TEMP} & \quad \text{ST} = \sin(\text{PHI}) \\
\text{XP} &= X * \cos(\text{PHI}) + Y * \sin(\text{PHI}) & \quad \text{YP} = X * \sin(\text{PHI}) - Y * \cos(\text{PHI})
\end{align*}
\]

I. Constant subexpressions are evaluated during compilation if they
precede the first variable in a subexpression. For example, the constant
expressions in \( 2.0 * 6.0 / A \) or \( 6.4 + 7.88 + B \) or \( 1.0 / 3.0 * C \) will be
evaluated but the constant expressions in \( 2.0 / A * 6.0 \) or \( B + 6.4 + 7.88 \)
or \( C * 3.6 / 7.4 \) will not. Parentheses may be used to group constants into
a separate subexpression so in \( A * (3.6 / 7.4) \), the constant will be
evaluated during compilation. Note also that MNF allows expressions to
be used in DATA statements so with proper coding,
there should never be arithmetic involving only constants during execution.

J. As with many computers, division is quite a bit slower on the CDC 6600
than multiplication. Thus multiplication by the reciprocal will be faster.
The compiler automatically does this for constants appearing in a denominator:
\( X / 2.0 \) is done as \( X * 0.5 \) and \( Y / 10.0 \) is done as \( Y * 0.1 \). The programmer
should multiply by reciprocals when possible. Thus,
DO 10 J = 1, N.
DO 10 I = 1, N
10 C(I) = A(I) / (B(J) + C(J))

should be re-written as
DO 10 J = 1, N
T = 1.0 / (B(J) + C(J))
DO 10 I = 1, N
10 C(I) = A(I) * T

Note, however, that replacement of division with multiplication by the reciprocal does not always produce the same result. This can occur when the reciprocal cannot be exactly represented on the machine (e.g. 0.1 does not have an exact representation on a binary machine such as the CDC 6600).

K. If possible, avoid the unary operator minus preceding a variable since an extra subtraction may be needed to generate it. Thus, SIN(-X - 0.5) should be SIN(-0.5 - X) because -0.5 is a constant generated by the compiler.

L. Shifts and adds are generated by the compiler to do the operations of multiplication and division with small integer power-of-2 constants (e.g. 2, 4, 8, 16, etc.) in expressions and subscripts. Thus, the user need not do his own shifts and adds to accomplish what the compiler can do better. Also, the user should not replace such constants with variables having the same value since the compiler will no longer recognize their special power-of-two form. Thus,
ICHAR = 64
JCHAR = I / ICHAR

is better written as
JCHAR = I / 64

Similarly, INTEGER, REAL or DOUBLE PRECISION constants which are (positive or negative) powers of 2 involved with DOUBLE PRECISION operands in multiplication or division are done as an add-to-exponent or subtract-from-exponent operation which does not require a full double precision multiply or divide. Again, such constants should be used directly in a multiplication or division rather than replacing them with variables having the same value.
M. The constant 2 multiplying any variable having as high or higher type than the constant is done as an addition by the compiler, e.g. \(2.0 \times R\) is \(R + R\). Also, a REAL, DOUBLE PRECISION or COMPLEX expression divided by 0.5 is done as an addition. The user does not have to sacrifice clarity and write variable+variable himself.

N. Complex arithmetic expressions are done with improved code inserted by the compiler directly in the program -- rather than calling execution time functions. The following operations are done this way (I is an INTEGER operand, R is a REAL operand and C, Cl, C2 are COMPLEX operands):

\[
\begin{align*}
& Cl + C2, \quad Cl \times C2, \quad Cl / C2, \quad C ** I, \quad C / R, \quad R \times C, \quad C * R, \quad C + R, \quad R + C \\
& \text{(these last five are done without converting R to COMPLEX). The programmer does not have to program COMPLEX operations in REAL arithmetic to optimize. Furthermore, the programmer should attempt to use REAL variables or constants rather than convert them to COMPLEX when using them in COMPLEX expressions.}
\end{align*}
\]

There are similar cases in expressions involving DOUBLE PRECISION -- use REAL values whenever they are exactly representable in single-precision (e.g. 2.0). In a related case, a DOUBLE PRECISION comparison in a logical IF statement is not done in full double precision since only the sign of the result is needed, so avoid doing such a one-time calculation outside of the IF statement. A further remark is relevant here: beware of mixing REAL and DOUBLE PRECISION in an expression because unless the REAL values are exact, accuracy may be lost in the results.

O. When using programs that are being converted from other manufacturers' machines with smaller word-size (e.g. IBM/360, IBM/370 or UNIVAC 1107,1108) to CDC machines having 60-bit words, beware of the use of DOUBLE PRECISION arithmetic. The CDC machines have about 14.4 digits in REAL precision so DOUBLE PRECISION arithmetic used on other machines should be translated to REAL arithmetic by changing type statements, constants and FUNCTION references. DOUBLE PRECISION should only be used where such accuracy is truly needed especially since there is about a factor of 4 difference in speed.
In a similar situation, programs written on CDC machines that are later run on other manufacturers' machines may well have to be converted to DOUBLE PRECISION in order to run correctly. The differences in accuracy are often critical here.

P. After a program is completely debugged, compilation and paper can be saved if the NO REFERENCES and NO LIST statements, or \( R = 0 \) and \( L = 0 \) on the MNF control card are used. If possible, generating an overlay and running the resultant binary version of the program is even better.

Q. In order to program large Fortran array problems so that they will run efficiently on the large computers currently in use, the programmer must be aware of the underlying structure concept of data contiguity that these computers use to operate effectively. Thus the processing of large amounts of scattered data is inherently less efficient on these machines than processing contiguous data. Some of the machines are:

- CDC 6600 with arrays stored in ECS
- CDC 7600 with arrays stored in LCM
- CDC STAR 100 pipeline processor
- IBM 360/85 (double precision) with cache storage
- IBM 370/168 with the (double precision) array in virtual storage

An example such as the usual way of programming matrix multiplication in Fortran illustrates the problem.

```fortran
DO 110 I = 1, 400
  DO 110 J = 1, 400
    S = 0.0
    DO 100 K = 1, 400
      100 S = S + A(I, K) * B(K, J)
    110 C(I, J) = S
```

Here, the elements \( A(I, K) \) used inside the loop are 400 words apart (CDC machines) or 3200 bytes apart (double precision on IBM machines). This defeats the purpose of cache storage on the 360/85, causes excessive page faults on the 370/168, and runs 5 times slower than using the transpose of \( A \) on the CDC 7600. The loop can be written (after transposing \( A \) into \( D \)) to operate on contiguous data as
DO 110 I = 1, 400
DO 110 J = 1, 400
S = 0.0
DO 100 K = 1, 400
100 S = S + D(K, I) * B(K, J)
110 C(I, J) = S

For a 131072-word CDC 6600 assuming the arrays are on ECS, we can generate A-transpose (calling it D) by reading A in halves into the arrays E and F in central memory, transposing the halves, and writing them out as the array D.

```
  A 200 400
  200  
  400  
```

ECS / XX / A(400, 400), B(400, 400), C(400, 400), D(400, 400)
COMMON E(200, 200), F(200, 200)
DIMENSION G(400, 50), H(400, 50), P(400, 50)
EQUIVALENCE (E(1, 1), G(1, 1)), (F(1, 1), H(1, 1)), (E(1, 101), P(1, 1))
C READ IN TWO DIAGONAL BLOCKS
DO 10 I = 1, 200
   CALL READEC (E(I, I), A(I, I), 200)
10 CALL READEC (F(I, I), A(201, I + 200), 200)
C TRANSPOSE DIAGONAL BLOCKS
DO 20 I = 1, 199
   IP1 = I + 1
   DO 20 J = IP1, 200
      T = E(I, J)
      E(I, J) = E(J, I)
      E(J, I) = T
      T = F(I, J)
      F(I, J) = F(J, I)
   20 F(J, I) = T
C WRITE OUT TRANSPOSED BLOCKS
   DO 30 I = 1, 200
       CALL WRITEC (E(1, I), D(1, I), 200)
   30 CALL WRITEC (F(1, I), D(201, I + 200), 200)
C READ IN TWO OFF-DIAGONAL BLOCKS
   DO 40 I = 1, 200
       CALL READEC (E(1, I), A(1, I + 200), 200)
   40 CALL READEC (F(1, I), A(201, I), 200)
C TRANSPOSE OFF-DIAGONAL BLOCKS
   DO 50 I = 1, 200
       DO 50 J = 1, 200
           T = E(I, J)
           E(I, J) = F(J, I)
       50 F(J, I) = T
C WRITE OUT TRANSPOSED BLOCKS
   DO 60 I = 1, 200
       CALL WRITEC (E(1, I), D(1, 200 + I), 200)
   60 CALL WRITEC (F(1, I), D(201, I), 200)
C DO MATRIX MULTIPLICATION IN 8 PARTS, 50 COLUMNS AT A TIME
   DO 90 L = 1, 400, 50
       CALL READEC (H(1, L), B(1, L), 20000)
   DO 80 I = 1, 50
       DO 80 M = 1, 400, 50
           CALL READEC (G(1, I), D(1, L), 20000)
       DO 80 J = 1, 500
           S = 0.0
       DO 70 K = 1, 400
           70 S = S + G(K, J) \* H(K, I)
   80 P(M + I - 1, J) = A
   90 CALL WRITEC (P(1, 1), C(1, M), 20000)

While this code is longer than the other method, it is quite fast since the largest possible transfers are made using ECS.
Approximate Timings

In order to help the programmer understand the execution time of his program, the following table gives the approximate number of machine cycles for some of the various operations of the MNF compiler on the different CDC 6000/700C machines.

For the 6000 series a cycle = 100 nanoseconds
For the 7000 series a cycle = 27.5 nanoseconds
SPC are special cases
P is a constant

<table>
<thead>
<tr>
<th>Operation</th>
<th>Machine:</th>
<th>6200</th>
<th>6700CPI</th>
<th>6400</th>
<th>6500</th>
<th>6600</th>
<th>6700CPO</th>
<th>7600</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER LOAD</td>
<td></td>
<td>16</td>
<td>15</td>
<td>13</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTEGER STORE</td>
<td></td>
<td>14</td>
<td>13</td>
<td>11</td>
<td>10</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTEGER + or -</td>
<td></td>
<td>9</td>
<td>8</td>
<td>6</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTEGER *</td>
<td></td>
<td>60</td>
<td>59</td>
<td>57</td>
<td>10</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTEGER /</td>
<td></td>
<td>112</td>
<td>106</td>
<td>94</td>
<td>44</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPC INTEGER * 2^P</td>
<td></td>
<td>18</td>
<td>16</td>
<td>12</td>
<td>7</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPC INTEGER / 2^P</td>
<td></td>
<td>18</td>
<td>16</td>
<td>12</td>
<td>7</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REAL LOAD</td>
<td></td>
<td>16</td>
<td>15</td>
<td>13</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REAL STORE</td>
<td></td>
<td>14</td>
<td>13</td>
<td>11</td>
<td>10</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REAL + or -</td>
<td></td>
<td>24</td>
<td>22</td>
<td>18</td>
<td>8</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REAL *</td>
<td></td>
<td>60</td>
<td>59</td>
<td>57</td>
<td>10</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REAL /</td>
<td>SPC</td>
<td>60</td>
<td>59</td>
<td>57</td>
<td>29</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPC INTEGER - REAL</td>
<td></td>
<td>21</td>
<td>19</td>
<td>15</td>
<td>9</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOUBLE LOAD</td>
<td></td>
<td>34</td>
<td>32</td>
<td>28</td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOUBLE STORE</td>
<td></td>
<td>30</td>
<td>28</td>
<td>24</td>
<td>13</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOUBLE + or -</td>
<td>SPC</td>
<td>137</td>
<td>127</td>
<td>107</td>
<td>41</td>
<td>23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOUBLE *</td>
<td></td>
<td>300</td>
<td>292</td>
<td>276</td>
<td>34</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOUBLE /</td>
<td></td>
<td>468</td>
<td>451</td>
<td>417</td>
<td>118</td>
<td>81</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPC DOUBLE ± REAL</td>
<td></td>
<td>123</td>
<td>114</td>
<td>96</td>
<td>36</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 * D or D + D</td>
<td></td>
<td>58</td>
<td>54</td>
<td>46</td>
<td>19</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOUBLE * 2^P or DOUBLE / 2^P</td>
<td></td>
<td>137</td>
<td>134</td>
<td>128</td>
<td>18</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D ** 2 or D * D</td>
<td></td>
<td>240</td>
<td>233</td>
<td>219</td>
<td>28</td>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPLEX LOAD</td>
<td>34</td>
<td>32</td>
<td>28</td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPLEX STORE</td>
<td>30</td>
<td>28</td>
<td>24</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPLEX + or -</td>
<td>50</td>
<td>46</td>
<td>38</td>
<td>12</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPLEX *</td>
<td>292</td>
<td>284</td>
<td>268</td>
<td>32</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPLEX /</td>
<td>506</td>
<td>488</td>
<td>451</td>
<td>121</td>
<td>80</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPLEX ± REAL</td>
<td>34</td>
<td>31</td>
<td>25</td>
<td>7</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 * C or C + C</td>
<td>29</td>
<td>27</td>
<td>23</td>
<td>9</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPC</td>
<td>COMPLEX * 2(^p) or COMPLEX / 2(^p)</td>
<td>137</td>
<td>134</td>
<td>128</td>
<td>18</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C ** 2 or C * C</td>
<td>221</td>
<td>215</td>
<td>203</td>
<td>27</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPLEX / REAL</td>
<td>121</td>
<td>119</td>
<td>115</td>
<td>59</td>
<td>38</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Some Functions</th>
<th></th>
<th>Machine: 6400</th>
<th>6600</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACOS</td>
<td></td>
<td>1399</td>
<td>264</td>
</tr>
<tr>
<td>ALOG</td>
<td></td>
<td>1041</td>
<td>206</td>
</tr>
<tr>
<td>ASIN</td>
<td></td>
<td>1383</td>
<td>259</td>
</tr>
<tr>
<td>ATAN</td>
<td></td>
<td>927</td>
<td>216</td>
</tr>
<tr>
<td>ATAN2</td>
<td></td>
<td>1017</td>
<td>248</td>
</tr>
<tr>
<td>COS</td>
<td></td>
<td>1248</td>
<td>221</td>
</tr>
<tr>
<td>EXP</td>
<td></td>
<td>898</td>
<td>180</td>
</tr>
<tr>
<td>SIN</td>
<td></td>
<td>1230</td>
<td>208</td>
</tr>
<tr>
<td>SQRT</td>
<td></td>
<td>856</td>
<td>153</td>
</tr>
<tr>
<td>TANH</td>
<td></td>
<td>974</td>
<td>196</td>
</tr>
</tbody>
</table>
Summary List of Optimization Do's and Don'ts
(Page numbers in parentheses)

General Optimizations
1. In DO, FOR, or IF loops, move variables, expressions and decisions that do not depend on the current index variable outside the loop. (2)
2. Remove unnecessary statement numbers that are not being used. (3)
3. Place variables and arrays needed by subprograms into COMMON where practical. (3)
4. Use local variables in subprograms when a simple variable formal parameter would be needed many times within the subprogram. (5)
5. Use the DATA statement to prestore constants for once-only initialization. (6)
6. Use single-dimension instead of two- or three-dimension arrays when feasible. (7)
7. Consider a multi-dimension array to be single-dimension for setting all elements to a constant. (8)
8. Incorporate one-use subprograms into their calling or referencing program units. (9)
9. Use Horner's rule to evaluate polynomials. (10)
10. Evaluate constant subexpressions elsewhere (e.g. on paper) and replace the subexpression with that result. (10)
11. A ** (REAL integer constant) should be A ** (INTEGER constant). (11)
12. A constant k * 10 ** n should be written as kEn. (11)
13. Rewrite (-1) ** n to take advantage of the even/odd result. (11)
14. Avoid unneeded decisions in loops. (12)
15. Use the distributive law to minimize the number of multiplications. (14)
16. When possible, use library routines provided by the computer installation rather than writing your own. (14)
17. Attempt to make the innermost loop inside nested loops have operands that are in adjacent words and keep references to important arrays or variables in a single block, if possible. (14)
18. Be careful when using non-standard mixed-mode arithmetic. (15)
19. Attempt to replace exponentiations by multiplications, multiplications by additions and loop tests by equivalent tests. (16)
Machine or Compiler Oriented Optimizations

A. Use one-statement DO or FOR loops whenever possible. (18)
B. Choose the fastest branch statements for the situation and put multiple decisions in decreasing likelihood of occurrence. (20)
C. Avoid decisions whenever possible especially by using ABS, MIN, MAX, and SIGN functions. (24)
D. Attempt to take advantage of MNF permanent subscripting. (24)
E. Write subscripts with the constant portion last so that the compiler can lump the constant portion with the array address. (27)
F. Put subscripts in standard form in a single statement (if possible) to help the compiler evaluate the subscripts only once. (27)
G. Help the compiler remember result values previously stored which remain in active registers. (27)
H. In single statements, help the compiler to remember to evaluate identical subexpressions only once; in multiple statements, do this yourself. (28)
I. If you don't evaluate constant subexpressions yourself, put them before the first variable in a subexpression or use expressions in the DATA statement. (29)
J. Multiply by the reciprocal whenever the possible slight inaccuracy can be tolerated. (29)
K. Avoid the unary operator minus preceding a variable. (30)
L. Use power-of-2 constants in multiplication or division because the compiler makes a special attempt to recognize them. (30)
M. Use 2 * variable rather than variable + variable. (31)
N. Use REAL constants or variables in operations involving COMPLEX or DOUBLE PRECISION since the compiler can often take advantage of this to the user's benefit (but beware of possible loss of accuracy in mixing DOUBLE PRECISION and REAL). (31)
O. Use REAL rather than DOUBLE PRECISION if it does not affect the accuracy of the program. (31)
P. After debugging is finished eliminate the listing and cross-reference map or generate and use an overlay. (32)
Q. Program large Fortran array problems to use adjacent data. (32)
EXTRA FACILITIES ALLOWED BY MNF

This appendix describes the special features of MNF which will not work on RUN76 nor FTN (Version 3). Apart from Fortran II style statements (accepted by RUN76, but not by FTN), and the FTN debug statements (not allowed by RUN76), MNF will accept nearly all RUN76 and FTN statements. Major RUN76 and FTN features which will not work on MNF are described in the Handbook chapter FORTRAN. A detailed compatibility study of the three compilers is contained in Appendix A of this manual.

The MNF extensions fall into 2 main classes - the Trace statements (these are similar to the FTN debug statements, and are described in Appendix D); and the removal of many arbitrary restrictions from the Fortran language. Thus, in MNF, arithmetic expressions may be used in places where, in most versions of Fortran, only simple variables may appear. A third important MNF extension is that of format free I/O statements - so that coded input and output may be done without using FORMAT statements. This is described in Appendix E.

Note that the non-standard features of MNF which are also accepted by RUN76 and FTN are not described in this appendix, which may, therefore, be misleading when comparing MNF with non-CDC compilers.
1.0 The MNF TRACE statements

These are used for debugging and as an aid to improving the efficiency of programs. They are described in Appendix D of this manual. The statements available are:

(1) TRACE ARITHMETIC and NOTRACE ARITHMETIC
(2) " DOLOOPING   " " DOLOOPING
(3) " FORMATIO    " " FORMATIO
(4) " STATEMENT NUMBERS " " STATEMENT NUMBERS
(5) " SUBPROGRAM CALLS " " SUBPROGRAM CALLS
(6) " SUBPROGRAM ENTRY " " SUBPROGRAM ENTRY
(7) " SUBPROGRAM FLOW " " SUBPROGRAM FLOW
(8) " SUBSCRIPTS   " " SUBSCRIPTS
(9) " TRANSFERS    " " TRANSFERS
(10) " trace list   " " trace list
(11) TRACE trace list and NOTRACE trace list

2.0 The Listing Control Statements

These are used to control the compiler listings, and are described in the BKY HANDBOOK subset FORTRAN.

(i) LIST and NOLIST
(ii) CODE and NOCODE
(iii) REFERENCES and NOREFERENCES
(iv) PAGE

3.0 The IMPLICIT Statement

This is a statement taken from IBM Fortran, and is used to change the implicit (i.e., default) typing of Fortran names - variable, array, and user functions, not system library or built-in functions. The usual implicit typing is still that of names starting with I to N being Integer, and other names of type Real. This corresponds to that obtained with

IMPLICIT REAL(A-H,O-Z),INTEGER(I-N)
The IMPLICIT statement is useful for programs which use many names of type DOUBLE, or COMPLEX, or INTEGER, since each name need not be explicitly typed. E.g.

\texttt{IMPLICIT DOUBLE(D),COMPLEX(C,Z)}

will cause all names beginning with D to be of type double precision, and all names beginning with C or Z to be complex. Of course, names appearing in type statements will have the type declared regardless of any IMPLICIT statement.

Any number of IMPLICIT statements may appear in a routine provided that they all come between the header card (PROGRAM, SUBROUTINE, etc.), and the first specification statement.

The formal syntax of the statement is:

\texttt{IMPLICIT type_1(list_1),type_2(list_2),...,type_n(list_n)}

where type\textsubscript{i} is: REAL, INTEGER, COMPLEX, DOUBLE, (or DOUBLE PRECISION), or LOGICAL

and list\textsubscript{i} is: A list of items separated by commas -- an item being a single letter or two letters separated by a dash (minus sign). E.g., A,H means A and H, whereas A-H means A through H.

Examples:

1. \texttt{IMPLICIT INTEGER(A-Z)}

All names are type Integer, unless explicitly typed.

2. \texttt{IMPLICIT INTEGER(A-Z),REAL(I-N)}

Names starting with I,J,K,L,M or N are type Real, others are Integer. This example will produce 6 caution error messages because the types of I through N have been set twice.

3. \texttt{IMPLICIT DOUBLE(D)}

Names starting with D are Double precision, I through N are Integer, others are Real.
Note: Arguments (if any) in the SUBROUTINE or FUNCTION statement preceding the IMPLICIT are also affected by the IMPLICIT statement. Note that the IMPLICIT statement is not recognised by RUN76, nor (yet) by FTN.

4.0 Extensions to Type Statements

MNF provides a useful extension to type statements; which obviates the need for the COMMON statement. Note, however, that the IBM/360 extension of allowing /data/ to be pre-set using type statements is not allowed in MNF.

The type statements of MNF (and FTN and RUN76) are:

- REAL
- INTEGER
- DOUBLE (or DOUBLE PRECISION)
- COMPLEX
- LOGICAL
- (and DIMENSION)

In addition, any of the above (except DIMENSION) may be preceded by the word TYPE - as in the CDC FORTRANS - however the word TYPE is redundant and its use is not recommended.

Since the MNF extension applies equally to all of the above type statements I will restrict my comments to the REAL statement. Note that this new MNF feature could make programs hard to follow (and to debug) if it is used extensively.

The extension is to allow labelled common names enclosed in slashes to be inserted into the list of items following the type statement, thus indicating that the following (typed) items are to be put into the specified common block (in order of appearance).
If the (block) name is omitted [as in REAL//A(2),I], then the following list items are put into blank common.

Examples:

(1) REAL//A(2),I
    same as REAL A(2),I
        COMMON A,I
    or REAL I
        COMMON//A(2),I [etc.]

(2) REAL/I/J(6),X,Z(20)
    same as REAL J(6),Z(20)
        COMMON/I/J,X,Z

(there are many ways of writing this, but this way - of typing and then dimensioning first is the best - assuming that the MNF extension is not used).

    REAL A(11)//B/X/C(5),D//J(6)
    same as REAL A(11),J(6),C(5)
        COMMON B,J
        COMMON/X/C,D
    or REAL A(11),B,C(5),D,J(6)
        COMMON B,J/X/C,D [etc.]

(4) COMMON A(10),Z/P/Q(3)/I/ZZ
    REAL B(3)//M/I/J(3),N/R/X(5)
    same as REAL A(10),Q(3),B(3),M,J(3),N,X(5)
        COMMON A,Z,M/I/ZZ,J,N/P/Q/R/X

(5) The following is allowed, but a cautionary message is printed:

    REAL A(11)//A

It means REAL//A(11)

or REAL A(11)
    COMMON A
5.0 Arithmetic Expressions

Two new special "arithmetic" constants are allowed:

(1) .INF. or .R. ["infinity" on CDC 6000/7600 - e.g. 1/0]
(2) .IND. or .I. ["indefinite" value on CDC 6000/7600 - e.g. 0/0]

These may be useful in masking expressions, or in IF statements. Use of these constants will often cause arithmetic errors, which are detected by the hardware and cause the job to abort.

In addition primed (..., #) (8-4 punch) text constants may be used instead of (or as well as) Hollerith (H) constants. Like RUN76, MNF also allows L, R, and *characters* text constants to appear in expressions.

In addition †(11-5-8 punch) is allowed as an alternative to ** for exponentiation.

5.1 Logical, Masking, and Relational Expressions

Ten new relational operators are allowed:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Card Holes Punched</th>
</tr>
</thead>
<tbody>
<tr>
<td>≠</td>
<td>(not =) [same as .EQ.]</td>
<td>0-6-8</td>
</tr>
<tr>
<td>&lt; or</td>
<td>.LS. [&quot;&quot; .LT.]</td>
<td>12-0</td>
</tr>
<tr>
<td>&gt; or</td>
<td>.GR. [&quot;&quot; .GT.]</td>
<td>11-7-8</td>
</tr>
<tr>
<td>≤ or</td>
<td>.LQ. [&quot;&quot; .LE.]</td>
<td>5-8</td>
</tr>
<tr>
<td>≥ or</td>
<td>.GQ. [&quot;&quot; .GE.]</td>
<td>12-5-8</td>
</tr>
<tr>
<td>.NQ. [&quot;&quot; .NE.]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In addition, four new logical (or 'masking') operators may be used:-
Card Holes Punched

(1) \(\land\) [same as \(A\) or \(\cdot AND.\)] 0-7-8
(2) \(\lor\) [" " \(\cdot\) or \(\cdot OR.\)] 11-0
(3) \(\neg\) [" " \(\cdot N.\) or \(\cdot NOT.\)] 12-6-8
(4) \(\cdot X.\) [" " \(\cdot X\cdot OR.\) (exclusive \(\cdot OR.\)]

The main advantage of 5.0 and 5.1 is that single (CDC) characters may be punched, where at least 4 characters are required by standard FORTRAN e.g.

IF(A.GT.B.\(\cdot AND.\)B.LE.I**3.\(\cdot OR.\)).....

may, in MNF, be written:

IF(A>B\(\cdot AND.\)=I +3\(\cdot OR.\))...

with a saving of 14 characters (out of 25). The disadvantage is that the MNF forms will fail to compile under most other versions of FORTRAN. (Also, only IBM 029 card punches have separate keys for all of these special characters).

The special forms \(\cdot NQ.\), \(\cdot LQ.\), \(\cdot LS.\), \(\cdot EQ.\), and \(\cdot GR.\) seem to have no advantages over the standard forms. Note that \(\equiv\) is used for \(\cdot EQ.\) : if \(\equiv\) were used then the syntax could be ambiguous due to the multiple replacement statement. E.g:

\[
\begin{align*}
A &= B = C \\
\text{means} &
B = C \\
A &= B
\end{align*}
\]

But

\[
\begin{align*}
A &= B \equiv C \\
\text{means} &
A = B.\cdot EQ.\cdot C
\end{align*}
\]

5.2 Extra Functions in MNF

Some of these are intrinsic, and some external.
The functions listed below are all intrinsic (i.e., built in) and do not reference the library. The object of these functions is to allow the Fortran user to simulate machine language programming, but some of the functions (e.g. SHIFT, and ICOUNT) are of general use.

<table>
<thead>
<tr>
<th>Sample Call</th>
<th>Explanation</th>
<th>COMPASS Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = DAD(B,C)</td>
<td>A=B+C (double precision)</td>
<td>DX6 X1+X2</td>
</tr>
<tr>
<td>A = DMU(B,C)</td>
<td>A=B*C</td>
<td>DX6 X1<em>X2 or IX6 X1</em>X2</td>
</tr>
<tr>
<td>A = DSB(B,C)</td>
<td>A=B-C</td>
<td>DX6 X1-X2</td>
</tr>
<tr>
<td>A = FAD(B,C)</td>
<td>A=B+C (unrounded single precision)</td>
<td>FX6 X1+X2</td>
</tr>
<tr>
<td>A = FDV(B,C)</td>
<td>A=B/C</td>
<td>FX6 X1/X2</td>
</tr>
<tr>
<td>A = FMU(B,C)</td>
<td>A=B*C</td>
<td>FX6 X1*X2</td>
</tr>
<tr>
<td>A = FSB(B,C)</td>
<td>A=B-C</td>
<td>FX6 X1-X2</td>
</tr>
<tr>
<td>J = ICOUNT(B)</td>
<td>J=count of 1 bits in B</td>
<td>CX6 X1</td>
</tr>
<tr>
<td>A = NORM(B)</td>
<td>A=normalized B</td>
<td>NX6 X1</td>
</tr>
<tr>
<td>X = PACK(I,J)</td>
<td>X=J*2**I</td>
<td>SB7 X1 and PX6 B7,X2</td>
</tr>
<tr>
<td>A = RAD(B,C)</td>
<td>A=B+C (rounded single precision)</td>
<td>RX6 X1+X2</td>
</tr>
<tr>
<td>A = RDV(B,C)</td>
<td>A=B/C</td>
<td>RX6 X1/X2</td>
</tr>
<tr>
<td>A = RMU(B,C)</td>
<td>A=B*C</td>
<td>RX6 X1*X2</td>
</tr>
<tr>
<td>A = RSB(B,C)</td>
<td>A=B-C</td>
<td>RX6 X1-X2</td>
</tr>
<tr>
<td>L = ICOEF(X)</td>
<td>L=coefficient of X</td>
<td>UX6 X1</td>
</tr>
<tr>
<td>L = IUEXP(X)</td>
<td>L=exponent of X</td>
<td>UX6 B7,X1 and SX6 B7</td>
</tr>
</tbody>
</table>

**A = AND(B,C) | A=B.AND.C | BX6 X1*X2 |

**A = EQV(B,C) | A=(B.AND.C).OR..NOT.(B.OR.C) | BX6 -X1-X2 |

**A = IMP(B,C) | A=.NOT.B.OR.C | BX6 -X1+X2 |

**A = COMPL(B) | A=.NOT.B | BX6 -X1 |

**A = OR(B,C) | A=B.OR.C | BX6 X1+X2 |

**A = XOR(B,C) | A=(B.OR.C).AND..NOT.(B.AND.C) | BX6 X1-X2 |

**A = SHIFT(B,C) | A=B shifted left circularly C bits if C>0 | SB7 X2 and LX6 B7,X1 |
| | A=B shifted arithmetic right C bits if C<0 |

*A = MASK(B) | A=B bits on in left of word. | MX6 B |

Note: B must be a constant (not a variable).

*These functions are also intrinsics in FTN

**These functions are also intrinsics in F77, and FTN.
6.0 **Allowed Uses of Expressions**

One of the main advantages of MNF syntax is that expressions may be used where normally only simple variables or constants are allowed. This removes many unnecessary and irritating restrictions, without introducing ambiguity. Arithmetic expressions may be used:

1) As DO statement parameters (integer expressions)
2) As FOR statement parameters (integer expressions)
3) As implied DO parameters (in I/O lists) (integer expressions)
4) As output list elements (any expressions)
5) As subscripts of arrays (any expressions)
6) As computed GOTO control 'variables' (any expressions)

6.1 **Subscripts**

MNF allows any arithmetic expression as a subscript - including function references and exponentiation, e.g.

\[ A(5^{**}I,SIN(X),A(J)) \]

Note that a maximum of 3 subscripts is allowed, as in FTN and RUN76.

7.0 **Computed GOTO**

The syntax of the MNF Computed GOTO is:

\[ \text{GOTO}(l_1,l_2,\ldots,l_n),\text{exp} \]

where \( l_1, \ldots, l_n \) are statement numbers (not **AS**IGNed label variables),

and **exp** is any arithmetic expression, and the comma before exp may be omitted.
exp will be truncated to type Integer before the statement is executed. Examples:

1) GOTO(1,2,3),I+J

2) GOTO(10,5,9,6)X(J)

which is the same as

GOTO(10,5,9,6)IFIX(X(J))

3) GOTO(3,21)(3.9,0.7) [complex constant]

which is the same as

GOTO(3,2,1)IFIX(REAL(3.9,0.7))

i.e.

GOTO(3,2,1)IFIX(3.9)

i.e.

GOTO 1

See also TRACE TRANSFERS in Appendix D for errors in execution of Computed GOTO statements.

8.0 Logical IF Statement

In MNF, the dependent statement of a logical IF may be any executable statement other than a DO, FOR, or END statement. In particular it may be another IF statement. E.g.

1) IF(A.EQ.B) IF(I) 2,7,2

2) IF(A.LT.B) IF(J.EQ.0) J=K

9.0 DO and FOR statements

The FOR statement is much the same as the DO statement - except that the step (third parameter) is negative. I.e.
F\(\text{FOR}\) \(n \ i = m_1, m_2, m_3\)

is the same as

D\(\text{DO}\) \(n \ i = m_1, m_2, (-m_3)\)

As in RUN76 and FTN, D\(\text{DO}\) (and implied D\(\text{DO}\)) steps \((m_3)\) must not be negative in MNF. Similarly, F\(\text{FOR}\) steps must not be negative, since they are assumed to be negative (i.e. their sign is reversed).

F\(\text{FOR}\) 4 \(I = 10, -10, 2\) [allowed]

is the same as

D\(\text{DO}\) 4 \(I = 10, -10, -2\) [not allowed]

and

F\(\text{FOR}\) 5 \(I = 10, 1\) [allowed]

is the same as

D\(\text{DO}\) 5 \(I = 2, 10, 1, -1\) [not allowed]

So, if the F\(\text{FOR}\) step is omitted, it is assumed to be -1 (rather than +1 as in a DO statement).

Since no step \((m_3)\) may have a negative value the following would also be illegal

F\(\text{FOR}\) 2 \(I = 10, 1, -2\) [use F\(\text{FOR}\) 2 \(I = 10, 1, 2\) since the - sign is assumed].

The validity of D\(\text{DO}\) statement parameters is automatically checked when a TRACE DO LOOPING is in operation (see Appendix D).

F\(\text{FOR}\) loops may be 'nested' just as D\(\text{DO}\) loops can be. In addition a mixture of F\(\text{FOR}\) and D\(\text{DO}\) loops can be nested. Apart from the 'negative step', F\(\text{FOR}\) statements are treated like D\(\text{DO}\) statements by the MNF Compiler.
9.1 Both DO and FOR statement parameters may be Integer expressions. E.g.

\[ \text{DO 2 I = J+K, L+M, N/4} \]

or

\[ \text{FOR 3 K = IFIX(X), 27+J} \]

9.2 In addition, an optional comma may be used after the statement number - as in:

\[ \text{DO 2, I = 1,2,3} \]

This usage is not recommended - MNF allows it because many beginners insert a comma there by mistake.

9.3 In MNF, the terminal statement of a DO or FOR range may be a logical or arithmetic IF statement (not allowed in ANSI or FTN). E.g.

\[ \text{DO 2 I = 1,50} \]
\[ \text{2 IF(A(I).EQ.B(I)) GOTO 3} \]
\[ \text{PRINT 1} \]
\[ \text{1 FORMAT(1X, \#NO ELEMENTS EQUAL\#)} \]
\[ \text{3 \ldots} \]

10.0 FORMAT Statements

One 'new' conversion is available (\#string\#), and one 'new' specification (Tn).

10.1 Primed (\#...\#) conversion

This is very similar to * conversion, but is preferable to it since * conversion is (more or less) only available on CDC machines, whereas primed conversion is widely available. Note that primed conversion is available in FTN, and in BKYIO (not the standard RUN library). The MNF implementation of primed
conversion is compatible with that of FTN. It is not quite compatible with the IBM/360 implementation in that the IBM version treats 2 contiguous embedded primes as a single prime, whereas MNF (etc.) ignores 2 consecutive embedded primes.

E.g. PRINT 10

10 FORMAT(1X,"DON'T")

would print DONT under MNF (etc.) and DON'T under /360.* Note that

10 FORMAT(1X,"DON'T")

would be illegal under all versions. To print DON'T using MNF use * or H conversion. E.g.

10 FORMAT(1X,5H"DON'T")
10 FORMAT(1X,"DON'T")

Note: The strange character ≠ (8-4 punch) is chosen as a delimiter, because it is represented as a'(apostrophe, or prime) on IBM /360 equipment, and on many CDC 6000/7600 machines.

10.2 T Specification

This is compatible with the FTN Tn specification - except for TO which is treated as T1 by FTN, but is an error under MNF.

The standard CDC RUN76 library does not accept T specification, but the BKYIO library does.

11.0 "Free Format" I/O Statements

This is a very useful extension, which allows "formatted" I/O to be done - without reference to a FORMAT statement.

Free Format I/O is described in Appendix E of this Manual.

*FTN Version 4 is compatible with IBM in this regard.
12.0 **END and ERR in READ**

The END and ERR parameters in READ statements provide simple means for controlling what happens when an end of file condition occurs; or when there is an I/O error - e.g. incorrect data. These parameters are described in Appendix E of this manual, and are similar to the IBM/360 Fortran facility (but not the same).

13.0 **Form of MNF I/O Lists**

These are more powerful than the FTN and RUN76 form.

13.1 **Input Lists**

These are used in READ, BUFFERIN, and DECODE statements.

The MNF extension is that any 'implied DO' specifications in the list may follow the extended rules of the MNF DO statement (see section 9.1). Note that the step of any implied DO must be positive (since there is no such thing as an "implied FOR loop"). Apart from this restriction, there is the following extension:

In the statement:

```
READ(a,b)(l_1,l_2,l_3, .... ,i=m_1,m_2,m_3)
```

any of \(m_1, m_2, m_3\) may be expressions of type integer).

```
READ(5,10)(A(I),I=1, 27+J, J/3)
```

13.2 **Output Lists**

These are used in WRITE, PRINT, PUNCH, and ENCODE statements. The extended MNF form incorporates the extension described under 13.1. In addition, output lists may contain expressions of any type (including constants) as list elements.
E. g.  
\[
\text{PRINT 5,5,} \neq X=\neq 5 \times X \\
5 \text{ FORMAT(1X,I1,A2,F10.1)}
\]
which would print (with X=1.1)

\[
5X = 5.5
\]

[more easily written as \text{PRINT,}\neq 5X=\neq ,5*X See Appendix E ]

The only restriction on the expressions (list items) is that they must not contain a function reference if that function itself contains any I/O statements. So that, since \text{SIN} does no I/O;

\[
\text{PRINT 5,} (\neq \text{\text{SIN}}=\neq , \text{SIN(FLOAT(I))}, \text{I}=1, \text{N+M)} \\
5 \text{ FORMAT(1X,A3,L4,1H=} , \text{F10.6})
\]

would be allowed. Note that \text{***} strings are not allowed in output lists. This is the only restriction on their use, and they \text{are} allowed in FORMAT Statements

14.0 \text{File Control Statements}

The file control statements of \text{MNF} are

\[
\text{REWIND list} \\
\text{BACKSPACE list} \\
\text{ENDFILE list}
\]

where list is a series of integer constants (0.LT.n.LE.99), and/or simple integer variables - separated by commas. (in FTN and RUN76 only one integer constant or variable is allowed), E.g.

\[
\text{REWIND 1,2,1 is allowed by MNF}
\]

15.0 \text{DATA Statements}

\text{MNF accepts the non-standard CDC forms for DATA statements.}
APPENDIX D

The MNF TRACE Statements

The statements described in this Appendix are all non-standard and their use causes note level diagnostics to be given. However, they provide very useful ways for the programmer to detect errors during the execution of his program, and to give additional information about the source program.

The TRACE and NOTRACE statements are normal non-executable Fortran statements. They should not be labelled, but are punched in Column 7 onwards -- and blanks are not significant.

It is recommended that the D option of the MNF control card and C$ in Columns 1 and 2 be used with the TRACE statements, since this method allows the program to run unchanged with other Fortran compilers where the statements will be treated as comments. The method also can be used to ignore the statements during later compilation by MNF. Note that the MNF TRACE statements are quite different from the FTN Debug statements, and that RUN76 will accept no such statements.

1.0 TRACE and NOTRACE Statements

With these statements, a programmer can establish automatic outputs and checks of various types as the program is executed. They provide a better and simpler method than the insertion of PRINT statements or the use of core-dumps for locating errors and checking program flow, and can aid in the optimization of frequently-run or long programs. They can also greatly help other users who wish to modify unfamiliar programs. Extra output is minimized by most of the statements in that they only print when errors occur or give summaries at the end of the program. They are keyed to the user's program either through the sequence number or the octal address assigned to each statement.

The statements have a number of forms. Tracing begins when initiated by a TRACE statement and continues until terminated by a corresponding NO TRACE statement or by the end of the program and all subprograms, except in the case of the TRACE statements described in 4.0. Several of the tracing forms are automatically turned on by the T option of the MNF control card and begin tracing when the first statement is encountered. These too can be individually turned off at any point by using NOTRACE statements.

There are two kinds of TRACE statements -- frequency and debugging. The frequency statements give a usage count after the program has finished executing and are best suited for improving program efficiency. They show where programs or subprograms spend most of their time or which subprograms are used most frequently. From the information they give, it can be determined which areas of the program need most attention for optimization. Their value is mostly for production programs that have been debugged and which are run frequently or use much computer time whenever they are run. To some extent, frequency statements can also be used for debugging since they show areas which were never executed. This same information can be used for optimization if the unused areas are removed. The use of frequency statements causes execution time to be increased, so they should only be used for a few program runs and removed afterwards, or the T or D options on the MNF control card turned off. The frequency information is printed even if Fortran-detected errors occur which abort the program.

Note: A reasonably complete introduction to the MNF trace statements is given in the HANDBOOK subset FORTRAN.
Debugging TRACE statements give printouts to show the flow of a program or errors detected during program execution. Their chief use is during the debugging phase of a program in which the compilation is correct but answers are incorrect or execution errors exist whose cause cannot easily be found. Debugging TRACE statements can cause much slower execution, and will usually increase printed output sometimes to the point of page limit if many errors are detected. For example:

\begin{verbatim}
C$ TRACE SUBSCRIPTS
  COMMON A(10),B(10)
  DO 2 I=1,10000
    A(I)=0.0
  2 B(I)=0.0
\end{verbatim}

would produce 19980 error messages, the number of times the subscript I was greater than 10. The library routine TRACSET may be used to limit TRACE output. See Section 8.0. Once a program has been debugged, the TRACE statements should be removed, or the T or D options on the MNF control card turned off, unless further protection is desired at the expense of slower execution as in the case of subscripts being read in from data. Note that the TRACE statements will slow down execution of correct programs as well as wrong ones, since all the tests must still be done. Usually, the T option will slow down execution by about 50%, which is not an excessive penalty. The most expensive trace is (usually) TRACE SUBSCRIPTS, followed by TRACE ARITHMETIC. The other traces are fairly cheap. The use of (T) should never slow down the program by more than a factor of 2.

If any errors in the TRACE or NO TRACE statements themselves are detected, a caution level diagnostic is given but it is not fatal. For example, if a NO TRACE statement appears but there is no corresponding previous TRACE statement, the NO TRACE is ignored. Also, TRACE and NO TRACE statements should not have statements number labels on them, but if they do, the labels will be ignored.

Under some circumstances, use of TRACE statements can cause a recursive attempt to use the system output routine which results in a fatal execution error message being printed on the OUTPUT file, after which the program will be terminated. Recursive use would occur if a traced array element subscript bound is exceeded in an output list, if there is illegal indexing in an implied DO loop that is being traced, or if the evaluation of a function in an output list causes trace output. Otherwise, only in the case described in 3.5 will tracing cause a program to be terminated. All other tracing may cause error messages to be printed but will not cause program termination by itself.

2.0 The frequency TRACE and NO TRACE statements forms are:

- **(1) TRACE STATEMENT NUMBERS** (Turned on by MNF(T))
  - NO TRACE STATEMENT NUMBERS
- **(2) TRACE SUBPROGRAM CALLS** (Turned on by MNF(T))
  - NO TRACE SUBPROGRAM CALLS
- **(3) TRACE SUBPROGRAM ENTRY**
  - NO TRACE SUBPROGRAM ENTRY
The debugging TRACE and NO TRACE statement forms are:

(4) TRACE ARITHMETIC
   NO TRACE ARITHMETIC
(5) TRACE DOLOOPING (Turned on by MNF(T))
   NO TRACE DOLOOPING
(6) TRACE FORMATIO (Turned on by MNF(T))
   NO TRACE FORMATIO
(7) TRACE SUBSCRIPTS (Turned on by MNF(T))
   NO TRACE SUBSCRIPTS
(8) TRACE TRANSFERS (Turned on by MNF(T))
   NO TRACE TRANSFERS
(9) TRACE SUBPROGRAM FLOW
   NO TRACE SUBPROGRAM FLOW

The above frequency and debugging TRACE statements are global in that they will stay on during more than one program or subprogram unless turned off by a corresponding NO TRACE statement (i.e., an END line does not turn them off). The debugging forms below may only be used locally and are active during the current program or subprogram. They are also the only forms which have a list.

(10) TRACE list
    NO TRACE list

The following statements complement the T option on the MNF control card. They turn on or off the same group of tracing forms as the T option turns on but these may be inserted anywhere within the program.

(11) TRACE and NOTRACE

2.1 TRACE STATEMENT NUMBERS

NO TRACE STATEMENT NUMBERS

When TRACE STATEMENTS NUMBERS is used, a count is kept of the number of times each statement number was executed and the number of times each FORMAT statement number was used in each program and subprogram. For FORMATS, the count is set up when the FORMAT number is encountered in an I/O statement. At the end of the program, the program and each subprogram name is printed on the OUTPUT file, followed by the list of statement numbers and format numbers sorted in ascending order for that program unit, and the number of times each was executed or was used within that program unit. In this way, sections of the program or formats may be examined for unusually heavy or light usage.

NO TRACE STATEMENT NUMBERS stops the insertion of further code to set up an increment count for any later statement or format numbers. At the end of the program, only statement or format numbers that had an increment count set up (or that were in a trace list) are printed. (Statement or format numbers that were in a trace list but did not have an increment count set by virtue of being between TRACE STATEMENT NUMBERS and NO TRACE STATEMENT NUMBERS will have a zero count. See Section 4.9.)
2.2 TRACE SUBPROGRAM CALLS

NO TRACE SUBPROGRAM CALLS

A count is kept of the number of times each subroutine is called or each function is referenced when TRACE SUBPROGRAM CALLS is used. At the end of execution, the subprogram or entry name and the number of times it was called or referenced is printed on the OUTPUT file. Library functions, such as SQRT, are also counted in the table. For calls only, the maximum count that can be correctly printed is 262143.

NO TRACE SUBPROGRAM CALLS stops the insertion of code to increment the count for each subprogram. At the end of execution, the subprogram and count list are printed but only those calls between the TRACE SUBPROGRAM CALLS and NO TRACE SUBPROGRAM CALLS will be counted.

2.3 TRACE SUBPROGRAM ENTRY

NO TRACE SUBPROGRAM ENTRY

Each time a subprogram is entered, when TRACE SUBPROGRAM ENTRY is in effect, a count is incremented and at the end of execution, the list of subprograms or entry points and the number of times they were entered is printed on the OUTPUT file.

This statement in conjunction with TRACE SUBPROGRAM CALLS tells how many times subprograms are used. TRACE SUBPROGRAM CALLS is needed because library subprograms cannot have TRACE SUBPROGRAM ENTRY statements, and TRACE SUBPROGRAM ENTRY is needed when COMPASS subprograms or subprograms compiled by other Fortran compilers call MNF-compiled subprograms. Another example would be a library integration routine which uses a programmer-supplied function. In general, TRACE SUBPROGRAM CALLS is the most useful, and if both are turned on, the totals at the end can be up to twice the number of times the subprogram was actually executed. NO TRACE SUBPROGRAM ENTRY stops the insertion of code to increment the count in each subprogram. At the end of execution, the subprogram name and count list are printed, but counts will be zero for subprograms or ENTRY statements which appeared after the NO TRACE SUBPROGRAM ENTRY statement, since their counts were never incremented.

3.1 TRACE ARITHMETIC

NO TRACE ARITHMETIC

When TRACE ARITHMETIC is in effect, before every store of an arithmetic result a check is made that a real or double precision or complex value is not out of range or undefined, that an integer value is < 2^48 (=281 474 976 710 656), or that a double precision value does not have a zero lower precision part with non-zero upper precision part (this occurs in the range 1.0D-292 to 1.0D-278, which has only single precision accuracy). The main purpose of this statement is to detect storage of values that will later give indefinite or infinite results - or cause erroneous results to be calculated. In this way, the user can see where a bad value was generated, instead of discovering an error caused by the value many statements or subprograms later. However, not all such errors will be caught by this statement, because intermediate results can overflow before the check is made, causing a hardware trap even before any values are stored. In this case, examination of the core dump will usually give enough information to detect the bad value. Checking of stored values is done instead of checking every calculation, because the latter could be extremely expensive in computer time. If an attempt is made to store any of the above described bad values, a message is printed on the OUTPUT file.
*TRACE* STORE INTO type VARIABLE name AT SEQUENCE NUMBER n* VALUE = k octalB
(where type is one of REAL, INTEGER, DOUBLE, or COMPLEX, and octalB is the octal value); and then the value is stored. In order to output complete decimal values, the octal equivalent of double precision and complex values is not printed.

Note: Because of a difference in design between the 7600 and 6000 computers, TRACE ARITHMETIC is not useful on the 7600 unless a MODE(O) control card is inserted before the MNF card. TRACE ARITHMETIC is not automatically turned on by MNF(T).

When a NO TRACE ARITHMETIC statement is encountered, no further checking is made of values to be stored.

3.2 TRACE DOLOOPING

NO TRACE DOLOOPING

During execution under the control of TRACE DOLOOPING, checks are made of each DO or FOR loop or implied DO loop that the initial value is not greater than the final (or that the final value is not greater than the initial value in FOR loops), that the increment is not zero or negative, that terminal + increment is not greater than 131071 for DO loops, or that terminal - increment is not less than -131071 for FOR loops, or that the absolute value of an indexing parameter is not greater than 131071. If an attempt is made to execute a loop with any of these conditions, one of the following messages is printed on the OUTPUT file:

*TRACE* DO LOOP INITIAL VALUE k AT SEQUENCE NUMBER n*
GREATER THAN TERMINAL VALUE = m
*TRACE* FOR LOOP INITIAL VALUE k AT SEQUENCE NUMBER n*
LESS THAN TERMINAL VALUE = m
*TRACE* DO/FOR INCREMENT VALUE k .LE. 0 AT SEQUENCE NUMBER n*
*TRACE* DO LOOP TERMINAL + INCREMENT AT SEQUENCE NUMBER n*
GREATER THAN 131071
*TRACE* FOR LOOP TERMINAL-INCREMENT AT SEQUENCE NUMBER n*
LESS THAN -131071
*TRACE* DO/FOR INCREMENT PARAMETER AT SEQUENCE NUMBER n*
ABS VALUE .GT. 131071
*TRACE* DO/FOR INITIAL PARAMETER AT SEQUENCE NUMBER n*
ABS VALUE .GT. 131071
*TRACE* DO/FOR TERMINAL PARAMETER AT SEQUENCE NUMBER n*
ABS VALUE .GT. 131071

(where n* is the sequence - or source line - number of the statement in error), after which control is returned to the DO or FOR loop and the loop is executed as written. Note that if terminal + increment is greater than 131071 for DO loops, or if terminal - increment is less than -131071 for FOR loops, it is
possible that no program error will occur if the illegal value is not actually achieved during execution of the loop -- i.e., when last value of loop + increment is less than or equal to 131071 for DO loops, or last value of loop - increment is greater than or equal to -131071 for FOR loops. For example, DO 10 I = 1,131070,2 will work although an error message is given. (However, DO 10 I = 2,131070,2 will not work).

Because 18-bit registers are used for DO and FOR loop control variables and parameters, DO/FOR tracing first checks to see if these numbers from 60-bit registers will fit (i.e., they must be less than 131071 in absolute value). Then the bottom 18 bits of each of the parameters are placed in the 18-bit registers where further parameter checks are made. Since the uppermost bit of the 18-bit register is its sign bit, the register value may be negative or positive even though the 60-bit number from which the bottom 18 bits were taken has a different sign. The result is that spurious trace error messages may be given when DO/FOR checks are made on the 18-bit registers.

NO TRACE DOOOPING stops further checking of DO, FOR, or implied DO loops.

If all of the indexing parameters are constants, checking of these is done at compile time and thus during executing there will not be any TRACE messages for indexing errors involving only constants.

3.3 TRACE FORMATIO

NO TRACE FORMATIO

Whenever a formatted READ, WRITE, PRINT, PUNCH, ENCODE, or DECODE statement is executed and TRACE FORMATIO is in effect, a check is made that each FORMAT specification implies a correct type for the corresponding list elements. If the format specification is E, F, or G, the list element must be real or complex. If the specification is D, the list element must be double precision; if it is I, the list element must be integer; if it is L, the list element must be logical. If the specification is Ø, the list element must not be double precision. If the specification is A or R, the list element must not be logical or double precision. If any of these conditions is not satisfied, a message is printed on the OUTPUT file:

*TRACE* I/O LIST ELEMENT, ADDRESS = octalB AT SEQUENCE NUMBER n*

    type LIST ELEMENT USES k FORMAT SPEC

(where type is one of: REAL, DOUBLE, LOGICAL, INTEGER, COMPLEX, -- and k is one of I, L, Ø, A, R, E, F, G, or D) and control is returned to the format processor which converts according to the given format specification even though an error has occurred. All messages are printed before any line that was requested to be printed on the OUTPUT file. The cross reference map containing octal addresses can be used to locate the I/O list elements.

NO TRACE FORMATIO stops further checking of formatted I/O statements.
3.4 TRACE SUBSCRIPTS

NO TRACE SUBSCRIPTS

TRACE SUBSCRIPTS causes subscript checking to be done on all subsequently
declared arrays. When TRACE SUBSCRIPTS is used, every array subscript reference
is checked to be sure it is between 1 and the total dimension specification of
the array. The subscript expression is compared with the product of the dimen-
sion. This means that an individual subscript may be larger than its corres-
ponding dimension, and yet be undetected as an error. However, overwriting of
areas outside the array is always detected. If any reference is illegal, a
message is printed on the output file:

*TRACE* SUBSCRIPT FOR ARRAY name AT SEQUENCE NUMBER n*

VALUE = k OUTSIDE RANGE 1 TO m

and control is returned to the user's program where the subscript is used
without being changed. In order to minimize computing time and unnecessary
extra output, for one-statement DO or FOR loops the value of the subscript is
checked using the initial and final value of the DO and FOR parameters. However,
this may cause a false message if the final value of the index is not attainable.
An example that would generate a false message is:

DIMENSION A(10,10)

DO 10 I=1,11,3
10 A(I,I)=6.4

This is an example of a 'one statement DO'

NO TRACE SUBSCRIPTS stops further checking of all subscripts. Note that the
statement "TRACE list" and "NO TRACE list" may be used to begin or end tracing
of individual arrays (see section 4.2).

3.5 TRACE TRANSFERS

NO TRACE TRANSFERS

Whenever an assigned GO TO statement is executed and TRACE TRANSFERS is in
effect, the assign variable is checked to be sure it is one of the statements in
the GO TO number list, as in:

ASSIGN 5 to I

GO TO I,(1,2,5,10) or GO TO I

If the value of the variable is not one of the statements in the list or if
there is no list, a message is printed on the OUTPUT file:

*TRACE* ASSIGNED VARIABLE, VALUE = octalB AT SEQUENCE NUMBER n*

IS NOT EQUIVALENT TO ANY ITEM IN THE STATEMENT NUMBER LIST

and the program makes the jump to whatever value the assign variable had.
The source listing which has the absolute octal address in the first column allows the actual Fortran statement number to be identified.

Also, when a computed GO TO is executed, the computed GO TO variable or expression is checked that it is between 1 and the number of statement numbers in the statement number list. If not, a message is printed on the OUTPUT file:

*TRACE* COMPUTED GO TO VALUE = k AT SEQUENCE NUMBER n*

IS NOT INSIDE RANGE 1 TO NUMBER OF STATEMENTS (FATAL)

after which the program is terminated.

NO TRACE TRANSFERS stops further checking of assigned and computed GO TO statements.

3.6 TRACE SUBPROGRAM FLOW

NO TRACE SUBPROGRAM FLOW

Each time a user subprogram or library function is called, the message:

*TRACE* CALL OF SUBPROGRAM NAME AT SEQUENCE NUMBER n*

is printed on the OUTPUT file. Note that a lot of output can be generated. NO TRACE SUBPROGRAM FLOW deactivates this trace for later calls.

4.0 TRACE list [E.g: TRACE ARRAY (-1.,100.),10,1001,X,I(0,2)]

Each element of the list is separated by a comma. The elements of the list may be

4.1 Simple Variable Name.

Whenever the value of the named variable could be changed (i.e., via arithmetic replacement, loop index variable change, or storage as a READ-DECODE list element) or whenever the variable appears in the argument list of a function reference or subroutine call, a message is printed on the OUTPUT file:

*TRACE* STORE INTO type VARIABLE name AT SEQUENCE NUMBER n*

VALUE = k octalB

where type is one of REAL, INTEGER, DOUBLE, COMPLEX, or LOGICAL and octalB is the octal value, after which the change of value (if any) is made as specified. The assignment of a statement number to a variable in an ASSIGN statement is not traced.

4.2 Array Name.

Each array subscript reference for the specified arrays is checked to insure that it is between 1 and the total dimension specification of the array. If any reference is illegal, a message is printed on the OUTPUT file:

*TRACE* SUBSCRIPT FOR ARRAY name AT SEQUENCE NUMBER n*

VALUE = k OUTSIDE RANGE 1 to m

and control is returned to the user’s program where the subscript is used without being changed. In order to minimize computing time and unnecessary extra output, for one-statement DO or FOR loops the value of the subscripts is checked using the initial and final value of the DO or FOR parameters. However, this may cause a false message is the final value of the index is not attainable. An example that would generate a false message is:
DIMENSION X(10)
TRACE X
DO 1 I=1,11,3
1 X(I)=1.5+FLOAT(I)

This is especially useful when an overall TRACE SUBSCRIPTS causes the program
to execute too slowly, and only a few arrays are really suspect.

4.3 Subroutine Names or Entry Names.
After each return from the named subroutines or entries, a message is printed
on the OUTPUT file:

*TRACE* CALL OF SUBPROGRAM name FROM SEQUENCE NO. n*

4.4 Function Names or Function Entry Names.
Whenever the named user or library functions or function entries are referenced,
the messages are printed on the OUTPUT file:

*TRACE* CALL OF SUBPROGRAM name FROM SEQUENCE NO. n*

and

*TRACE* STORE INTO type VARIABLE name AT SEQUENCE NUMBER n*

VALUE = k octalB

where k is the decimal value and octalB is the octal value of the function or
function entry after the return from the function is made.

4.5 Simple Variable Name Followed by (Lower Bound, Upper Bound).
"Lower Bound" and "Upper Bound" must be constants or simple variables of the same
type as the specified variable -- which may be double precision, real, complex
or integer. Every time the value of the variable is changed or whenever the
variable appears in the argument list of a function reference or subroutine
call, it is checked as to whether it is between the lower bound and the upper
bound. For complex numbers, the real and imaginary parts of the variable are
separately checked against the real and imaginary parts of the lower and upper
bounds. For double precision numbers, only the upper precision part is checked
against the upper precision part of the bound. If the bounds are exceeded, a
message is printed on the OUTPUT file:

*TRACE* BOUNDS ON type VARIABLE name AT SEQUENCE NUMBER n*

VALUE = k

and the change of value is made as specified.

4.6 Array Names Each Followed by (Lower Bound, Upper Bound).
This is the same as Section 4.5, except that every element of the array is
checked and the subscript is also checked.

4.7 Function Names or Function Entry Names Each Followed by (Lower Bound, Upper Bound).
This is the same as 4.4, except that the check is made each time the named
function or entry name is referenced. However, the following message is always
printed even if bounds are not exceeded:

*TRACE* CALL OF SUBPROGRAM name FROM SEQUENCE NO. n*
4.8 Statement Numbers.

Every time a branch is made to the statement number or it is executed in normal sequence, a message is printed on the OUTPUT file:

*TRACE* TRANSFER TO STATEMENT NUMBER m FROM SEQUENCE NO. n*

and control returns to the statement number.

4.9 Format Numbers.

Whenever a formatted READ, WRITE, PRINT or PUNCH statement or an ENCODE or DECODE statement is executed with the specified format numbers, a check is made that each format specification implies a correct type for the corresponding list element (see Section 3.3). If not, the following message is printed on the OUTPUT file:

*TRACE* I/O LIST ELEMENT, ADDRESS = octalB AT SEQUENCE NUMBER n*

  type LIST ELEMENT USES k FORMAT SPEC

(where type is one of: REAL, DOUBLE (PRECISION), LOGICAL, INTEGER, COMPLEX and k is one of I, L, $\phi$, A, R, E, F, G, or D) and control is returned to the format processor which converts according to the given format specification. All messages are printed before any line that was requested to be printed on the OUTPUT file. The cross reference map containing octal addresses can be used to locate the I/O list elements.

Note: If any statement or format numbers are traced, at the end of execution the list of program and subprogram names followed by statement and format numbers with counts is printed. However these counts will be zero unless TRACE STATEMENT NUMBERS is used. (See Section 2.1).

Note: If MNF(T) is used, an array name appears in a TRACE list, then the message "array REPEATED IN TRACE" may be printed. This is because both (T) and the TRACE turn on TRACE SUBSCIRPTS for the array. No error is implied by the message.

5.1 NO TRACE List

Each element of the list is separated by a comma. List elements are variable, array, or subprogram names, statement numbers or format numbers. They should correspond to names or numbers declared in a previous TRACE statement unless they are turning off tracing of individual items within a global trace. If not, an error message will be given and those names or numbers are ignored. The type of tracing specified by the TRACE statement is terminated by the NO TRACE statement.

6.0 TRACE

NO TRACE

TRACE turns on the same tracing that the T option of the MNF card turns on, namely TRACE DOLOOPING, TRACE FORMATIO, TRACE STATEMENT NUMBERS, TRACE SUBPROGRAM CALLS, TRACE SUBSCIRPTS, and TRACE TRANSFERS. These remain in effect until the end of the whole program and unless turned off individually or in toto by a NO TRACE statement. These two statements provide a simple way of limiting tracing to only a given area of the program (e.g. to one subroutine).
7.0 Example of TRACE list

Section number [4.6] [4.0] [4.7] [4.4] [4.5]

TRACE A(-10.0,10.0), 10, 20, SQRT, X(0.0,2.5).
REAL A(5,6)

10 FORMAT(1X,I6)

20 X=SQRT(Y)
!OTRACE A,X,20 [leaves tracing on for SQRT and 10 only]

8.0 TRACSET - To Limit TRACE Output

TRACSET provides a means of limiting the number of messages produced under
MNF(T) - TRACE mode. The amount of output produced can be very large, e.g.,

CALL TRACSET(1,500,20,20)
would execute the loop 500 times, printing error messages for only the last 20
executions, and then stop the job.
Note that only TRACE subscript errors are controlled by the particular call. Many calls to TRACSET may be made during execution, and it will be seen that this routine offers complete control over MNF TRACE output, except that TRACE DO LOOPING, TRACE TRANSFERS, and TRACE FORMATIO messages are not affected by TRACSET, since these kinds of errors are usually too serious to be suppressed.
The MNF Format free I/O statements, and other extensions to I/O

<table>
<thead>
<tr>
<th>Section</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction</td>
<td></td>
</tr>
<tr>
<td>2. FORTRAN file handling</td>
<td></td>
</tr>
<tr>
<td>3. Normal format processing</td>
<td></td>
</tr>
<tr>
<td>4. Read statement status and error checking (END and ERR)</td>
<td></td>
</tr>
<tr>
<td>5. Free Format input</td>
<td></td>
</tr>
<tr>
<td>6. Free Format output</td>
<td></td>
</tr>
<tr>
<td>7. Example program</td>
<td></td>
</tr>
<tr>
<td>8. Other FORTRAN I/O statements</td>
<td></td>
</tr>
<tr>
<td>9. Error messages</td>
<td></td>
</tr>
</tbody>
</table>

**NOTE:** In this Appendix the Hollerith delimiter character ≠ is an 8-4 punch on cards.
1. Introduction

This appendix contains a basic introduction to the input and output facilities in the MNF FORTRAN compiler. Emphasis is placed on the READ statement facility to jump to labels on special conditions, and on the facility to use normal READ and WRITE statements without specifying an associated format number.

The .END. and .ERR. parameters on the READ statement are similar to those available in IBM/360 FORTRAN. They enable a user to recover directly from incorrect data or an end of data condition (see Section 4).

Of more interest to the general user will be the descriptions of the free format input and output statements (see Sections 5, 6 and 7). For many elementary applications and especially for beginners to FORTRAN, the power of the FORMAT statement is not needed. When using MNF, a WRITE or PRINT statement without a format specification may be used to output the value of a variable in a standard layout which depends on the type of value to be printed. An input string (i.e. data) for free format READ statements consists of values separated by spaces or commas. These values will be assigned to the variables in the input list. These free format statements are especially useful for teletype users.

The extensions described above are provided by new library routines (called INPUT$ and OUTPUT$) written at the University of Minnesota. MNF programs do not use the CDC routines INPUTC, OUTPUTC, KODER and KRAKER. This change is reflected in the loader map printed at the bottom of the compilation output.

Please note that most of the facilities described in this document are non-standard FORTRAN and will only be available with MNF and not with
Acknowledgement is made to Imperial College, London for permission to reproduce the material in this Appendix.

2. File Handling.
2.1 PROGRAM Card

FORTRAN programs refer to files by means of unit numbers. The identification of a unit number with a filename is made on the PROGRAM card, which should normally be the first card of a FORTRAN main program.

Every FORTRAN job has some special files associated with it. The file named INPUT is the standard input stream (card reader) and the file called OUTPUT is the standard destination for output (printer). In addition, any information written on the file PUNCH will be punched on BCD cards at the end of the job.

Thus, if the statements used for Input and Output are the simple forms:

```
READ fmt, L
PRINT fmt, L
```

where fmt is a format number and L is the I/O list, a suitable program card is:

```
PROGRAM XX (INPUT,OUTPUT)
```

If the usual conventions for unit numbers are followed (that is input on unit 5 and output on unit 6) in statements like:

```
WRITE (6,fmt) L
READ (5,fmt) L
```

unit numbers 5 and 6 must be made to reference the files INPUT and OUTPUT respectively by using the following program card:
PROGRAM TEST (INPUT, OUTPUT, TAPE5=INPUT, TAPE6=OUTPUT)

This is the file environment which MNF sets up if the PROGRAM card is omitted.

As a further example, consider the case where some coded data on a magnetic tape is to be processed.

PROGRAM TEST(FRED, OUTPUT, TAPE5=FRED, TAPE6=OUTPUT)

REWIND 5  rewinds file FRED
READ (5, 1000) A  read from file FRED

WRITE (6, 2000) B  write on file OUTPUT

END

6/7/8/9

The program card is needed here because the input file instead of being on cards is actually on the file (tape) called FRED.

Another way of writing this program is:

PROGRAM TEST(TAPE5, OUTPUT, TAPE6=OUTPUT)

REWIND 5
READ (5, 1000) A  [REQUEST(TAPE5) would be in the control cards]

WRITE (6, 2000) B

END

6/7/8/9
2.2 File manipulation

2.2.1 REWIND u

This causes the file referenced by unit u to be repositioned at the start of the file. u may be an integer from 1 to 99 or a simple integer variable having a value in this range. It may also be a list of unit numbers separated by commas. (This is a non-standard facility of MNF).

2.2.2 BACKSPACE u

Unit u is backspaced one record. If the file is already positioned at its beginning, the statement does nothing. As in REWIND, u may be a single specification or a list.

2.2.3 ENDFILE u

This causes an end of file to be written on the file referenced by unit u. As in REWIND, u may be a list.

3. Normal Formatted Input/Output Processing

3.1 Input/Output Statements

The various forms of formatted input/output statement are:

- PRINT fmt, L or PRINT fmt
- READ fmt, L or READ fmt
- PUNCH fmt, L or PUNCH fmt
- WRITE (u,fmt) L or WRITE (u,fmt)
- READ (u,fmt) L or READ (u,fmt)

fmt is a format number (or the name of a variable or array element which contains the start of the format information).

u is the unit number (see Section 2.1)

L is a list of items separated by commas.

The items may be:

(a) Simple variable names
(b) Subscripted array names
(c) Implied do lists
(d) Unsubscripted array names (in which case the whole array is implied)
Each individual item in a list L will require a single conversion specification in the corresponding format fmt, except in the case of COMPLEX variables which require two real specifications.

In addition, in output lists the following are also permitted:

(e) Any FORTRAN expression (except those which contain references to functions which themselves do input or output)

Free format output statements may also contain:

(f) Character strings enclosed in quotes (ie ≠ ... ≠) and Hollerith constants.

(g) A slash properly delimited by commas, i.e. the following is allowed:
   X=1.0 $ Y=2.0
   PRINT,/,≠X PLUS Y IS ≠, X+Y
   This produces the output starting on a new line:
   X PLUS Y IS  3.000000

3.2 Format Specifications

A full description of the various possible types of FORTRAN input/output elements will not be given here. A full description may be found in this Reference Manual.

A table of format specifications which may be used in FORMAT statements with the MNF compiler follows:

Symbols

n = repeat count: unsigned integer constant.
w = field width: unsigned integer constant.
d = number of digits after decimal point:
   unsigned integer constant.
x = any character in the FORTRAN character set.
y = any character except * 
z = any character except ≠
r = signed or unsigned integer constant.
j = column position: unsigned integer constant.
k = Hollerith character count: unsigned integer constant.
Legal format specifications

<table>
<thead>
<tr>
<th>Type</th>
<th>Form</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>nAw or Aw</td>
<td>Alphanumeric conversion (left justified-blank filled)</td>
</tr>
<tr>
<td>D</td>
<td>nDw.d or Dw.d</td>
<td>Double precision conversion</td>
</tr>
<tr>
<td>E</td>
<td>nEw.d or Ew.d</td>
<td>Real conversion</td>
</tr>
<tr>
<td>F</td>
<td>nFw.d or Fw.d</td>
<td>Real conversion</td>
</tr>
<tr>
<td>G</td>
<td>nGw.d or Gw.d</td>
<td>Real conversion</td>
</tr>
<tr>
<td>H</td>
<td>kHxx...xx</td>
<td>Hollerith (number of x's must equal k)</td>
</tr>
<tr>
<td>I</td>
<td>nIw or Iw</td>
<td>Integer conversion</td>
</tr>
<tr>
<td>L</td>
<td>nLw or Lw</td>
<td>Logical conversion</td>
</tr>
<tr>
<td>O</td>
<td>nOw or Ow</td>
<td>Octal conversion</td>
</tr>
<tr>
<td>P</td>
<td>rP or P</td>
<td>Scale factor</td>
</tr>
<tr>
<td>R</td>
<td>nRw or Rw</td>
<td>Alphanumeric conversion (right justified-zero filled)</td>
</tr>
<tr>
<td>T</td>
<td>Tj</td>
<td>Tabulation</td>
</tr>
<tr>
<td>X</td>
<td>nX or X</td>
<td>Space</td>
</tr>
<tr>
<td>*</td>
<td><em>yy...yy</em></td>
<td>Hollerith</td>
</tr>
<tr>
<td>#</td>
<td>#zz...zz#</td>
<td>Hollerith [Note: #DON/#T/ will print DONT]</td>
</tr>
<tr>
<td>/</td>
<td>/</td>
<td>New record</td>
</tr>
</tbody>
</table>

3.3. Execution Time Errors

3.3.1 Output data errors

In certain special circumstances, unexpected characters will turn up in numeric output fields.

* means that the field width specified in the format was not big enough to accommodate the value to be printed (e.g. J=2222 under I3).

I means that a real, complex or double precision variable has the special hardware value "indefinite" (commonly caused by an uninitialised variable).

R means that a real, complex, or double precision variable had the special value "infinite" (commonly caused by dividing by zero), and X means that an attempt was made to print an integer with a value more than 2**48-1 (usually printing a real number on I format).
3.3.2 List/Format disagreements

When processing an input/output list with its associated format, it may happen that the type of the format specification does not correspond with the data to be transferred. For example:

```
12* READ (5,1000) I,J,K
13* 1000 FORMAT (I4,F6.2,I4)
```

In this case, the following error message will be printed and processing will continue if TRACE FORMATIO is switched on.

*TRACE I/O LIST ELEMENT,ADDR 004761B AT SEQUENCE NUMBER 12* INTEGER LIST ELEMENT USES F FORMAT SPEC

The most general control possible is given by specifying the T option on the MNF control card, which turns on tracing for all input/output statements. The I/O tracing action which is globally initiated by the control card T option may be switched on or off more selectively by using the following statements:

```
C$ TRACE FORMATIO
C$ NOTRACE FORMATIO
C$ TRACE tracelist
C$ NOTRACE tracelist
```

The first two of the above switch on/off the tracing on all textually succeeding input and output statements. An individual format number may be one of the elements of the tracelist in the general trace statement, allowing yet closer control. For a full explanation refer to Appendix D.
4. Read Statement Status and Error Checking

4.1 Detecting End of File status

There are many occasions when a FORTRAN program must read an indeterminate amount of data which does not contain any signal to say it is ending. Thus, the programmer must have control over what should happen when the physical end of data is reached. The normal way of detecting this situation is to test for the end of file condition after each read statement using the EOF Function. For example, this program counts how many cards are on the input file:

```
INTEGER CARD (8)
NUMBER=0
10 READ (5,1000) (CARD(I), I=1,8)
1000 FORMAT (8A10)
   IF (EOF(5).NE.0.0) GOTO 20
   NUMBER=NUMBER+1
   GO TO 10
20 PRINT 2000, NUMBER
STOP
2000 FORMAT (*THERE ARE*,I4,*CARDS*)
END
```

The EOF function returns a zero value unless an end of file was detected during the preceding read on the specified unit number. The same procedure may be accomplished, when using MNF, by an extension to the READ statement. The example above may be rewritten:

```
INTEGER CARD (8)
NUMBER=0
10 READ (.END.=20,5, 1000) (CARD(I),I=1,8)
1000 FORMAT (8A10)
   NUMBER=NUMBER+1
   GO TO 10
20 PRINT 2000, NUMBER
STOP
2000 FORMAT (* THERE ARE *,I4,*CARDS*)
END
```
Control will be transferred to the statement number specified by the .END. = when the end of file condition is detected. Note that FORTRAN programs do not distinguish between end of file and end of record. Thus, a 7/8/9 card on the input file will be detected by an end of file test. A further read will read the next card past the 7/8/9 card. If an end of file is detected during the processing of an input list, the remaining elements of the list will be filled as if blank fields had been read.

4.2 Recovery from bad data

An error which occurs during data input may be recovered if the programmer specifies some action to be taken. This can be done by using another extension to the READ statement:

```
READ(.ERR.=sn,u,fmt) L
```

The .ERR. parameter indicates the label to jump to if there is a data error during input. The user is then free to take any appropriate action.

The .END. and .ERR. parameters may both appear on the same READ statement. They may be placed in either order but both should appear before the unit number.

The following program demonstrates how a program could be constructed to count the number of cards in a file which have a valid integer in the first 10 columns:

Note: The END and ERR feature of MNF is very similar to that of IBM/360 Fortran, but in IBM there are no periods round the END or ERR.
PROGRAMME (INPUT, OUTPUT, TAPE5 = INPUT)
NUM = 0 $ NUMERR = 0
10 READ (.ERR. = 220, .END. = 330, 5, 1000) L
1000 FORMAT (I10)
NUM = NUM + 1
GO TO 10
220 NUMERR = NUMERR + 1
GO TO 10
330 PRINT 2000, NUM, NUMERR
STOP
2000 FORMAT (*$NUM=$, I4$ NUMBER OF ERROR CARDS*, I4)
END

5. Free-Format Input

5.1 Statement Form

An MNF program may contain READ statements which do not have corresponding FORMAT statements. Data may be read where each item is separated from the next either by a comma or by one or more spaces. Thus the statement:

READ, L

reads free-format data from the file INPUT, and similarly:

READ(u,), L

reads free format data from unit number u. (Note that the comma is essential here, as READ(u) L is a binary read statement and means something else).

The other free-format input statements are:

READ (.END. = sn1, u,) L
READ (.ERR. = sn2, u,) L
READ (.END. = sn1, .ERR. = sn2, u,) L
READ (.ERR. = sn2, .END. = sn1, u,) L

where sn1 and sn2 are statement numbers in the current program or subroutine.

Each READ statement starts a scan for data at the beginning of a new card or record. If the list L requires more data than appears on the first card or record, more cards will be read until the list is satisfied.
The .END. parameter operates as described in section 4. If the end of file condition occurs part of the way through the list, the rest of the variables have values stored in them as if all blank fields had been read. The input field for a list element must contain a legal value of the same type as the list element. If this is not the case, a fatal error will occur unless the .ERR. parameter was specified in the READ statement. If it was, control will be transferred to sn2 on completion of the READ statement. Any illegal values will be stored as the octal constant 3777377377737773777B. This value will usually give an "infinite result" fatal error if used in arithmetic statements.

5.2 Layout of Input data

Values for any type of variable may be read in by the free-format routines. Integer, real, double precision and logical variables require single values from the input stream, whereas complex variables require two real values. Character strings may also be read into integer variables or arrays. Each data value must be separated from the next by a comma or at least one blank. Two consecutive commas indicate an all-blank field. Data values must not be split over two cards because the last data item on a card is considered terminated by the end of the card.

5.2.1 Real, Double precision and Complex values

If the next variable in the read statement list is a real or double precision variable, the next characters on the input stream are converted to a real or double precision value. If the field is all blank, the value -0 is stored. The input field should contain a number in F or E or D format, the only difference from normal processing being that trailing blanks are discarded rather than being taken as zeros. A complex
variable requires two consecutive real values, which are then stored as the real and imaginary parts of the variable.

For example, the program:

```fortran
COMPLEX CC
REAL X,Y :
READ,X,CC,Y :
```

operating on the data lines:

```
26.734,1.8E3 7.7E4
29.7E-36
```

has the same effect as the statements:

```
X=26.734
CC=(1.8E3,7.7E4)
Y=29.7E-36
```

5.2.2 Logical values

Reading values into logical variables with the free-format READ statement requires the data to follow the same rules as for L format input. If the input field is blank, it is read as the value FALSE.

For example, the program:

```fortran
LOGICAL FRIEND,SWITCH(3)
:
READ(5,)SWITCH,FRIEND
```

operating on the data card:

```
TRUE,,F
```

has the same effect as the statements:

```
SWITCH(1)=.TRUE.
SWITCH(2)=.FALSE.
SWITCH(3)=.FALSE. (or .F.)
FRIEND=.FALSE.
```
5.2.3 Integer values and strings

Integer values will be input in the same way as formatted input using I format, except that trailing blanks terminate the number instead of being treated as zeros.

For example, the program:

```
INTEGER X
READ,I,X,M
```

operating on the data line:

```
91,23 14
```

has the same effect as the statements:

```
I=91
X=23
M=14
```

If either of the characters * or # appears as the first non-blank character of an input field for an integer variable, the field will be taken to be an alphanumeric string. The field must be ended by a matching * or # followed by a comma, a blank or the end of a card. If the list element is an unsubscripted integer array, the string is allowed to overflow from the first to succeeding elements of the array. Characters are packed 10 to a word, the characters in the last word being packed up against the left-hand end of the word with zeros filling the rest of the word. Unused elements of the array are also set to zero. The character * may be represented in a field bracketed by asterisks by **. The character # in a field bracketed by quotes may be represented by # #. An input data field of #*### is stored as **#

Fatal errors occur if any of the conditions above are violated.

Note: The character # has been replaced by ' (apostrophe, or quote, or prime) on many CDC computers.
For example, the program:

```
INTEGER STRING(4)
:
READ (5,)I, STRING
:
```

acting on the data record:

```
*HELLO*, /*THIS IS A STRING*/
```

has the same effect as:

```
I=5LHELLO
STRING(1)=10LTHIS IS A
STRING(2)=6LSTRING
STRING(3)=STRING(4)=0
```

6. Free-Format Output

6.1 Statement Form

```
PRINT, L    Free format output on file OUTPUT
PUNCH, L    Free format output on file PUNCH
WRITE(u,) L Free format output on logical unit u.
```

Each of these statements starts the output of the list at the
second character position of a line and continues to more than on line
if necessary. As well as the normal list elements, the list L may
contain character strings and slashes. Character strings enclosed in
quotes will be written out in groups of 10 characters. If the next
group does not fit on a line, it is placed on the next line starting in
column two. A slash may be used to indicate that a new line is to be
started. A string or a slash is a normal list element in that it must
be separated from other list elements by commas. For example, the
statement:
PRINT, \*FIRST LINE\*, /, /, \*WE MISSED ONE LINE\*

Produces the output:

FIRST LINE
WE MISSED ONE LINE

Note that the quotes are not written out around the strings, so strings cannot be read back using free-format READ statements.

6.2. Layout of Output

Individual list items are written out in a format which depends on the type of the variable as shown below:

<table>
<thead>
<tr>
<th>Type</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>I15</td>
</tr>
<tr>
<td>Real</td>
<td>G15.7</td>
</tr>
<tr>
<td>Double Precision</td>
<td>D25.12</td>
</tr>
<tr>
<td>Complex</td>
<td>2G15.7</td>
</tr>
<tr>
<td>Logical</td>
<td>L10</td>
</tr>
<tr>
<td>Alphanumeric</td>
<td>A10</td>
</tr>
</tbody>
</table>

If an integer variable or array element contains a number > \(2^{**48}-1\), it is assumed to contain character information and will be printed in A10 format. For example the statements:

```
DOUBLE DX
COMPLEX CX
DX=55.0123456789012345678901D-12
CX=(47.6,67.4)
R=200E12
I=25
:
PRINT, /I/=\$, I, /R/=\$, R, /, /CX/=\$, CX, /, /DX/=\$, DX
```

produce the output

```
I= 25
CX= 47.6000000 67.4000000
DX= .550123456789 D-10
```

6.3. Further controls

Three parameters involved in the layout of free-format output may be altered or reset by the user. They are held in the first three locations
of a labelled common block called /FREEFOR/. The parameters are completely dynamic and may be changed repeatedly during execution of the program. For demonstration purposes, the following declaration is assumed to be present in the source program:

COMMON/FREEFOR/IFREEL,IFREEW,IFREED

The number of characters per output record (line) may be changed by assigning another value to IFREEL.

e.g. IFREEL=80

This sets the width of the output to 80 characters and this would be suitable if output was to be to punched cards. The normal default value is 72. The upper limit is 150 characters and the lower limit is that just large enough to accommodate the largest variable to be output.

The fixed formats for the different types of variables may be controlled by two parameters (W and D). W is the field width controller and D is the number of places to appear after the decimal point.

(W and D reside in locations 2 and 3 of the FREEFOR common block respectively.)

For example, after the assignments:

IFREEW=9
IFREED=4

real numbers will be printed in G18.4 format.

Note that, when characters are being output if W >10, A10 format will be used and the rest of the field filled with blanks.

The initial settings of the /FREEFOR/ block are:

IFREEL=72  IFREEW=10  IFREED=5
The formats of section 6.2 may be given now more generally as:

- Integer: I(1.5*W)
- Real: G(1.5*W).(1.5*D)
- Double: D(2.5*W).(2.5*D)
- Complex: 2G(1.5*W).(1.5*W)
- Logical: L(W)
- Alphanumeric: A(W)

It can be seen that the default values of W and D are W=10 and D=5

7. Example Program

```
PROGRAM P16 (INPUT, OUTPUT, TAPE5=INPUT, TAPE6=OUTPUT)
DIMENSION NAME(8)
C     C THIS PROGRAM CALCULATES AND PRINTS AREAS OF ROOMS
C
WRITE(6,1000)
1000   FORMAT(*1*,*THIS IS A TITLE ON A NEW PAGE*)
C     C CARD READING LOOP
200    READ(.END.=600,.ERR.=400,5,) NAME,X,Y
PRINT(/,AREA OF NAME IS X*Y,SQ.FT.***GOTO 200
C     C PRINT MESSAGE FOR CARD IN ERROR
400    PRINT,/*** CARD IS IN ERROR ***
GOTO 200
C     C EXIT TO HERE WHEN END OF FILE SENSED ON INPUT
600    PRINT(/,END OF OUTPUT***
STOP
END
```

Output starting on a new page:

```
THIS IS A TITLE ON A NEW PAGE

AREA OF KITCHEN IS 103.2000000 SQ.FT.
AREA OF LIVING ROOM IS 193.7250000 SQ.FT.
*** CARD IS IN ERROR ***
AREA OF BATHROOM IS 74.4970000 SQ.FT.
END OF OUTPUT
```
8. Other FORTRAN I/O Statements

8.1 DECODE Statement

The DECODE statement provides a way of reading character data from core storage rather than from an external device.

The normal DECODE statement is:

```
DECODE (c,fmt,v)L
```

where

- `c` is the number of characters
- `fmt` is the format
- `v` is the array with data in it.

The error parameter may be used in this statement as follows:

```
DECODE(.ERR.=sn,c,fmt,v)L
```

If an attempt is made to process a character which is illegal under the current format conversion, a jump will be taken to `sn` when the whole of list `L` has been processed.

ENCODE and BUFFER statements are also allowed, and are described in the **Reference Manual**.

9. Error Messages

The following is a list of error messages which may be generated by the Input/Output processing routines INPUT$ and OUTPUT$. Only error numbers 73 and 80 concern the user of free-format input/output statements.

In cases where it is relevant, the format may be printed out with an arrow pointing to where the error occurred, or the input record may be printed with an arrow pointing to an illegal character.
<table>
<thead>
<tr>
<th>Error Number</th>
<th>Message and Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>ERROR DATA INPUT *** UNDEFINED FILE NAME: A READ was attempted on a file not defined on the PROGRAM card.</td>
</tr>
<tr>
<td>65</td>
<td>ERROR DATA INPUT *** READ PAST END OF FILE: An attempt was made to read past an end of record (7/8/9 card) or an end of file (6/7/8/9) - see section 4.</td>
</tr>
<tr>
<td>66</td>
<td>DECODE ERROR ***** CHARACTER COUNT EXCEEDS 150: Bad decode statement.</td>
</tr>
<tr>
<td>68</td>
<td>ILLEGAL FUNCTIONAL LETTER: An illegal conversion letter occurred in an output format.</td>
</tr>
<tr>
<td>69</td>
<td>IMPROPER PARENTHESES NESTING: Parentheses do not match in an output format or are nested deeper than two levels i.e. deeper than FORMAT ((( ))) .</td>
</tr>
<tr>
<td>70</td>
<td>RECORD SPECIFICATION EXCEEDS 150 CHARACTERS: Tried to output more than 150 characters in one line.</td>
</tr>
<tr>
<td>71</td>
<td>SPECIFIED FIELD WIDTH OR REPEAT COUNT ZERO OR NEGATIVE: A field width is negative or a repeat count is less than 1 in an output format.</td>
</tr>
<tr>
<td>73</td>
<td>ATTEMPT TO READ OR WRITE DATA ITEM WITH FORMAT CONTAINING NO DATA CONVERSION FIELDS: Attempt to write value of a variable to a format which contains no data conversion fields.</td>
</tr>
<tr>
<td></td>
<td>or RECURSIVE CALL TO INIT80: output list item is in TRACE statement.</td>
</tr>
<tr>
<td></td>
<td>or ERROR FREE FORMAT OUTPUT *** COMMON BLOCK/FREEFOR/ PARAMETERS ZERO OR NEGATIVE: Common block parameters are illegal - see section 6.</td>
</tr>
<tr>
<td></td>
<td>or ERROR FREE FORMAT OUTPUT *** COMMON BLOCK/FREEFOR/ PARAM 1 EXCEEDS 150 OR PARAM 1 TOO SMALL FOR FIELD: First word of common block set to illegal value - see section 6.</td>
</tr>
<tr>
<td>74</td>
<td>ILLEGAL FUNCTIONAL LETTER: A illegal conversion letter occurred in an input format.</td>
</tr>
<tr>
<td>75</td>
<td>IMPROPER PARENTHESES NESTING: Parentheses do not match in an input format or are nested deeper than two levels i.e. deeper than FORMAT ((( ))) .</td>
</tr>
<tr>
<td>Error Number</td>
<td>Message and Cause</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>76</td>
<td>SPECIFIED FIELD WIDTH OR REPEAT COUNT ZERO OR NEGATIVE: A field width is negative or a repeat count is less than 1 in an input format.</td>
</tr>
<tr>
<td>77</td>
<td>RECORD SPECIFICATION EXCEEDS 150 CHARACTERS: Tried to input a record more than 150 characters long.</td>
</tr>
<tr>
<td>78</td>
<td>ERROR DATA INPUT *** ILLEGAL DATA IN FIELD: Input data characters do not match format. Usually caused by mispunched characters.</td>
</tr>
<tr>
<td>79</td>
<td>ERROR DATA INPUT *** DATA OVERFLOW: Integer input item is $2 ** 48 - 1$ or exponent of real number is too large. Usually caused by not keeping numbers to the right of an input field. Trailing blanks are treated as zeros.</td>
</tr>
<tr>
<td>80</td>
<td>ATTEMPT TO READ OR WRITE DATA ITEM WITH FORMAT CONTAINING NO DATA CONVERSION FIELDS: Attempt to read a value of a variable with a format which does not contain any data conversion fields:</td>
</tr>
<tr>
<td>81</td>
<td>or RECURSIVE CALL TO INIT81: input list item is in TRACE statement.</td>
</tr>
<tr>
<td></td>
<td>or ERROR FREE FORMAT ALPHANUMERIC INPUT *** ILLEGAL OR MISSING DELIMITER: Alphanumeric input string too long or terminated illegally - see section 5.</td>
</tr>
<tr>
<td>82</td>
<td>or ERROR FREE FORMAT ALPHANUMERIC INPUT *** ARRAY OR VARIABLE NOT TYPE INTEGER: Alphanumeric data may only be read into integer variables.</td>
</tr>
<tr>
<td>83</td>
<td>ERROR DATA OUTPUT *** UNDEFINED FILE NAME: A WRITE was attempted on a file not defined on the PROGRAM card.</td>
</tr>
<tr>
<td>84</td>
<td>ERROR DATA OUTPUT *** OUTPUT FILE LINE LIMIT EXCEEDED: tried to print too many lines of output.</td>
</tr>
<tr>
<td>85</td>
<td>ENCODE ERROR *** CHARACTER COUNT EXCEEDS 150: bad encode statement.</td>
</tr>
<tr>
<td>88</td>
<td>ERROR DATA INPUT *** ILLEGAL READ AFTER WRITE: An attempt was made to read a file after writing on it, without an intervening REWIND or BACKSPACE.</td>
</tr>
</tbody>
</table>
1. MNF LISTABLE OUTPUT

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 SOURCE LISTING</td>
<td>2</td>
</tr>
<tr>
<td>1.2 CROSS REFERENCE MAP DISCREPANCIES</td>
<td>3</td>
</tr>
<tr>
<td>STATEMENT NUMBERS</td>
<td>3</td>
</tr>
<tr>
<td>NAMES</td>
<td>4</td>
</tr>
<tr>
<td>1.3 COMPILATION DIAGNOSTICS</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>

(Please have an MNF listing on hand when reading this Appendix)
1.1 SOURCE LISTING

The first line of the listing gives the date this version of the
MNFX compiler was assembled, the computer installation, the version of
the operating system, and the psr (correction) level of the compiler.
The date and time the user's program was compiled appears at the end
of the first line.

The second line shows the MNFX control card which initiated this
compilation.

The remaining lines of the source listing can each be divided into
three fields, as follows...

1 - An octal address (e.g. -- 0001053) which is the absolute location
within the job field length where the machine instructions
generated for the current statement (if any) will reside at
execution. [An appendix shows the general layout of storage for
variables, arrays and machine instructions.]

2 - The MNFX-assigned line number of the first FORTRAN statement
followed by an asterisk (e.g. -- 1*). Comment and continuation
lines are not assigned sequence numbers, and multiple statements
count one per statement.

3 - The FORTRAN source card itself.

Diagnostic messages, if any, begin with the sequence number of the
statement to which they refer (e.g. -- LINE 9*), and are followed by
the relevant message. If the error is fatal to execution, the message
is padded out with asterisks (so it is easy to find) followed by an
ordinal error number, a consecutive count of the number of fatal
errors encountered in the program (e.g. -- ERROR NO. 000001).

NOTE -- Neither line numbers nor their absolute octal address are
reset at the start of each routine; they are cumulative over the whole
MNFX source program.
1.2 CROSS REFERENCE MAP is a dictionary of all names and statement numbers appearing in the current subprogram with their usage listed according to MNF line number. It is controlled by the R-option on the MNF control card, as follows...

**R = 0** (zero) - no cross reference map is generated.

1. CROSS REFERENCES are listed for statement numbers sorted in ascending order and FORTRAN names sorted alphabetically except for unused variables (nulls) having no execution references in the current routine. Common block names and lengths are also listed.

2. A cross reference map for variables and arrays sorted by octal address is listed. This is particularly useful in interpreting octal dumps arising from abnormal program termination.

3. Combination of R=1 and R=2. This is the default if no R-option appears on the control card.

5. Same as R=1 but includes unused variables (nulls) having no execution references in the current routine.

7. Combination of R=2 and R=5; this is the most complete map.

If the L-option is on, the default value is R=3; however, if L=0 is specified, then R=0 is forced. The cross reference map may be further manipulated using the MNF FORTRAN statements references and no references.

Deletion of the cross reference map saves paper and about 20 percent of compile time, but eliminates a valuable aid for debugging and program information. Also, some compilation space is saved (under R=0) since not all of the map information is generated.

**DISCREPANCIES**

If there are errors in the program, the cross reference map may not be completely correct. Also, in certain cases the map is not accurate because the reference map information is gathered from the (MACHINE CODE), rather than from the source statements. In particular, a given FORTRAN statement may have more references than actually appeared since the generated instructions may refer to some items more than once. Sometimes there are fewer references than those appearing within a statement when the compiler attempts to optimize code. However, when a variable appears in a statement, at least one cross reference is always generated for it.

A final inaccuracy is also due to optimization. In order to achieve maximum speed, storage of results is delayed as long as possible. Thus, the variable or array element appearing on the left-hand side of a replacement statement will often not be in the cross reference map until the next statement.
MNF LISTABLE OUTPUT
CROSS REFERENCE MAP

STATEMENT NUMBERS

LINES IN THE STATEMENT NUMBER CROSS REFERENCE MAP EACH HAVE FOUR FIELDS. THE FIRST IS THE NUMBER ITSELF SORTED IN ASCENDING NUMERICAL ORDER. THE NEXT IS A TWO-CHARACTER IDENTIFIER -- SN FOR AN EXECUTABLE STATEMENT, OR FN FOR A FORMAT STATEMENT. THE THIRD IS EITHER THE ABSOLUTE OCTAL ADDRESS OF THE STATEMENT (E.G. -- 000503B) OR THE DECIMAL CHARACTER LENGTH OF THE LONGEST UNIT RECORD WITHIN THE FORMAT (E.G. -- LN=11); THUS, FOR FORMAT (3F10.5/4X,I20), LN=30 WOULD APPEAR. THE LENGTH SHOULD BE 80 OR LESS FOR A FORMAT USING PUNCH CARDS, AND 132 OR LESS FOR LINE PRINTER OR MICROFICHE OUTPUT.


J JUMP - STATEMENT NUMBER WAS IN A GO TO OR IF STATEMENT
R READ - FORMAT WAS USED IN A READ OR DECODE STATEMENT
W WRITE - FORMAT WAS USED IN A WRITE, ENCODE, PRINT OR PUNCH STATEMENT
L LOCATED - DENOTES THE LOCATION (DEFINING LINE) OF THE STATEMENT NUMBER
D DO/FOR - STATEMENT NUMBER WAS IN A DO OR FOR STATEMENT
A ASSIGN - STATEMENT WAS IN AN ASSIGN OR ASSIGNED GO TO STATEMENT
T TRACE - STATEMENT NUMBER WAS IN THE LIST OF A TRACE STATEMENT

NAMES

LINES IN THE NAME CROSS REFERENCE MAP EACH HAVE SIX FIELDS. FIRST THE NAME APPEARS SORTED ALPHABETICALLY. THEN A LETTER DENOTING TYPE IS GIVEN (R FOR REAL, D FOR DOUBLE PRECISION, I FOR INTEGER, C FOR COMPLEX AND L FOR LOGICAL) IF THE NAME HAS AN ASSOCIATED TYPE. SUBROUTINE NAMES, HOWEVER, WILL HAVE FORTRAN IMPLICIT TYPE. THE NEXT FIELD INDICATES THE MAIN USE OF THE NAME, AS FOLLOWS...

VARIABLE SIMPLE VARIABLE NAME (INCLUDING THAT ON THE FUNCTION STATEMENT, IF WITHIN A FUNCTION SUBPROGRAM) WHICH IS NOT A FORMAL PARAMETER AND DOES NOT APPEAR IN AN EQUIVALENCE STATEMENT.
ARRAY ARRAY NAME WHICH IS NEITHER A FORMAL PARAMETER NOR IN AN EQUIVALENCE STATEMENT.
PARAMETER SIMPLE VARIABLE IN THE FORMAL PARAMETER LIST.
PAR ARRAY ARRAY WHICH IS A FORMAL PARAMETER.
EQUIVALENCE  SIMPLE VARIABLE APPEARING IN AN EQUIVALENCE STATEMENT
EQ ARRAY    ARRAY APPEARING IN AN EQUIVALENCE STATEMENT
NAMELIST   NAMELIST NAME
STFUNCTION ARITHMETIC OR LOGICAL STATEMENT FUNCTION.
ENTRY      ADDITIONAL ENTRY POINT NAME
EXTERNAL   NAME OF THE CURRENT ROUTINE (IF IT IS A PROGRAM OR
           SUBROUTINE) OR SUBROUTINES OR FUNCTION NAMES WHICH
           ARE NON-STANDARD.
STANDARD   KNOWN STANDARD FUNCTIONS OR SUBROUTINES.
SYSTEM     SYSTEM SUBROUTINES (I.E. -- THOSE NAMES INDIRECTLY
           NEEDED BY THE PROGRAM, SUCH AS I/O ROUTINES).
FILE       INPUT/OUTPUT FILE NAMES (EACH FOLLOWED BY THE
           CHARACTER *)

IF R=5 OR R=7 IS SPECIFIED, NAMES WITH NO EXECUTION REFERENCES ARE
LISTED WITH AN ASTERISK PRECEDING THE MAIN-USE FIELD.

THE FOURTH FIELD IS THE ABSOLUTE OCTAL LOCATION OF THE VARIABLE OR
ARRAY, OR THE OCTAL PARAMETER NUMBER OF A NAME APPEARING IN THE FORMAL
PARAMETER LIST. THE NEXT FIELD IS THE BLOCK NAME - A NAME OR NUMBER
FOR A LABELED COMMON BLOCK, OR / / FOR BLANK COMMON.

THE LAST FIELD IS A LIST OF MNF-ASSIGNED LINE NUMBERS DENOTING
WHERE THE NAME WAS REFERENCED. EACH IS FOLLOWED BY AN IDENTIFYING
CHARACTER (E.G. -- 28D) AND POSSIBLY FOLLOWED BY A / AND A DECIMAL
INTEGER (E.G. -- 28U/5) WHICH GIVES THE NUMBER OF TIMES THE NAME WAS
REFERRED IN THAT WAY IN THE GIVEN LINE. THE CHARACTERS IDENTIFYING
THE TYPE OF REFERENCE ARE, AS FOLLOWS...

U USED - THE NAME WAS USED IN THE STATEMENT.
S STORED - THE NAME WAS ON THE LEFT-HAND SIDE OF AN EQUAL SIGN,
           WAS IN AN INPUT LIST, OR WAS IN A DATA STATEMENT.
D DECLARED - THE NAME WAS IN A DECLARATIVE STATEMENT
I INDEX - THE VARIABLE WAS A DO OR FOR INDEX.
P PARAMETER - THE VARIABLE WAS A DO OR FOR INDEX PARAMETER.
A ASSIGNED - THE VARIABLE WAS AN ASSIGN VARIABLE IN AN ASSIGN OR
           ASSIGNED GO TO STATEMENT.
R REFERENCED - IF THE NAME WAS AN ARGUMENT IN A CALL OR FUNCTION
           REFERENCE.

THE BLOCK NAMES FOR LABELED AND BLANK COMMON, AND THEIR DECIMAL
LENGTHS ARE LISTED NEXT.

THE LAST PORTION OF THE MAP APPEARS ONLY IF R=2, R=3 (THE DEFAULT),
OR R=7 IS SPECIFIED; IT LISTS VARIABLE AND ARRAY NAMES SORTED BY OCTAL
ADDRESS. FIRST, THE ADDRESS IS PRINTED FOLLOWED BY EITHER B, DENOTING
AN ABSOLUTE ADDRESS, OR BY C, DENOTING AN ADDRESS RELATIVE TO THE
ORIGIN OF BLANK COMMON. THE NAME FOLLOWS THE ADDRESS. THE ADDRESSES
APPEAR SORTED HORIZONTALLY ACROSS THE PAGE.
1.3 COMPILATION DIAGNOSTICS

There are over 400 different compile-time diagnostics. They appear in the source listing, usually immediately following the statement which caused them. However, comment lines and format statements may appear between a statement and its error message. Messages for the declarative statements often appear together around the first executable statement.

Diagnoses come in five flavors (levels), as follows...

1. Comment usually inefficient programming practice.
2. Note non-standard (not ANSI) Fortran usage.
3. Caution possible programming error.
4. Warning usually indicates a serious error.
5. Fatal inhibits execution, unless the D-option is specified on the control card.

The levels of messages printed are controlled by the E-option on the control card, where E=N suppresses messages of level N and lower levels. E=0 means print all messages; E=5 means print none.

The default is E=2, so that notes and comments are not printed.

When the listing is inhibited (via L=0 [ZERO]) it may be advisable to specify E=4 also, since messages are still listed together with the three source lines which follow them.

Finally, fatal diagnostics are counted, and this count (if non-zero) appears in the job dayfile. In addition, these lines are padded to the right margin with asterisks, so as to be more obvious.
EACH PROGRAM UNIT (PROGRAM, SUBROUTINE OR FUNCTION) HAS THE FOLLOWING MEMORY STRUCTURE...

<table>
<thead>
<tr>
<th>COMMENTS</th>
<th>DESCRIPTION</th>
<th>PSEUDO-COMPASS LOCATION TAG</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILE BUFFERS DEFINED</td>
<td>FILE BUFFERS DEFINED IMMPLICITLY OR BY PROGRAM STATEMENT (MAIN PROGRAM ONLY)</td>
<td></td>
</tr>
<tr>
<td>ASSIGNED WHEN FIRST</td>
<td>Labeled COMMON BLOCKS WHICH WERE NOT PREVIOUSLY DECLARED</td>
<td></td>
</tr>
<tr>
<td>EXECUTABLE STATEMENT</td>
<td>INCLUDES ALL ARRAYS AND VARIABLES APPEARING IN DIMENSION, TYPE AND DATA STATEMENTS.</td>
<td></td>
</tr>
<tr>
<td>IS ENCOUNTERED;</td>
<td>THIS BINARY CODE IS GENERATED DURING THE PROCESSING OF EXECUTABLE STATEMENTS</td>
<td></td>
</tr>
<tr>
<td>INCLUDES ALL VARIABLES AND ARRAYS IN DIMENSION, TYPE AND DATA STATEMENTS.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>APPEARING IN DECLARATIVES.</td>
<td>THIS AREA IS FORMAL PARAMETER ADDRESS STORAGE (IF THE RUN-STYLE CALLING SEQUENCE IS USED AND THIS SUBPROGRAM HAS MORE THAN SIX FORMAL PARAMETERS)</td>
<td></td>
</tr>
<tr>
<td>NAMELIST STORAGE</td>
<td>THIS IS THE ENTRY POINT TRACEBACK WORD</td>
<td>TRA. (TRACEBACK)</td>
</tr>
<tr>
<td>LIST NAMES NEEDED FOR NAMELIST STATEMENTS</td>
<td>PROGRAM UNIT ENTRY POINT XIT. (ENTRY/EXIT)</td>
<td></td>
</tr>
<tr>
<td>THIS IS THE ENTRY POINT</td>
<td>INSTRUCTIONS TO PRESTORE ACTUAL PARAMETER ADDRESSES AND VARIABLE DIMENSION PRODUCTS (SUBROUTINE OR FUNCTION ONLY); OR, A CALL TO QENTRY (MAIN PROGRAM ONLY) AND SETUP OF FETS (MNF(C) ONLY)</td>
<td></td>
</tr>
<tr>
<td>GENERATED WHEN THE END STATEMENT IS ENCOUNTERED.</td>
<td>(CONTINUED)</td>
<td></td>
</tr>
<tr>
<td>COMMENTS</td>
<td>DESCRIPTION</td>
<td>PSEUDO-COMPASS LOCATION TAG</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>THIS STORAGE IS ASSIGNED AND/OR PRESET DURING PROCESSING OF THE END STATEMENT.</td>
<td>REMOTE PARAMETER LISTS FOR ARITHMETIC STATEMENT FUNCTIONS (AND FOR CALLED ROUTINES IF MNF(C) SPECIFIED)</td>
<td>PPTAG. (ACTUAL PARAMETER TAG)</td>
</tr>
<tr>
<td></td>
<td>SUBSCRIPT EXPRESSIONS USING INNER LOOP INDICES</td>
<td>IND. (INDICES)</td>
</tr>
<tr>
<td></td>
<td>TEMPORARIES FOR SUBSCRIPTS AND EXPRESSIONS</td>
<td>TMP. (TEMPORARIES)</td>
</tr>
<tr>
<td></td>
<td>TEMPORARIES FOR LOOP PARAMETER EXPRESSIONS DURING THE DO/FOR LOOP RANGE</td>
<td>DOTMP. (DO TEMPORARIES)</td>
</tr>
<tr>
<td></td>
<td>CONSTANTS</td>
<td>CON.</td>
</tr>
<tr>
<td></td>
<td>FORMATS</td>
<td>FORMAT</td>
</tr>
<tr>
<td></td>
<td>VARIABLE-DIMENSION VARIABLE VALUES AND PRODUCTS</td>
<td>VARDIM. (VARIABLE DIMENSION)</td>
</tr>
<tr>
<td></td>
<td>SIMPLE VARIABLES NOT SEEN BEFORE THE FIRST EXECUTABLE STATEMENT</td>
<td>VAR. (VARIABLES)</td>
</tr>
<tr>
<td></td>
<td>NAMES AND STATEMENT NUMBERS REQUIRED BY TRACE STATEMENTS</td>
<td>GV. (GENERATED VARIABLES)</td>
</tr>
</tbody>
</table>

G-2
THE FOLLOWING CHART SHOWS THE GENERAL FORM OF A MNF PROGRAM UNIT
(I.E. A MAIN OR SUBPROGRAM). STATEMENTS WITHIN A GROUP (I.E. A BOX)
MAY APPEAR IN ANY ORDER, BUT GROUPS MUST BE ORDERED AS SHOWN.
VARIABLES AND/OR ARRAYS MUST BE TYPED AND DIMENSIONED BEFORE BEING
USED IN DATA OR EQUIVALENCE STATEMENTS.
ALL GROUPS ARE OPTIONAL, EXCEPT THE END LINE.

NOTE - THE FTN ORDER WHICH IS MORE STRICT THAN MNF, IS
RECOMMENDED. THIS IS BECAUSE THE FTN ORDER WILL WORK ON ALL
THREE BKY COMPILERS (AND ON OTHER MACHINES), IS EASIER TO
UNDERSTAND AND TO DEBUG, AND IS MORE EFFICIENT IN COMPILATION.

** MEANS THIS GROUP IS NOT ALLOWED IN BLOCK
DATA SUBPROGRAMS.
THE FOLLOWING CHART SHOWS THE GENERAL FORM OF A FTN PROGRAM UNIT (I.E. A MAIN OR SUBPROGRAM). STATEMENTS WITHIN A GROUP (I.E. A BOX) MAY APPEAR IN ANY ORDER, BUT GROUPS MUST BE ORDERED AS SHOWN. VARIABLES AND/OR ARRAYS MUST BE TYPED AND DIMENSIONED BEFORE BEING USED IN DATA OR EQUIVALENCE STATEMENTS. ALL GROUPS ARE OPTIONAL, EXCEPT THE END LINE.

OVERLAY CARD

| SUBPROGRAM HEADER STATEMENT (I.E. PROGRAM, SUBROUTINE, FUNCTION, OR BLOCK DATA STATEMENT) |

| ** | SPECIFICATION STATEMENTS (I.E. TYPE, DIMENSION, ETC., COMMON, AND EXTERNAL STATEMENTS) |

| ** | EQUIVALENCE STATEMENTS |

| NAMELIST | DATA STATEMENTS |

| ** | STATEMENTS |

| DEFINESTATEMENTS | DEFINITION STATEMENT |

| ** | MUST COME FUNCTIONS |

| ** | BEFORE USE |

| ** | EXECUTABLE |

| ** | AND ENTRY |

| ** | STATEMENTS |

END LINE

** MEANS THIS GROUP IS NOT ALLOWED IN BLOCK DATA SUBPROGRAMS.
RUN76

THE FOLLOWING CHART SHOWS THE GENERAL FORM OF A RUN76 PROGRAM UNIT (I.E. A MAIN OR SUBPROGRAM). STATEMENTS WITHIN A GROUP (I.E. A BOX) MAY APPEAR IN ANY ORDER, BUT GROUPS MUST BE ORDERED AS SHOWN. VARIABLES AND/OR ARRAYS MUST BE TYPED AND DIMENSIONED BEFORE BEING USED IN DATA OR EQUIVALENCE STATEMENTS. ALL GROUPS ARE OPTIONAL, EXCEPT THE END LINE.

NOTE - THE FTN ORDER WHICH IS MORE STRICT THAN RUN76 IS RECOMMENDED. THIS IS BECAUSE THE FTN ORDER WILL WORK ON ALL THREE BKY COMPILERS (AND ON OTHER MACHINES), IS EASIER TO UNDERSTAND AND TO DEBUG, AND IS MORE EFFICIENT IN COMPILATION.

** MEANS THIS GROUP IS NOT ALLOWED IN BLOCK DATA SUBPROGRAMS.
### APPENDIX J

#### MNF FORTRAN STATEMENT LIST

**Explanation of SYMBOL:**

- **P** program unit statement (first statement)
- **D** declarative statement (must appear between program statement and arithmetic statement function definitions)
- **A** arithmetic statement function (must appear between declarative and executable statements)
- **E** executable statement (must appear between arithmetic statement function definitions and END statement)
- **N** END (must be the last statement of a program unit)
- **S** specification statement (may appear anywhere between first statement and last statement)
- **L** listing statement (may appear anywhere)
- ***P** non-standard Fortran statement

#### PROGRAM UNIT STATEMENTS

<table>
<thead>
<tr>
<th>Statement Type</th>
<th>Symbol</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIN</td>
<td></td>
<td>PROGRAM name (file&lt;sub&gt;1&lt;/sub&gt;,...,file&lt;sub&gt;n&lt;/sub&gt;)</td>
</tr>
<tr>
<td>PROGRAM</td>
<td>P</td>
<td>PROGRAM name</td>
</tr>
<tr>
<td>SUBPROGRAM</td>
<td>P</td>
<td>FUNCTION name (P&lt;sub&gt;1&lt;/sub&gt;,...,P&lt;sub&gt;n&lt;/sub&gt;)</td>
</tr>
<tr>
<td></td>
<td>P</td>
<td>type FUNCTION name (P&lt;sub&gt;1&lt;/sub&gt;,...,P&lt;sub&gt;n&lt;/sub&gt;)</td>
</tr>
<tr>
<td></td>
<td>P</td>
<td>SUBROUTINE name (P&lt;sub&gt;1&lt;/sub&gt;,...,P&lt;sub&gt;n&lt;/sub&gt;)</td>
</tr>
<tr>
<td>SPECIFICATION</td>
<td>P</td>
<td>BLOCK DATA</td>
</tr>
<tr>
<td>SUBPROGRAM</td>
<td>P</td>
<td>BLOCK DATA name</td>
</tr>
<tr>
<td>ENTRY POINT</td>
<td>E</td>
<td>ENTRY name</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>EXTERNAL name&lt;sub&gt;1&lt;/sub&gt;,...,name&lt;sub&gt;n&lt;/sub&gt;</td>
</tr>
<tr>
<td>INTERSUBPROGRAM</td>
<td>E</td>
<td>CALL name</td>
</tr>
<tr>
<td>TRANSFER</td>
<td>E</td>
<td>CALL name (P&lt;sub&gt;1&lt;/sub&gt;,...,P&lt;sub&gt;n&lt;/sub&gt;)</td>
</tr>
<tr>
<td>STATEMENTS</td>
<td>E</td>
<td>RETURN</td>
</tr>
<tr>
<td>STATEMENT</td>
<td>A</td>
<td>name (P&lt;sub&gt;1&lt;/sub&gt;,...,P&lt;sub&gt;n&lt;/sub&gt;) = expression</td>
</tr>
<tr>
<td>DATA DECLARATION AND STORAGE ALLOCATION</td>
<td>SYMBOL</td>
<td></td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>TYPE DECLARATION</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTEGER name₁, ..., nameₙ</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>TYPE name₁, ..., nameₙ</td>
<td>*D</td>
<td></td>
</tr>
<tr>
<td>REAL name₁, ..., nameₙ</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>TYPE name₁, ..., nameₙ</td>
<td>*D</td>
<td></td>
</tr>
<tr>
<td>COMPLEX name₁, ..., nameₙ</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>TYPE name₁, ..., nameₙ</td>
<td>*D</td>
<td></td>
</tr>
<tr>
<td>DOUBLE name₁, ..., nameₙ</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>TYPE name₁, ..., nameₙ</td>
<td>*D</td>
<td></td>
</tr>
<tr>
<td>LOGICAL name₁, ..., nameₙ</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>TYPE name₁, ..., nameₙ</td>
<td>*D</td>
<td></td>
</tr>
<tr>
<td>ECS name₁, ..., nameₙ</td>
<td>*D</td>
<td></td>
</tr>
<tr>
<td>TYPE name₁, ..., nameₙ</td>
<td>*D</td>
<td></td>
</tr>
<tr>
<td>STORAGE ALLOCATION</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIMENSION name₁(d₁), ..., nameₙ(dₙ)</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>COMMON name₁, ..., nameₙ</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>COMMON /name₁, ..., nameₙ</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>COMMON /blkname₁/name₁, ..., nameₙ/blkname₂/name₁, ..., nameₙ</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>EQUIVALENCE (name₁, ..., nameₙ)</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>DATA DECLARATION</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATA vlist₁/dlist₁, ..., vlistₙ/dlistₙ</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>DATA (vlist₁ = dlist₁), ..., (vlistₙ = dlistₙ)</td>
<td>*D</td>
<td></td>
</tr>
</tbody>
</table>

ASSIGNMENT AND CONTROL

ASSIGNMENT

v = arithmetic expression
logical v = logical or relational expression
v = masking expression

MULTIPLE

v₁ = ... = vₙ = expression

ASSIGNMENT
CONTROL

GO TO an
GO TO (an,...,an), iv
GO TO (an,...,an), iv
GO TO (sn,...,sn), expression
GO TO (sn,...,sn), expression
GO TO iv, (sn,...,sn)
GO TO iv (sn,...,sn)
GO TO iv
ASSIGN an TO iv

DECISION

IF (arithmetic expression) sn,...,sn, expression
IF (masking expression) sn,...,sn, expression
IF (logical expression) sn,...,sn
IF (logical expression) statement

LOOP

DO an iv = m1,m2,m3
DO an iv = m1,m2
DO an v = expression1,expression2,expression3
DO an v = expression1,expression2
FOR sn iv = m1,m2,m3
FOR sn iv = m1,m2
FOR sn v = expression1,expression2,expression3
FOR sn v = expression1,expression2
an CONTINUE

STOP, PAUSE,

STOP
STOP n
PAUSE
PAUSE n
END

INPUT/OUTPUT

FORMATTED

PRINT fn,iolist
PRINT fn
PUNCH fn,iolist
PUNCH fn
WRITE (u,fn) iolist
WRITE (u,fn)
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ fn, iolist</td>
<td>Binary READ operation</td>
</tr>
<tr>
<td>READ fn</td>
<td>Binary READ operation</td>
</tr>
<tr>
<td>READ (u, fn) iolist</td>
<td>Binary READ operation</td>
</tr>
<tr>
<td>READ (.END. = sn₁, ERR. = sn₂, u, fn) iolist</td>
<td>Binary READ operation</td>
</tr>
<tr>
<td>READ (u, fn)</td>
<td>Format-free READ operation</td>
</tr>
<tr>
<td>READ (.END. = sn₁, ERR. = sn₂, u, fn)</td>
<td>Format-free READ operation</td>
</tr>
<tr>
<td>WRITE (u) iolist</td>
<td>Binary WRITE operation</td>
</tr>
<tr>
<td>WRITE (u)</td>
<td>Format-free WRITE operation</td>
</tr>
<tr>
<td>READ (u) iolist</td>
<td>Binary READ operation</td>
</tr>
<tr>
<td>READ (.END. = sn₁, ERR. = sn₂, u) iolist</td>
<td>Binary READ operation</td>
</tr>
<tr>
<td>PRINT, iolist</td>
<td>Format-free PRINT operation</td>
</tr>
<tr>
<td>PUNCH, iolist</td>
<td>Format-free PUNCH operation</td>
</tr>
<tr>
<td>WRITE (u,) iolist</td>
<td>Binary WRITE operation</td>
</tr>
<tr>
<td>READ, iolist</td>
<td>Binary READ operation</td>
</tr>
<tr>
<td>READ (u,) iolist</td>
<td>Binary READ operation</td>
</tr>
<tr>
<td>READ (.END. = sn₁, ERR. = sn₂, u,) iolist</td>
<td>Binary READ operation</td>
</tr>
<tr>
<td>BUFFER OUT (u, j) (a, b)</td>
<td>Buffered WRITE operation</td>
</tr>
<tr>
<td>BUFFER IN (u, j) (a, b)</td>
<td>Buffered READ operation</td>
</tr>
<tr>
<td>ENCODE (cc, fn, v) iolist</td>
<td>Conversion and Transfer ENCODE operation</td>
</tr>
<tr>
<td>DECODE (cc, fn, b) iolist</td>
<td>Conversion and Transfer DECODE operation</td>
</tr>
<tr>
<td>DECODE (.ERR. = sn₁, cc, fn, v) iolist</td>
<td>Conversion and Transfer DECODE operation</td>
</tr>
<tr>
<td>NAMELIST /lastname, /name₁, ..., nameₙ/</td>
<td>Name-list NAMELIST operation</td>
</tr>
<tr>
<td>WRITE (u, lastname)</td>
<td>Name-list WRITE operation</td>
</tr>
<tr>
<td>READ (u, lastname)</td>
<td>Name-list READ operation</td>
</tr>
<tr>
<td>PRINT lastname</td>
<td>Name-list PRINT operation</td>
</tr>
<tr>
<td>PUNCH lastname</td>
<td>Name-list PUNCH operation</td>
</tr>
<tr>
<td>READ lastname</td>
<td>Name-list READ operation</td>
</tr>
<tr>
<td>REWIND u</td>
<td>File REWIND operation</td>
</tr>
<tr>
<td>REWIND ulist</td>
<td>Format-free REWIND operation</td>
</tr>
<tr>
<td>BACKSPACE u</td>
<td>Binary BACKSPACE operation</td>
</tr>
<tr>
<td>BACKSPACE ulist</td>
<td>Format-free BACKSPACE operation</td>
</tr>
<tr>
<td>ENDFILE u</td>
<td>Binary ENDFILE operation</td>
</tr>
<tr>
<td>ENDFILE ulist</td>
<td>Format-free ENDFILE operation</td>
</tr>
<tr>
<td>UNLOAD u</td>
<td>Binary UNLOAD operation</td>
</tr>
<tr>
<td>UNLOAD ulist</td>
<td>Format-free UNLOAD operation</td>
</tr>
</tbody>
</table>
## INPUT/OUTPUT LISTS AND FORMATS

<table>
<thead>
<tr>
<th>INPUT/OUTPUT LISTS AND FORMATS</th>
<th>SYMBOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPLIEd</td>
<td>((\ldots ((\text{to} \text{list}, l_1_{m_1}, l_2_{m_2}, l_3_{m_3}), i_1_{n_1}, i_2_{n_2}, i_3_{n_3}), \ldots, i_n_{k_1}, k_2, k_3)))</td>
</tr>
<tr>
<td>DO LOOP</td>
<td></td>
</tr>
<tr>
<td>FORMAT</td>
<td>(s_n \text{FORMAT} (f_{s_1}, \ldots, f_{s_n}))</td>
</tr>
<tr>
<td>DATA</td>
<td>(r_{Ew.d})</td>
</tr>
<tr>
<td>CONVERSION</td>
<td>(r_{Fw.d})</td>
</tr>
<tr>
<td></td>
<td>(r_{Gw.d})</td>
</tr>
<tr>
<td></td>
<td>(r_{Dw.d})</td>
</tr>
<tr>
<td></td>
<td>(r_{lw})</td>
</tr>
<tr>
<td></td>
<td>(r_{lw})</td>
</tr>
<tr>
<td></td>
<td>(r_{Aw})</td>
</tr>
<tr>
<td></td>
<td>(r_{Rw})</td>
</tr>
<tr>
<td></td>
<td>(r_{ow})</td>
</tr>
<tr>
<td></td>
<td>(s_{P})</td>
</tr>
<tr>
<td>RECORD AND CHARACTER MANIPULATION</td>
<td>(n_{X})</td>
</tr>
<tr>
<td></td>
<td>(n_{ih_1}, \ldots, h_n)</td>
</tr>
<tr>
<td>MANIPULATION</td>
<td>(<em>\ldots</em>)</td>
</tr>
<tr>
<td></td>
<td>(#\ldots#)</td>
</tr>
<tr>
<td></td>
<td>(n_{T})</td>
</tr>
<tr>
<td></td>
<td>(/)</td>
</tr>
<tr>
<td></td>
<td>(r())</td>
</tr>
</tbody>
</table>

## DEBUGGING AND OUTPUT LISTING CONTROL

<p>| GENERAL                        | (\text{TRACE}) |
| TRACING                        | (\text{NO TRACE}) |
| SPECIFIC TYPE OF TRACING       | (\text{TRACE ARITHMETIC}) |
|                                | (\text{NO TRACE ARITHMETIC}) |
|                                | (\text{TRACE DOLOOPING}) |
|                                | (\text{NO TRACE DOLOOPING}) |
|                                | (\text{TRACE FORMATO}) |
|                                | (\text{NO TRACE FORMATIO}) |
|                                | (\text{TRACE STATEMENT NUMBERS}) |
|                                | (\text{NO TRACE STATEMENT NUMBERS}) |
|                                | (\text{TRACE SUBPROGRAM ENTRY}) |
|                                | (\text{NO TRACE SUBPROGRAM ENTRY}) |
|                                | (\text{TRACE SUBPROGRAM CALLS}) |
|                                | (\text{NO TRACE SUBPROGRAM CALLS}) |</p>
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace Subscripts</td>
<td>*L</td>
</tr>
<tr>
<td>No Trace Subscripts</td>
<td>*L</td>
</tr>
<tr>
<td>Trace Transfers</td>
<td>*L</td>
</tr>
<tr>
<td>No Trace Transfers</td>
<td>*L</td>
</tr>
<tr>
<td>Specific</td>
<td>Track list</td>
</tr>
<tr>
<td>Element Tracing</td>
<td>No Trace list</td>
</tr>
<tr>
<td>Source</td>
<td>List</td>
</tr>
<tr>
<td>Listing</td>
<td>No List</td>
</tr>
<tr>
<td>Control</td>
<td>Page</td>
</tr>
<tr>
<td>Pseudo-Compass</td>
<td>Code</td>
</tr>
<tr>
<td>Listing Control</td>
<td>No Code</td>
</tr>
<tr>
<td>Cross</td>
<td>References</td>
</tr>
<tr>
<td>Reference Map</td>
<td>No References</td>
</tr>
<tr>
<td>Overlay</td>
<td>Overlay (filename,i,j)</td>
</tr>
</tbody>
</table>

*L
Fortran Library Functions and Subroutines

Two lists are printed below. The first is a list of the functions and subroutines by category and each is numbered. The second is an alphabetical list and also serves as an index for the first list.

In the category list, the definition uses a, or a to denote formal parameters. Type complex parameters are also denoted this way or as \( x + iy \).

In the column labelled "Compiler Type" which uses I for "intrinsic" and L for "library resident," subprograms labelled I can be one of two forms: the first is actually called intrinsic (or in-line) because code for the subprogram is inserted into the program at the point where it is referenced. The second form, although library resident, has a special calling sequence and is thus virtually a reserved name (see Appendix R) — a user subprogram having the same name and argument types must recognize the same calling sequence and would have to be written in COLOSSAL assembly language. If a user references a subprogram using a different number of arguments than the number in this list or if any arguments are of a different type than those given in the list, a caution level compilation diagnostic is given and it is assumed that the user has provided his own subprogram with that name. Further, a normal FORTRAN calling sequence will be generated to use it.

If an array has the same name as an intrinsic subprogram, a caution level diagnostic is given during compilation since the use as an array removes the intrinsic name from use as a subprogram in the current program unit. It is possible (although non-standard) for the user to supply his own subprogram having the same name as an intrinsic subprogram. However, because of the almost reserved status of these names, the user must have the name in an EXTERNAL statement or else the intrinsic subprogram will be referenced instead of the user's subprogram.

Subprograms labelled L for library resident are not reserved names and may be replaced by user subprograms of the same name and argument types. No compilation message will be given in this case. However, if the argument types differ, a caution level compilation diagnostic is given. This message may be suppressed by using the subprogram name in an EXTERNAL statement.

In the tables below, an "octal" result means that the result has no particular type associated with it. It acts the same way as an octal constant — see 2.4.6.
## Fortran Library Functions

<table>
<thead>
<tr>
<th>Symbolic Name</th>
<th>Definition and Limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>Absolute value $</td>
</tr>
<tr>
<td>IABS</td>
<td>Modulus of $a$, i.e. $\sqrt{\text{REAL}^2(a) + \text{AYMA}^2(a)}$</td>
</tr>
<tr>
<td>DABS</td>
<td>Conversion from integer to real</td>
</tr>
<tr>
<td>CABS</td>
<td>Conversion from real to integer with truncation (same as INT)</td>
</tr>
<tr>
<td>AINT</td>
<td>Truncation of $a$: $[a] = \text{Sign of } a \times \text{largest integer } \leq</td>
</tr>
<tr>
<td>INT</td>
<td></td>
</tr>
<tr>
<td>IDINT</td>
<td></td>
</tr>
<tr>
<td>AMOD</td>
<td>Remaindering. Result is $a_1 - \lfloor a_1/a_2 \rfloor \times a_2$, where $[x]$ is defined as the integer resulting from the truncation of $a_1/a_2$.</td>
</tr>
<tr>
<td>MOD</td>
<td></td>
</tr>
<tr>
<td>DMOD</td>
<td>If $a_2$ is zero, these functions are undefined. For DMOD, $</td>
</tr>
<tr>
<td>AMAXO</td>
<td>Determine maximum argument. Maximum $(a_1, a_2, \ldots)$</td>
</tr>
<tr>
<td>AMAX1</td>
<td>For AMAXO and MAXO, $</td>
</tr>
<tr>
<td>MAXO</td>
<td></td>
</tr>
<tr>
<td>MAX1</td>
<td></td>
</tr>
<tr>
<td>DMAX1</td>
<td></td>
</tr>
<tr>
<td>AMINO</td>
<td>Determine minimum argument. Minimum $(a_1, a_2, \ldots)$</td>
</tr>
<tr>
<td>AUN1</td>
<td>For AMINO and MINO, $</td>
</tr>
<tr>
<td>MINO</td>
<td></td>
</tr>
<tr>
<td>MIN1</td>
<td></td>
</tr>
<tr>
<td>DMIN1</td>
<td></td>
</tr>
<tr>
<td>SGN</td>
<td>Transfer of sign. Sign of $a_2$ times $</td>
</tr>
<tr>
<td>ISIGN</td>
<td>If $a_2 = 0$, use $</td>
</tr>
<tr>
<td>DSGN</td>
<td></td>
</tr>
<tr>
<td>DIM</td>
<td>Positive difference: $a_1 - \text{minimum } (a_1, a_2)$</td>
</tr>
<tr>
<td>IDIM</td>
<td>For IDIM, $</td>
</tr>
<tr>
<td>SNGL</td>
<td>Truncate to obtain most significant part of double precision argument</td>
</tr>
<tr>
<td>NULG</td>
<td>Express single-precision argument in double precision form</td>
</tr>
<tr>
<td>Standard or Non-standard</td>
<td>Number of Arguments</td>
</tr>
<tr>
<td>-------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>FORTRAN*</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>≥ 2</td>
</tr>
<tr>
<td></td>
<td>≥ 2</td>
</tr>
<tr>
<td></td>
<td>≥ 2</td>
</tr>
<tr>
<td></td>
<td>≥ 2</td>
</tr>
<tr>
<td></td>
<td>≥ 2</td>
</tr>
<tr>
<td></td>
<td>≥ 2</td>
</tr>
<tr>
<td></td>
<td>≥ 2</td>
</tr>
<tr>
<td></td>
<td>≥ 2</td>
</tr>
<tr>
<td></td>
<td>≥ 2</td>
</tr>
<tr>
<td></td>
<td>≥ 2</td>
</tr>
<tr>
<td></td>
<td>≥ 2</td>
</tr>
<tr>
<td></td>
<td>≥ 2</td>
</tr>
<tr>
<td></td>
<td>≥ 2</td>
</tr>
<tr>
<td></td>
<td>≥ 2</td>
</tr>
<tr>
<td></td>
<td>≥ 2</td>
</tr>
<tr>
<td></td>
<td>≥ 2</td>
</tr>
<tr>
<td></td>
<td>≥ 2</td>
</tr>
</tbody>
</table>

* S means listed in the ANSI standard, N means not listed
** I means intrinsic, L means library resident
Symbolic Name | Definition and Limitations
---|---
30 REAL | Obtain real part of complex argument
31 AIMAG | Obtain imaginary part of complex argument
32 CMPLX | Convert two real arguments to complex form \( a_1 + a_2 \cdot \sqrt{-1} \)
33 CONJG | Obtain conjugate of a complex argument
34 AND | Boolean AND \( a_1 \land a_2 \land \ldots \land a_n \)
35 OR | Boolean OR \( a_1 \lor a_2 \lor \ldots \lor a_n \)
36 EOR | Boolean Exclusive OR
37 XOR | Logical complement \( \neg a \)
38 COMPL | Logical complement \( \neg a \)
39 SHIFT | Shift \( a_1 \) by \( a_2 \) bit positions: left circular if \( a_2 \) is positive, right end-off with sign extension if \( a_2 \) is negative
40 LRSHIFT | Count the number of bits in a word that are 1
41 ICOUNT | Logical complement \( \neg a \)
42 EXP | Exponential, \( e^a \). For complex: \( e^{\cos(y) + i \cdot \sin(y)} \)
43 DEXP | For real and double precision: \( a < 740.3 \)
44 CEXP | For complex: \( x < 740.3, |y| < 1.4 \cdot 10^{14} \). If \( a \) (or \( x \)) \(< -675.82 \), result is zero.
45 ALOG | Natural logarithm, \( \log_e (a) \). For complex: \( 0.5 \cdot \log_e (x^2 + y^2) + i \cdot \arctan(y/x) \)
46 DLOG | For real and double precision: \( a > 0.0 \)
47 CLOG | For complex: \( x \neq 0.0 \) or \( y \neq 0.0 \)
48 ALOG10 | Common logarithm, \( \log_{10} (a) \). \( a > 0.0 \)
49 DLOG10 | For real and double precision: \( a > 0.0 \)
50 SQRT | Square root, \( (a)^{1/2} \). For complex: \( (x^2 + y^2)^{1/2} \)
51 DSQRT | \( (\cos(\arctan(y/x)) + i \cdot \sin(\arctan(y/x))) \) For real and double precision: \( a > 0.0 \) For complex, \( x \neq 0.0 \) or \( y \neq 0.0 \)
52 CSQRT | Trigonometric sine, \( \sin(a) \), \( a \) in radians. For real and double precision: \( |a| < 1.4 \cdot 10^{14} \). For complex, \( |x| < 1.4 \cdot 10^{14}, |y| < 740.3 \)
53 SIN | \( \sin(x) \cdot \cosh(y) + i \cdot \cos(x) \cdot \sinh(y) \)
54 DSIN | Trigonometric cosine, \( \cos(a) \), \( a \) in radians. For real and double precision: \( |a| < 1.4 \cdot 10^{14} \). For complex, \( |x| < 1.4 \cdot 10^{14}, |y| < 740.3 \)
55 COS | \( \cos(x) \cdot \cosh(y) - i \cdot \sin(x) \cdot \sinh(y) \)
56 DCOS | \( \sinh(a) = \frac{e^a - e^{-a}}{2} \quad \cosh(a) = \frac{e^a + e^{-a}}{2} \)
<table>
<thead>
<tr>
<th>Standard or Non-standard FORTRAN*</th>
<th>Number of Arguments</th>
<th>Type of Argument(s)</th>
<th>Function Result</th>
<th>Compiler Type**</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>1</td>
<td>COMPLEX</td>
<td>REAL</td>
<td>I</td>
<td>COMPLEX C $ R=REAL(C)</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>COMPLEX</td>
<td>REAL</td>
<td>I</td>
<td>COMPLEX C $ R=AIMAG(C)</td>
</tr>
<tr>
<td>S</td>
<td>2</td>
<td>REAL</td>
<td>COMPLEX</td>
<td>I</td>
<td>COMPLEX C $ C=CMPLX(A,B)</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>COMPLEX</td>
<td>COMPLEX</td>
<td>I</td>
<td>COMPLEX C,D $ C=CONJG(D)</td>
</tr>
<tr>
<td>N</td>
<td>≥ 2</td>
<td>Single words Octal</td>
<td>I</td>
<td>COMORB(A1,A2)</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>≥ 2</td>
<td>Single words Octal</td>
<td>I</td>
<td>C=OR(R1,R2)</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>≥ 2</td>
<td>Single words Octal</td>
<td>I</td>
<td>D=OR(C1,C2)</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>≥ 2</td>
<td>Single words Octal</td>
<td>I</td>
<td>E=OR(D1,D2)</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>1</td>
<td>Single word Octal</td>
<td>I</td>
<td>B=COMPL(A)</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>2</td>
<td>Single word Octal</td>
<td>I</td>
<td>B=SHIFT(A,1)</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>2</td>
<td>Integer</td>
<td>Octal</td>
<td>R=LSHIFT(X,-3)</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>1</td>
<td>Integer</td>
<td>INTEGER</td>
<td>J=ICOUNT(H)</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>REAL</td>
<td>REAL</td>
<td>L</td>
<td>R=EXP(Y)</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
<td>L</td>
<td>DOUBLE X,Y $ X=DEXP(Y)</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>COMPLEX</td>
<td>COMPLEX</td>
<td>L</td>
<td>COMPLEX Z,W $ W=CEXP(Z)</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>REAL</td>
<td>REAL</td>
<td>L</td>
<td>Z=ALOG(Y)</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
<td>L</td>
<td>DOUBLE U,V $ U=DLGCV(V)</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>COMPLEX</td>
<td>COMPLEX</td>
<td>L</td>
<td>COMPLEX I,J $ J=CLOG1(I)</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>REAL</td>
<td>REAL</td>
<td>L</td>
<td>B=ALOG10(A)</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
<td>L</td>
<td>DOUBLE T,U $ T=DLOG10(U)</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>REAL</td>
<td>REAL</td>
<td>L</td>
<td>Y=SQRT(X)</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
<td>L</td>
<td>DOUBLE Z,U $ Z=DSQRT(W)</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>COMPLEX</td>
<td>COMPLEX</td>
<td>L</td>
<td>COMPLEX A,B $ A=CSQRT(B)</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>REAL</td>
<td>REAL</td>
<td>L</td>
<td>X=SIN(Y)</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
<td>L</td>
<td>DOUBLE E,F $ E=SIN(F)</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>COMPLEX</td>
<td>COMPLEX</td>
<td>L</td>
<td>COMPLEX C,D $ C=CSIN(D)</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>REAL</td>
<td>REAL</td>
<td>L</td>
<td>Z=CO5(W)</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
<td>L</td>
<td>DOUBLE C,J $ G=DCOS(H)</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>COMPLEX</td>
<td>COMPLEX</td>
<td>L</td>
<td>COMPLEX I,J $ I=CCOS(J)</td>
</tr>
</tbody>
</table>

* S means listed in the ANSI standard, N means not listed
** I means intrinsic, L means library resident
<table>
<thead>
<tr>
<th>Symbolic Name</th>
<th>Definition and Limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td>59 TAN</td>
<td>Trigonometric tangent, ( \tan(a) ), ( a ) in radians. (</td>
</tr>
<tr>
<td>60 ASIN</td>
<td>Arcsin (a), (</td>
</tr>
<tr>
<td>61 ACOS</td>
<td>Arccos (a), (</td>
</tr>
<tr>
<td>62 ATAN</td>
<td>Arctan (a), ( -\pi/2 \leq \arctan(a) \leq \pi/2 ) (radians)</td>
</tr>
<tr>
<td>63 DATAN</td>
<td>Hyperbolic tangent: ( \tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}} ), (</td>
</tr>
<tr>
<td>64 ATAN2</td>
<td>( \arctan\left(\frac{a_1}{a_2}\right) ), not both ( a_1 = 0.0 ) and ( a_2 = 0.0 ) ( -\pi \leq \arctan\left(\frac{a_1}{a_2}\right) \leq \pi ) (radians)</td>
</tr>
<tr>
<td>65 DATAN2</td>
<td>I/O status on unit a: = 0 if no EOF on previous read</td>
</tr>
<tr>
<td>66 TANH</td>
<td>Address of argument a</td>
</tr>
<tr>
<td>67 EOF</td>
<td>Returns -1 if a is indefinite, +1 if out of range (infinite) and 0 if normal</td>
</tr>
<tr>
<td>68 IOCHEC</td>
<td>Uniform random number generator on ((0.0,1.0)). The next random number is generated each time RANF is referenced. The argument a is ignored. The initial generative value is set by default unless RANSET is used.</td>
</tr>
<tr>
<td>69 UNIT</td>
<td>Returns floating-point CP seconds since beginning of job</td>
</tr>
<tr>
<td>70 LENGTH</td>
<td>Returns integer CP milliseconds since beginning of job</td>
</tr>
<tr>
<td>71 LORC</td>
<td>Returns integer CP milliseconds since beginning of job</td>
</tr>
<tr>
<td>72 LEGVAR</td>
<td>Uniform random number generator on ((0.0,1.0)). The next random number is generated each time RANF is referenced. The argument a is ignored. The initial generative value is set by default unless RANSET is used.</td>
</tr>
<tr>
<td>Standard or Non-standard FORTRAN*</td>
<td>Number of Arguments</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
</tr>
<tr>
<td>S</td>
<td>2</td>
</tr>
<tr>
<td>S</td>
<td>2</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
</tr>
</tbody>
</table>

* S means listed in the ANSI standard, N means not listed
** I means intrinsic, L means library resident
Portran Library Subroutines

<table>
<thead>
<tr>
<th>Symbolic</th>
<th>Definition and Limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong> EXIT</td>
<td>Terminate program execution and return control to operating system.</td>
</tr>
<tr>
<td><strong>2</strong> SLITE</td>
<td>Turn on or turn off sense lights. If ( 1 \leq i \leq 6 ), turn on sense light ( i ). If ( i=0 ), turn all sense lights off. Otherwise, ignore.</td>
</tr>
<tr>
<td><strong>3</strong> SLITET</td>
<td>Test, then turn off sense light. If ( 1 \leq i \leq 6 ), and sense light ( i ) is on, then ( j=1 ) and sense light ( i ) is turned off; else ( j=2 ).</td>
</tr>
<tr>
<td><strong>4</strong> SSWITCH</td>
<td>Test sense switch. If ( 1 \leq i \leq 6 ) and sense switch ( i ) is on, then ( j=1 ); else ( j=2 ). See SCOPE manual for information on setting sense switches on or off.</td>
</tr>
<tr>
<td><strong>5</strong> REMARK*</td>
<td>Place a message of up to 80 display code characters in the dayfile. The message must be terminated with 12 binary zeros (0000,0) in the lowest bits of the last word of the message even if an entire word is necessary.</td>
</tr>
<tr>
<td><strong>6</strong> DISPLA*</td>
<td>Place a message of up to 40 display code characters in the dayfile followed by a numerical value given by the second argument. The message must be terminated with a zero word and must not contain 12 binary zeros in the lowest 12 bits of any of the message words. The value is displayed as an integer if not normalized, or is displayed in floating-point if normalized (bits 59 and 47 are opposite i.e. 0 and 1 for positive numbers, 1 and 0 for negative numbers).</td>
</tr>
<tr>
<td><strong>7</strong> TIME**</td>
<td>Place the central processor time used by the job followed by a message of up to 50 display code characters on the OUTPUT file. The message must be terminated with 8 binary zero bits (00000,0) even if an entire word is necessary.</td>
</tr>
<tr>
<td><strong>8</strong> SECOND</td>
<td>After calling SECOND, the argument ( x ) contains the central processor time used by the job in floating-point seconds (accurate to the nearest millisecond).</td>
</tr>
<tr>
<td><strong>9</strong> DATE</td>
<td>After calling DATE, the argument ( x ) contains the current date in an A10 format: ( bmm/dd/yyb ), where ( b ) is a blank character, ( mm ) is the month, ( dd ) is the day, and ( yy ) is the year.</td>
</tr>
</tbody>
</table>

* The legal message characters are A through Z, 0 through 9, + - * / ( ) $ = space, comma, and period, i.e. display codes 01,8 through 57,8.
** This is different from the RUN and FTN manual version of the TIME routine.
<table>
<thead>
<tr>
<th>Standard or Non-standard FORTRAN*</th>
<th>Number of Arguments</th>
<th>Type of Argument(s)</th>
<th>Compiler Type**</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>0</td>
<td>(none)</td>
<td>I</td>
<td>CALL EXIT</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
<td>INTEGER</td>
<td>I</td>
<td>CALL SLITE(3)</td>
</tr>
<tr>
<td>N</td>
<td>2</td>
<td>a₁ INTEGER, a₂ INTEGER variable</td>
<td>I</td>
<td>CALL SLITE(2,J)</td>
</tr>
<tr>
<td>N</td>
<td>2</td>
<td>a₁ INTEGER, a₂ INTEGER variable</td>
<td>I</td>
<td>CALL SWITCH(1,1)</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
<td>Hollerith</td>
<td>L</td>
<td>CALL REMARK (SUBFGIN)</td>
</tr>
<tr>
<td>N</td>
<td>2</td>
<td>a₁ Hollerith, a₂ REAL, or INTEGER</td>
<td>L</td>
<td>CALL DISPLA (ZM*, 20.2)</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
<td>Hollerith</td>
<td>L</td>
<td>CALL TIME (9THCALCULATE)</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
<td>REAL</td>
<td>L</td>
<td>CALL SECOND (TIMER)</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
<td>Hollerith</td>
<td>L</td>
<td>CALL DATE (IDAT)</td>
</tr>
</tbody>
</table>

* S means listed in the ANSI standard, N means not listed.
** I means intrinsic, L means library resident
<table>
<thead>
<tr>
<th>Symbolic Name</th>
<th>Definition and Limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDATE</td>
<td>After calling JDATE, the argument a contains the year and day of the year in an A10 format: bbbyyddbb where b is a blank character, yy is the year and dd is the integer number of the day of the year (January 1 is 1).</td>
</tr>
<tr>
<td>CLOCK</td>
<td>After calling CLOCK, the argument a contains the time of day in an A10 format: bhh.mm.ssbb where b is a blank character, hh is hours, mm is minutes, and ss is seconds.</td>
</tr>
<tr>
<td>RANGET</td>
<td>After calling RANGET, the argument a contains the current generative value of RANF.</td>
</tr>
<tr>
<td>PANSST</td>
<td>Initialize generative value of RANF. Usually used in conjunction with RANGET for stop and restart capability. If the generative value is defined by the user, it must be a positive, odd, floated, (bit 58 is a 1) unnormalized integer.</td>
</tr>
<tr>
<td>OPENMS</td>
<td>Open mass storage random access file</td>
</tr>
<tr>
<td>STINDX</td>
<td>Change index array of mass storage random access file</td>
</tr>
<tr>
<td>READMS</td>
<td>Read mass storage random access file</td>
</tr>
<tr>
<td>WRITMS</td>
<td>Write mass storage random access file</td>
</tr>
<tr>
<td>READEC</td>
<td>Read extended core</td>
</tr>
<tr>
<td>WRITEC</td>
<td>Write extended core</td>
</tr>
<tr>
<td>FYTBNR</td>
<td>Blocked binary flag</td>
</tr>
<tr>
<td>Standard or Non-standard FORTRAN*</td>
<td>Number of Arguments</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>N</td>
<td>4</td>
</tr>
<tr>
<td>N</td>
<td>3</td>
</tr>
<tr>
<td>N</td>
<td>4</td>
</tr>
<tr>
<td>N</td>
<td>4</td>
</tr>
<tr>
<td>N</td>
<td>3</td>
</tr>
<tr>
<td>N</td>
<td>3</td>
</tr>
<tr>
<td>N</td>
<td>3</td>
</tr>
</tbody>
</table>

* S means listed in the ANSI standard, N means not listed.

** I means intrinsic, L means library resident
### Alphabetic List and Index

<table>
<thead>
<tr>
<th>Name</th>
<th>Page</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>ACOS</td>
<td>6</td>
<td>61</td>
</tr>
<tr>
<td>AIMAG</td>
<td>4</td>
<td>31</td>
</tr>
<tr>
<td>AINT</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>ALOG</td>
<td>4</td>
<td>45</td>
</tr>
<tr>
<td>ALOG10</td>
<td>4</td>
<td>48</td>
</tr>
<tr>
<td>ASX</td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>AMAX1</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>AIINO</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>AMLN1</td>
<td>2</td>
<td>19</td>
</tr>
<tr>
<td>AMXD</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>AND</td>
<td>4</td>
<td>34</td>
</tr>
<tr>
<td>ASIN</td>
<td>6</td>
<td>60</td>
</tr>
<tr>
<td>ATAN</td>
<td>6</td>
<td>62</td>
</tr>
<tr>
<td>ATAN2</td>
<td>6</td>
<td>64</td>
</tr>
<tr>
<td>CABS</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>COOS</td>
<td>4</td>
<td>58</td>
</tr>
<tr>
<td>CEXP</td>
<td>4</td>
<td>44</td>
</tr>
<tr>
<td>CLOCK</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>CLOG</td>
<td>4</td>
<td>47</td>
</tr>
<tr>
<td>CHPLX</td>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>COMPL</td>
<td>4</td>
<td>38</td>
</tr>
<tr>
<td>COMJG</td>
<td>4</td>
<td>33</td>
</tr>
<tr>
<td>COS</td>
<td>4</td>
<td>56</td>
</tr>
<tr>
<td>CPTIME</td>
<td>6</td>
<td>73</td>
</tr>
<tr>
<td>CSIN</td>
<td>4</td>
<td>55</td>
</tr>
<tr>
<td>CSORT</td>
<td>4</td>
<td>52</td>
</tr>
<tr>
<td>DABS</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>DATAN</td>
<td>6</td>
<td>63</td>
</tr>
<tr>
<td>DATAN2</td>
<td>6</td>
<td>65</td>
</tr>
<tr>
<td>DATE</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>DBLE</td>
<td>2</td>
<td>29</td>
</tr>
<tr>
<td>DCOS</td>
<td>4</td>
<td>57</td>
</tr>
<tr>
<td>DEXP</td>
<td>4</td>
<td>43</td>
</tr>
<tr>
<td>Name</td>
<td>Page</td>
<td>Line</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>DIM</td>
<td>2</td>
<td>26</td>
</tr>
<tr>
<td>DISPLA</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>DLOG</td>
<td>4</td>
<td>46</td>
</tr>
<tr>
<td>DLOG10</td>
<td>4</td>
<td>49</td>
</tr>
<tr>
<td>DHAXI</td>
<td>2</td>
<td>17</td>
</tr>
<tr>
<td>DIN1</td>
<td>2</td>
<td>22</td>
</tr>
<tr>
<td>DMOD</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>DSIGN</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>DSIN</td>
<td>4</td>
<td>54</td>
</tr>
<tr>
<td>DSQRT</td>
<td>4</td>
<td>51</td>
</tr>
<tr>
<td>EOF</td>
<td>6</td>
<td>67</td>
</tr>
<tr>
<td>EOR</td>
<td>4</td>
<td>36</td>
</tr>
<tr>
<td>EXIT</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>EXP</td>
<td>4</td>
<td>42</td>
</tr>
<tr>
<td>FLOAT</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>FTNBIN</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>IABS</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ICOUNT</td>
<td>4</td>
<td>41</td>
</tr>
<tr>
<td>ICPTIME</td>
<td>6</td>
<td>74</td>
</tr>
<tr>
<td>IDIM</td>
<td>2</td>
<td>27</td>
</tr>
<tr>
<td>IDINT</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>IFIX</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>INT</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>IOCHRC</td>
<td>6</td>
<td>65</td>
</tr>
<tr>
<td>ISIGN</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>JDATE</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>LEGVAR</td>
<td>6</td>
<td>72</td>
</tr>
<tr>
<td>LENGTH</td>
<td>6</td>
<td>70</td>
</tr>
<tr>
<td>LRSHFT</td>
<td>4</td>
<td>40</td>
</tr>
<tr>
<td>LOCF</td>
<td>6</td>
<td>71</td>
</tr>
<tr>
<td>MAXD</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>MAXI</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>MINO</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>MIN1</td>
<td>2</td>
<td>21</td>
</tr>
<tr>
<td>MOD</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>OPERMS</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>OR</td>
<td>4</td>
<td>35</td>
</tr>
<tr>
<td>Name</td>
<td>Page</td>
<td>Line</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>RANF</td>
<td>6</td>
<td>75</td>
</tr>
<tr>
<td>RANGET</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>RANSET</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>READEC</td>
<td>10</td>
<td>18</td>
</tr>
<tr>
<td>READMS</td>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>REAL</td>
<td>4</td>
<td>30</td>
</tr>
<tr>
<td>REMARK</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>SECOND</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>SHIFT</td>
<td>4</td>
<td>39</td>
</tr>
<tr>
<td>SIGN</td>
<td>2</td>
<td>23</td>
</tr>
<tr>
<td>SIN</td>
<td>4</td>
<td>53</td>
</tr>
<tr>
<td>SLITE</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>SLITET</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>SNGL</td>
<td>2</td>
<td>28</td>
</tr>
<tr>
<td>SQRT</td>
<td>4</td>
<td>50</td>
</tr>
<tr>
<td>SWTCH</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>STINDX</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>TAN</td>
<td>6</td>
<td>59</td>
</tr>
<tr>
<td>TANF</td>
<td>6</td>
<td>66</td>
</tr>
<tr>
<td>TIME</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>UNIT</td>
<td>6</td>
<td>69</td>
</tr>
<tr>
<td>WRITEC</td>
<td>10</td>
<td>19</td>
</tr>
<tr>
<td>WRITMS</td>
<td>10</td>
<td>17</td>
</tr>
<tr>
<td>XOR</td>
<td>4</td>
<td>37</td>
</tr>
</tbody>
</table>
LEGAL NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Atomic Energy Commission, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights.