Title
THE 'MNF' FORTRAN COMPILER

Author
Schofield, Clive F.

Publication Date
1973

THE 'MNF' FORTRAN COMPILER

Clive F. Schofield

January 1973

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

DISCLAIMER

LBL-1570

UNIVERSITY OF CALIFORNIA

Lawrence Berkeley Laboratory
Berkeley, California

AEC Contract No. W-7405-eng-48

THE 'MNF' FORTRAN COMPILER

Clive F. Schofield

January 1973

TABLE OF CONTENTS

APPENDICES

*9. Explanation of MNF source listing

10. Optimisation of FORTRAN programs

*11. Layout of execution code and compile core usage

* = IMPORTANT INFORMATION FOR ALL MNF USERS

0.0 Introduction

This document assumes the reader to have some knowledge of the CDC FORTRANs and of SCOPE, or a similar operating system.

The MNF compiler was written at the University of Minnesota between 1967 and 1971, the main authors being L.A. Liddiard and E.J. Mundstock. Their aim was to produce a compiler which was compatible with the CDC compilers (RUN, FUN, and FTN) and to provide:

(1) extensive compile time error and cautionary messages, and good cross reference listings;

(2) good execution code [better than RUN, perhaps not as good as FTN(OPT=2)];

(3) execution time tracing statements - which, if used, would not unduly degrade (2);

(4) logical extensions to the language such as expressions as DO parameters, and format free I/O statements; and

(5) efficient compilation.

The aim, then, was to produce the maximum information per run (in order to reduce the number of runs), and to write a fast compiler producing good code (in order to make those runs as cheap as possible).

0.1 Present State of MNF

The compiler has been in serious production use in (at least) the Universities of Minnesota and London for over 2 years. As the availability of MNF has become more widely known so have more installations used it -

and there are now about a dozen users. Recently, the University of Minnesota has provided CDC with the compiler to be installed in the INCOSYL library.

Minnesota still maintain the compiler and issue corrective code about 3 times a year. In addition to correcting bugs this code usually provides extensions to the compiler's facilities. Since MNF is maintained by its authors the quality of the corrective code is excellent. Following CDC terminology, new code is issued as a 'PSR'. The present version of the compiler (February 1973) is 'PSR 12' - but the version used at BKY is PSR9, since it takes time to install the new code and to rewrite any local changes.

0.2 <u>What MNF contains</u>

The MNF compiler is one COMPASS routine of about 70000 statements. In addition, Minnesota provided new versions of the often used FORTRAN mathematical functions (SQRT, SIN, EXP, etc.) These are both faster and more accurate than the equivalent RUN/FUN or FTN functions. Also provided is a new formatted I/O library of 5 routines, which is again faster (up to twice as fast) than the CDC libraries. These execution routines could also be used by RUN/FUN or FTN with some change to those compilers. The routines also handle FORMAT-free I/O statements.

MNF is at present being run on most recent CDC systems, including:

	SCOPE 3.1,3.2,3.3 and 3.4	(6000 computers)
and	SCOPE 2.0, and SCOPE 1.0	(7600 computer)
and	KRONOS	(6000 computers)

Because of its good design, MNF has proved amenable to arbitrary changes in operating systems. However, transferring it to SCOPE 2.0 was not

trivial, and required about 4000 cards changing.

0.3 Special Compiler Features

MNF has (of course) been written so that it can compile several programs 'at once' [i.e., the compiler is re-entrant]. It is not run in re-entrant mode because the system overheads in saving tables etc. (core areas) have been found to degrade compile speed by a factor of 3. This would, however, still leave MNF about 3 times faster than FTN - so we are talking about orders of magnitude of improvement in this case.

A further facility of MNF is to operate in 'BATCH' mode. This means that the compiler can accept many (student) jobs presented as one file. This mode of operation is very efficient when many small jobs are to be run, but has not yet been implemented under SCOPE 2.0. It is not used at London, due to operational difficulties, but is used extensively at Minnesota where a typical 'batch' may contain about 8 jobs. In this mode the compiler does its own banner pages and accounting, and so needs modification to suit the whims of the installation. One set of figures which demonstrate the efficiency of MNF batch is provided by the 'OHIO STATE Benchmark' the (real time) figures being

SCOPE 3.2 - FTN - 7.5 mins
" - MNF - 2.5 "
" - MNF(batch)-0.75 mins

This benchmark contains about 150 short FORTRAN jobs.

In addition, the KRONOS version of MNF is well adapted to the needs of TELEX users.

These various modes of MNF (BATCH,SCOPE 2.0, etc.) are selected by assembly time switches. The SCOPE 2.0 version has been implemented at the University of London.

0.4 Documentation

Minnesota produced a MNF Reference Manual which contains good descriptions of the new features of MNF. Since the new features include execution trace statements, free format I/O and the like - some parts of this manual are vital. Copies of the most important sections of this manual are attached as Appendices 5 to 11. A partial IMS is available which includes flow charts and table formats, but there is little explanatory text. The IMS is improved from time to time, but at present is useful only to expert maintainers.

0.5 MNF Performance

It is very difficult to compare compiler performance, however, a set of 30 'typical' FORTRAN jobs run on a 6400 provided:-

Compiler	Total CP Time (sum of dayfiles) MINUTES	Total "Real Time" (from system dayfile) MINUTES
FTN3.0 (OPT=2)	30.6	39
FUN	28.9	43
MNF	24.2	30
MNF(T)	35.2*	(not measured)

[All the jobs reached normal termination - STOP/END]

*MNF(T) is like FTN with all DEBUG options 'on' - i.e., TRACE ARRAYS etc.

So MNF appears to beat FTN in its highest optimization mode. In execution CP time only, MNF was faster than FTN (OPT=2) in 18 out of the 30 jobs.

In 13 cases FUN was faster than MNF in execution CP time. The league table was (on 6400 - 65K SCM SCOPE3.2):-

Compiler	No. of Jobs with fastest xeq. CP time	No. of Jobs with fastest compile CP time	No. of Jobs with fastest total CP time	No. of Jobs with fastest total PP time
FTN (OPT=2)	6	0	4	0
FUN.	8	1	16 ***	18 **
MNF.	16	29	10	12 **

** MNF was printing a cross reference listing which, with short routines, doubles the compiler output. At London we have deleted the FUN cross reference map because it contained no useful information.

*** When FUN was bad it was really bad. FTN and MNF were never really bad in execution. FTN was really bad in compilation however.

In compilation MNF was always greatly superior to FTN. The execellent 'real time' of MNF is partly due to its small demands on the system compared to FTN. On the 7600 at London (32KSCM) MNF compile rates have varied from 50,000 to 200,000 statements per minute (will all source listings printed). The average compile rate on the BKY 7600 using SCOPE 1.1.4 is 120,000 statements per minute.

The FTN (OPT=1) rate for these jobs varies between 500 and 30,000 statements per minute The RUN(FUN) rate is always better than FTN but never as good as MNF.

These league tables are confusing. Since there seem to be few experts in 'throughput', I prefer to believe the real time figures which reduce to MNF=1 minute /FTN=1.3 /FUN=1.4. [FUN is more or less the same as RUN76].

Referring back to section 0.0 it must be realized that the above times do not take account of the 'maximum information per run'. The number of runs of any program would be reduced by the better messages etc. of MNF. This is something that is very hard to measure, but I have known serious programs where the number of runs may have been halved. So perhaps we should improve the MNF figures by a factor of 2. Note, however, that the use of full execution tracing degrades MNF CP execution time by about 50%, TRACE SUBSCRIPTS (bounds checking) being fairly expensive. These tracing facilities are a valuable feature of MNF and are superior to the equivalent FTN DEBUG.

0.6 Errors

The errors known to exist in the MNF compiler are listed in Appendix 1. All of these errors are in process of being corrected.

0.7 Limitations

There are 3 major disadvantages of the present MNF compiler on the 7600:-

(1) It does not (yet) have the LEVEL statement so that LCM cannot be accessed. The LEVEL statement will be implemented in 1973, and is similar to the old RUN LARGE statement.

(2) It does not produce relocatable binary object code, and can only generate zero level overlays. An option to generate normal relocatable code will be added in 1973, and full overlay processing will be added at the same time.

(3) It may take a lot of compile core. The minimum core (i.e. just the compiler code) is about 16K decimal (40K octal). To this must be added the tables and code generated during compilation and areas needed for non-COMMON arrays. The actual compile core is (more or less) dependent on the worst routine - some tables being kept between routines. The declaration A(10000) will add 10000 words to the compile core, unless A is COMMON. The minimum compile core needed for a small program is 45000 octal words. See Appendix 11 for further details.

A further limitation is that COMPASS subprograms may not be freely intermixed with MNF FORTRAN routines. Such subprograms must be assembled by means of an explicit COMPASS call. Examples are given in Appendix 2.

1.0 How to run a simple MNF Job

Since MNF accepts most RUN/FTN syntax there should be no need to change your program. This should be true even if you use strange CDC statements like:

```
        IF(EOF,5)1,2
or      L=SHIFT(I,6).A.77B
```

However all declaration statements (type, COMMON, DIMENSION, EQUIVALENCE, EXTERNAL, and DATA) must be placed before the first executable statement.

The control card for MNF is (more or less) like that of FTN or RUN, but the B option is different.

The simple control card 'MNF.' will read the program from the INPUT file, compile it, and print a source listing with error messages and cross reference map on the OUTPUT file. Then MNF will load the program and execute it. So there is no need for an LGO. card in this simple case. (See Appendix 2).

If you like the cautionary and non ANSI usage error messages then you can obtain them by using

MNF(E=0)

If you want to start execution of the program even if it contains 'fatal' source errors (like I+1=2) then use

MNF(D)

If you want an object listing use

MNF(O)

and, if you want execution time tracing (like FTN DEBUG) use

MNF(T)

This will slow down execution of the program by a factor of up to 2 (usually a factor of 1.5) - but will provide excellent debugging information including counts of the number of times that statements are executed, and information on illegal subscripting, incorrect FORMAT conversions, etc., etc. The MNF trace facilities are both cheaper and better than current versions of FTN DEBUG.

SIMPLEST JOB:

```
JOBCARD
MNF.
EOR
   MNF program
EOR
   Data (if any)
EOF
```

N.B.
These examples assume that MNF has been installed in the system. Current control cards to use are given in Appendix 2.
N.B.

HARDER JOB:

Source program (I) is on COMPILE file, compiler output (L) is to go to file TAPE1. Tracing required (T), all possible error messages required (E=0), object listing required (O), and execute even if fatal errors in source program (D), full cross reference map required (R=7)

```
JOBCARD
REQUEST(OPL)                          [say]
UPDATE(P=OPL)                         [output on COMPILE file]
REQUEST(TAPE1)                        [say]
MNF(DOT=TAPE1, I=COMPILE,E=0,R=7)
EOR
   UPDATE cards
EOR
   MNF program
EOR
   Data (if any)
EOF
```

1.1 The MNF Control Card

The MNF control card options are briefly described below. A full description is given in Appendix A of the MNF Reference Manual (attached).

B=file — B- Binary output file (must be loaded first - see below). Default is B=LGO (ie.- execute) (MNF cannot punch binary cards yet, nor produce relocatable binary). This option is NOT correctly described in Appendix 6.

C — C- Call FTN library routines and compile FTN style calling sequence (in A1) rather than RUN style (in B1). The MNF coded I/O library is used - but all other library routines are taken from FTN. Default is not C.

D — D- Debug - compile all statements with 'C$' in c.c. 1 and 2 and execute even if there are fatal source errors. MNF does not accept FTN DEBUG statements (C$ or not). Execution terminates when the first fatal error is "executed". Default is not D.

E=number (0-5) — E- Error message level:

1 = COMMENT	(e.g. "CONTINUE not DO terminator")
2 = NOTE	(non-standard (ANSI) FORTRAN used)
3 = CAUTION	(e.g. 'Is A=A correct' or redundant statement number). May be rather serious errors.
4 = WARNING	(e.g. argument type in subprogram is difficult from the call). MNF checks syntax across routines. Warnings are usually serious errors.
5 = fatal	(e.g. I+1=2). These are numbered 'ERROR 0001', etc.
6 = deadly	('Not enough core' or 'Compiler bug'). Usually not enough core for compiling-try putting local arrays into COMMON - or ask for more core.

Default is E=2 (notes and comments not printed).

I=file — I- Input file (source program) for compiler. Default is I=INPUT.

L=file — L- Listing output file for compiler. Default is L=OUTPUT. Should only be used if listing is not to go to OUTPUT (see O, and R), below.

O=file — O- Object listing file (must be the same as L file - in fact O 'turns on 'L, so that LO, and O are the same as O = OUTPUT)
Default is not O.

P=number P- Page (actually line - at present) limit in execution. MNF will abort the job if number is exceeded. Default is P=5000.

R=number R- Reference listing (on the same file as L, or O - usually OUTPUT). A cross reference listing is printed for each routine.)

R=0 No cross reference map is printed
R=1 C.R. map of statement numbers, and variables
R=2 C.R. map of variables and arrays sorted by address.
R=3 Combination of R=1 and R=2
R=5 Same as R=1 but includes unused variables (nulls).
R=7 Complete map is printed. Combination of R=2 and R=5

Default is R=3. This map is much better than those produced by RUN or FTN. See Appendix I of Reference Manual.(attached).

T T- Tracing during execution. Turns on:

TRACE SUBSCRIPTS [out of bounds]
TRACE SUBPROGRAM CALLS [count calls]
TRACE STATEMENT NUMBERS [count execution]
TRACE FORMATIO [check type of list item vs. type in FORMAT]
TRACE DOLOOPING [check for infinite or negative DO iteration]
TRACE TRANSFERS [check for valid Computed and ASSIGNed GOTO's]

This tracing is very useful, but slows down execution. The statements can be controlled (turned on and off) within the program also (see below).
Default is not T. See chapter 10 of Reference Manual (attached).

Other (illegal) options will be detected by MNF, and the program will (probably) be compiled but not start execution - unless a D option has been accepted before the control card error was found.

The MNF card has the usual SCOPE format so that

MNF(options)

can be written as

MNF,options.

1.2 Running MNF with other programs

MNF can be run with COMPASS, or relocatable binary (etc.) routines by using the MNF(B=file) option. The MNF file must always be loaded FIRST, however - so that the MNF 'B' option differs from the B option on the FTN and RUN cards. Note that the B option, as it is at present, does NOT work as described in Appendix 6 (i.e. Appendix A of the Manual), and that COMPASS subprograms must be assembled using an explicit COMPASS call. Examples are given below.

COMPASS WITH MNF

```
                                    ***N.B.***
                                      See Appendix 2 for control cards to
                                      use until MNF is installed in the
                                      system(s).
                                    ***N.B.***
          JOBCARD
          MNF(B=GASBAG)             [MNF binary file on GASBAG]
          COMPASS(B=GASBAG)         [Append COMPASS binary to GASBAG]
          LINK(F=GASBAG,X)          [Execute file GASBAG]
          EOR
           MNF program
          EOR
           COMPASS program
          EOR
           Data (if any)
          EOF
```

Binary with MNF

```
          JOBCARD
          MNF(B=BALLOON)            [Use B=file to stop automatic execution]
          COPYBR(INPUT,BALLOON)     [Put MNF file in first but don't execute]
          LINK(F=BALLOON,X)         [Load binary cards and execute]
          EOR
           MNF program
          EOR
           Binary deck(s)
          EOR
          EOR
           Data (if any)
          EOF
```

NOTE

The above control cards are for the 7600 SCOPE 1.1 system. MNF can also be run on the 6600. See Appendix 2 for details.

Of course, if the MNF program calls the binary (or COMPASS) subprograms then these subprograms must have a calling sequence which is compatible with MNF. The MNF calling sequence is the same as RUN/FUN, unless MNF(C) is used - in which case the FTN calling sequence is assumed.

Note that MNF(B=LGO) will automatically load and execute. In MNF, LGO really <u>does</u> mean LOAD and GO. Also, the MNF binary output must be loaded FIRST. Otherwise it behaves like RUN and FTN.

APPENDIX 1

Compatibility and Bugs

Compatibility Problems with RUN76 and FTNX

There are some major restrictions with the PSR9 version of MNF. These will be removed during the course of the next 6 months or so:

A. OVERLAYs are not accepted by PSR9 - and fully relocatable binary code is not produced.

B. The ECS (FTNX), and LARGE (RUN76) statements are not available, so that there is no way to access LCM.

C. Embedded COMPASS subprograms are not accepted (MNF recognizes the IDENT card, treats it as a fatal error, and then skips to the next END card). COMPASS subprograms must be assembled by means of an explicit call to COMPASS. This in itself is trivial, but may require re-ordering of the source deck. Note that on RUN76 and FTNX compilation will be considerably more efficient if COMPASS routines are placed all together at the end of the FORTRAN programs. Intermixing COMPASS and FORTRAN routines causes the COMPASS and RUN76 (XFTNX) compilers to be swapped in and out from disc, which is an expensive operation. If the deck is on cards, then the re-ordering is trivial, and if the deck is an UPDATE OLDPL, then the * MOVE directive may be used to advantage. *MOVE is described in the BKY UPDATE reference Manual Page 2-17. The format is simple: *MOVE, $deckname_1$, $deckname_2$ which places the deck whose *DECK name is $name_1$ after the deck whose *DECK name is $name_2$.

D. Compile core requirement will usually approximate to that needed by RUN76, but if there are large local arrays (i.e. arrays which are not in blank or labelled COMMON) then MNF will set up those arrays at compile time and thus need more compile core than the same job on RUN76. Most jobs can overcome this problem by putting large arrays into COMMON. Core usage is further described in Appendix 11.

Since MNF is more than twice as fast as RUN76 in compilation the cost of compilation is less, even when more core is used.

Some further, less serious, incompatibilities are listed below.

(1) The EOF (endfile) test is different on all 3 compilers:

Statement Form	Compiler		
	MNF	FTNX	RUN76
IF(EOF,n)a,b	yes	no	yes
IF(ENDFILE,n)a,b	no	no	yes
IF(EOF(n))a,b	no	yes	no
IF(EOF(n).NE.0)GOTOa	yes	yes	no
IF(EOF(n).NE.0)a,b	yes	yes	no
IF(EOF(n))GOTOa	no	no	no

From this we draw the following conclusions:

The IF(ENDFILE,n) and IF(EOF(n))a,b forms should never be used even on RUN76 and FTNX.

The recommended forms are:

IF(EOF,n)a,b - RUN76 compatible; and IF(EOF(n).NE.0)GOTOa - FTNX compatible; or IF(EOF(n).NE.0)a,b - FTNX compatible. Note that EOF(n) is a type REAL function in both MNF and FTNX - which means that the form IF(EOF(n))a,b is a two branch arithmetic IF statement. This is a strange statement form - unique to CDC FORTRAN, and is flagged as an error by MNF. Ideally, the EOF function should be type LOGICAL since it has a true/false result, but FTNX does it the wrong way and MNF has tried to be compatible with FTN and RUN. Note that the declaration LOGICAL EOF will probably cause wrong results to be obtained on all 3 compilers. A further imbecility of this feature is that the function called EOF(n) will actually reference the library function (routine) called IFENDF, so that a user desiring his own routine EOF must insert the statement EXTERNAL EOF in the calling program, on all three compilers. This is the worst implemented feature of any of the CDC FORTRANs, and can be avoided by MNF users by means of the more elegant statement

READ(.END.=a,u,fmt) list; which is fully described in Appendix 4.

(2) Specification statements (type, DIMENSION, COMMON, EQUIVALENCE, DATA, EXTERNAL, and statement functions) must all appear before the first executable statement in MNF. The order of the specification statements themselves is free (statement functions must be last) but the following order is recommended since it is compatible

with FTNX (which has stricter rules than MNF) and with standard ANSI FORTRAN:

(1) type (e.g. REAL) and DIMENSION
(2) EXTERNAL
(3) COMMON
(4) EQUIVALENCE
(5) DATA
(6) statement functions
(7) the first executable statement.

FORMAT and NAMELIST statements may appear anywhere.

(3) FORTRAN II statements are not accepted by MNF nor FTNX, but are (to some extent) allowed in RUN76. In particular, the following statement forms are not allowed in MNF nor FTN:

WRITE OUTPUT TAPE
READ INPUT TAPE
FORTRAN II SUBROUTINE (etc.)
FORTRAN IV SUBROUTINE (etc.)
ASCENTF SUBROUTINE (etc.)

It is very strongly recommended that such statements be removed from any programs that still use them, since they have no chance of working on other computers - they will only work on the CDC RUN compilers, and are not maintained even on RUN. The header statements (FORTRAN IV SUBROUTINE, etc.) have no meaning and the words before SUBROUTINE (or FUNCTION) should be removed. An exception to this is ASCENTF SUBROUTINE - and such subroutines must be re-written in COMPASS since ASCENTF is no longer available at BKY, not even as part of RUN.

(4) The FORTRAN II library functions - i.e. with names ending in 'F' are not available in MNF nor FTNX (LOCF is available). If your programs contain a few references to such functions, then the 'F's should be removed - and remove the 'X' from the beginning if it is an integer function e.g.

```
Change ABSF to ABS
  "    COSF  " COS
  "    SINF  " SIN
  "    XINTF " INT
    and so on
```

If there are many references to (say) COSF then the following routine could be added to the source deck:

```
FUNCTION COSF(X)
COSF=COS(X)
RETURN
END
```

The deck will then run correctly on all BKY FORTRAN compilers.

(5) The 0 nnn form of octal (Boolean) constants is not allowed on MNF nor FTNX. These should be changed to the nnnB form - which will also work on RUN76. Note the fatuity of the 0 form in the statement

```
I = 0777
```

which will be taken by all 3 compilers (RUN76 included) to refer to the variable of name 0777 rather than the constant 777B.

(6) MNF and FTNX evaluate mixed mode expressions (i.e. mixtures of REAL and INTEGER values) differently from RUN76. RUN evaluates the whole expression in the mode of the dominant operand, whereas MNF and FTN evaluate each term in the expression separately and then convert the result to the dominant type. This makes a difference with expressions such as

```
A = R +I/J  with  I = 1$J = 2$R = 0.5
```

RUN76 gives

```
A = R + FLOAT(I)/FLOAT(J) = 0.5 + 1.0/2.0 = 1.0
```

MNF and FTN give

```
A = R + FLOAT(I/J) = 0.5 + FLOAT (1/2) = 0.5 + 0.0 = 0.5
```

The main problem arises with integer divisions, as above - but loss of accuracy can also result when real and double precision values are mixed indiscriminately. When in doubt about the way a mixed mode expression will be evaluated, it is best to write the expression explicitly - as in

```
A = R + FLOAT(I)/J
```

In order to alleviate any compatibility problems with RUN76, MNF prints a message for each statement where mixed mode arithmetic has been used, it is:

CAUTION CHANGE OF TYPE IN STATEMENT - DOMINANT TYPE USED.

(7) A further difference between MNF and the CDC compilers is that MNF generates rounded hardware instructions for real (i.e. floating point) expressions, whereas FTNX and RUN76 generate unrounded instructions. It is well established that rounding produces more accurate results than does unrounded arithmetic, and both sets of instructions run at the same speed. Most programs are not affected by rounding in that the printed results are exactly the same. However, where results differ significantly between RUN/FTN and MNF (and the difference is due to rounding) it is likely that the algorithm used in the program is incorrect - in that undue reliance is placed upon the accuracy of the machines. Dubious results will often arise through taking the difference of two nearly equal numbers and then using that difference in further computations.

(8) The FTNX 'debug' statements are not syntactically the same as the MNF tracing statements. The MNF traces will do all that the FTN

statements will do - and more - but the statement form is different. It is recommended that MNF tracing statements are punched with a C$ in columns 1 and 2, and that the MNF(D) option is used to 'turn on' the C$ statements, since if this is done the program may be run without change on other compilers - the C$ statements then being taken as comments. It will be noted that most of the tracing statements are activated simply by using the MNF(T) option.

The use of MNF tracing is very strongly encouraged since it greatly reduces debugging time at small cost. These statements are fully described in Appendix 5. The source listings printed by MNF are also well designed for debugging purposes. Users are urged to read the listings and error messages before seeking further advice, since the aim of MNF is to present error messages which are intelligible to the normal user.

Changing your programs to conform to the ideas in the above section will make them easier to transfer to other machines and to other compilers. In addition, CDC cannot be expected to maintain the archaic features of RUN for ever, and so, if my suggestions are followed, your programs will be less at the mercy of any changes made in the operating systems on the BKY machines. The NOTE level messages printed by MNF(E=Ø) will ease conversion by drawing attention to all possible problem areas. Apart from obvious exceptions - such as the control card descriptions - the various RUN and FTN reference manuals can also be taken to apply to MNF. With the exceptions described above, MNF is "upwards compatible" with FTNX and RUN76 in that RUN and FTN programs will work on MNF. Of course, the advanced features of MNF - such as free format I/O and

statements like PRINT, (SIN(X), X = 0,3.142,3.142/180) will not work on FTN nor RUN.

The writer would appreciate comments on the MNF compiler and its library, and descriptions of any bugs and incompatibilities found which are not already mentioned here. Enquiries should be addressed to C. F. Schofield, Room 1127 - Building 50A (Telephone extension 5118).

Errors in the February 1973 PSR9 Version of MNF

The first object was to get MNF working on the 7600, and although the compiler and library is available on the 6600, there are some serious restrictions upon the I/O operations available on that computer. Below are listed the known errors in the compiler and library as at 10 March 1973. The errors are listed in approximate order of importance. It will be seen that most of the errors are of a trivial nature, and it is hoped that MNF is about as reliable as RUN76 - and probably more reliable than FTNX. Of course, questions of reliability will only be determined when MNF has a large number of users. It is, however, certain that bug lists for RUN 76 and for FTNX, were they to be published, would be quite as long as the list below.

(1) The only I/O statements handled by the 6600 MNF library are the formatted (coded) READ,WRITE,PRINT, and PUNCH statements. The following statements will cause a field length (MODE 1) error in the BKY I/O routine QFNDFET:

READ(u) and WRITE(u) - Binary I/O.

ENDFILE, REWIND and BACKSPACE - Coded and binary.

Note that all of these statements work on the 7600. The 6600 restriction is due to the non standard FORTRAN library used on the BKY 6000 system.

(2) MNF(C) - i.e. the FTN calling sequence mode - has been relatively little used in the past, and probably contains more errors than the other MNF modes. However, the compiler and libraries have been arranged to accept MNF(C), and any errors not described here should be reported to the undersigned.

(2A) There is an error in the MNF(C) library (both machines) which causes wrong results unless all buffer sizes on the PROGRAM statement are set to at least 2020B. E.g.

PROGRAMME (INPUT=2020B, OUTPUT=2020B) Note that 2020B is the default buffer size on the 6600 but the default size on the 7600 is 422B. So 6600 jobs will run without a declared buffer size - but 7600 jobs will not. The error only occurs if the C parameter is used.

(3) MNF(CT) - i.e. FTN calling sequence and tracing (debug) turned on produces false fatal error messages for the following sequence:
COMMON A (100) $ A(J) = exp$ X = SIN(A(J)) $ A(J) = exp ← fatal error here.

(4) The PROGRAM statement must be present. If it is omitted the loader prints the fatal message - NO MAIN PROGRAM. (both 6600 and 7600).

(5) ATAN (0.0) produces an indefinite result. This error should be fixed soon. In fact several of the errors described here may have been corrected by the time this document is printed.

(6) A mode error occurs in execution if the character length of an ENCODE or DECODE statement is outside the range 1.LE.n.LE.150.

(7) TRACE FORMATIO (also turned on by MNF(T)) gives a correct error message, but the octal address of the list element is wrong.

(8) A variable with the same name as an intrinsic function is given the same type as the function. E.g. CMPLX is assumed to be of type complex unless declared otherwise.

(9) The statement following a non-recognized (illegal) statement is lost during compilation.

(10) Sense switches set up (by control cards) prior to compilation are destroyed by the compilation process.

(11) The name of an ASSIGNed variable which occurs in a TRACE statement is sometimes printed incorrectly. The printed message is otherwise correct.

(12) The following input causes MNF to stop in compilation:

(a) some FTN debug statements - e.g. C$ FUNCS

(b) illegal BUFFER statements - e.g. BUFFER IN (1,1) (1.0,2.0)

(c) a statement function as the <u>first</u> card in a deck - e.g.

SF(A,B) = A*B/(A + B)

(13) An error occurs if a program having more than 15 actual (i.e. not equivalenced) files is loaded.

(14) The constant 0.0E0 is not set up correctly at compile time.

(15) Parenthesized expressions in logical IF statements occasionally cause problems. E.g. IF((A.LT.B).OR.B.LT.C.OR.C.GT.D) GO to 5 is compiled wrongly. Note that the parentheses are redundant.

(16) The compiler tries to remember constants over several statements, and this rememberance is false when there is an intervening ASSIGN statement. E.g. the sequence

```
J = K+4
ASSIGN 20 to N
L = I+4
```

is compiled wrongly.

(17) Statements for which fatal compile errors are detected do not always generate code to jump to the ABORT routine if the statement is 'executed' by means of MNF(D).

(18) The sequence number (i.e. source line number) in a TRACE message generated from an input list is not correctly printed. E.g.

```
3*   TRACE B
4*   READ 5, B
```

The sequence number printed in the message below is not actually correct.

TRACE STORE INTO REAL VARIABLE B AT SEQUENCE NUMBER 4* VALUE = 1.0

(19) In certain cases, legal EQUIVALENCE statements get a spurious error message, although the statement is correctly compiled. E.g. the following sequence produces spurious messages:

```
COMMON A,B
REAL C(2)
EQUIVALENCE (A,C(1)), (B,C(2))
```

Note that the second equivalence is redundant in this case.

(20) A greatly abbreviated cautionary message is printed when a COMMON block is defined which has a different length than in a previous definition. E.g. the statement COMMON/DISMAL/B(1) may produce the message CAUTION - DISMAL.

If any errors are found which are not described above, they should be reported to Clive Schofield - Room 1127 - Building 50A (X5118).

Appendix 2

Control Cards Required

MNF has been installed in the 7600 system with the library as a system library file (named MNFLIB). At present the compiler is available on the 6600 as a PSS (datacell) file (library MNF, subset MNF), and the library as another such file (library MNF, subset LIB66).

The identical compiler is used on the 6600 and the 7600 (this is like RUN76) - and the MNFLIB and LIB66 libraries are basically the same, with LIB66 containing four extra routines which are available on the 7600 system, but not on the BKY 6000 system.

7600

MNFLIB (7600) contains:

```
INPUT$    (replaces INPUTC and KRAKER)
OUTPUT$   (    "     OUTPTC  "  KODER)
INOUT$
ALNLOG    (ALOG,ALOG10)
ASINCOS   (ASIN,ACOS)
ATAN
ATAN2
EXP
RUNSYS    (The 7600 RUNSYS plus mods  for MNF tracing)
SINCOS    (SIN,COS)
SQRT
SYSTEM    (The same as the 7600 RUN SYSTEM)
SYSTEM$   (The 7600 FTNX SYSTEM$ plus mods for MNF tracing.
           Used only by MNF(C) jobs).
TANH
and TRACE.   and ABORT$ (called by MNF(T) jobs)
```

All other routines needed during the execution of an MNF job (e.g. those for binary I/O) are taken from the 7600 RUN library or the FTNX library if MNF(C) is used.

The control cards needed for a simple MNF job on the 7600 are:

```
(1) JOBCARD
    MNF(B=BIN, other options if any)
    LINK(F=BIN,P=MNFLIB,X)
    (eor)
      MNF program
    (eor)
      Data (if any)
    (eof)
```

Note that the file name BIN is arbitrary, but it must not be LGO.

7600 MNF with COMPASS subprograms

```
(2) JOBCARD
    MNF(B=BIN,etc.)
    COMPASS(B=BIN)
    LINK(F=BIN,P=MNFLIB,X)
    (EOR)
      MNF program
    (EOR)
      COMPASS subprograms
    (EOR)
      Data (if any)
    (EOF)
```

Note that the MNF program must be loaded first (it must be loaded into RA+101B), and that COMPASS subprograms cannot be simply intermixed with MNF programs - they must be separately assembled using COMPASS.

7600 MNF with UPDATE and COMPASS subprograms

```
(3) JOBCARD
    REQUEST(OLDPL,etc)
    REWIND(OLDPL)
    UPDATE(options)
    RETURN(OLDPL)
    MNF(I=COMPILE,B=BIN)
    COMPASS(I=COMPILE,B=BIN)
    LINK(F=BIN,P=MNFLIB,X)
    (EOR)
       UPDATE cards for MNF and COMPASS decks.
    (EOR)
       Data (if any)
    (EOF)
```

6600

As mentioned above, MNF is not yet installed in the 6000 system, but is available as two PSS files. The library file (MNF subset LIB66) is identical to the 7600 library, MNFLIB, with the following four routines added:

SIO$	(used only by SYSTEM$ on MNF(C) jobs)
GETBA	(used by all jobs)
CPUSYS	(used by all jobs)
CPUCIO	(used by all jobs with I/O statements)

These routines are not in the "standard" BKY 6000 system library, but are in the 7600 system.

Any routines not present in LIB66 that are needed by an MNF job during execution are taken from the RUNF library - or the FTN library if MNF(C) is used.

The control cards necessary to run a simple MNF job on the 6600 are:

(4)
```
JOBCARD
*6600
LIBCOPY(MNF,MNF,MNF)
MNF(B=BIN, other options if any)
LIBCOPY(MNF,LIB,LIB66)
LODE(I=BIN,L=LIB)
XEQ.
(EOR)
  MNF program
(EOR)
  Data (if any)
(EOF)
```

6600 MNF with UPDATE, and COMPASS subprograms

(5)
```
JOBCARD
*6600
REQUEST(OLDPL,etc)
REWIND(OLDPL)
UPDATE(options)
RETURN(OLDPL)
LIBCOPY(MNF,MNF,MNF)
MNF(I=COMPILE,B=BIN)
COMPASS(I=COMPILE,B=BIN)
LIBCOPY(MNF,LIB,LIB66)
LODE(I=BIN,L=LIB)
XEQ.
(EOR)
  UPDATE cards for MNF and COMPASS decks
(EOR)
  Data (if any)
(EOF)
```

The file name BIN is arbitrary, but it must not be LGO.

Appendix 3

The MNF Library

The MNF library contains 5 routines used for coded I/O. These are: INPUT$,OUTPUT$,INOUT$,CPUCIO, and CPUSYS. These routines are faster by a factor of 1.5 to 2 then the equivalent RUN library routines, and are further described below.

When tracing - MNF(T) - is used, two further MNF library routines are called to print the trace output:

	TRACE.	(765B)
and	ABORT$	(5B)

ABORT$ is also called by jobs with fatal source errors which are forced into execution by means of MNF(D). Finally, there are 11 mathematical functions:

ALNLOG(ALOG,ALOG10)	(107B)
ASINCOS(ASIN,ACOS)	(115B)
ATAN	(63B)
ATAN2	(74B)
SINCOS(SIN,COS)	(72B)
TANH	(63B)
EXP	(54B)
SQRT	(43B)

(Other functions are taken from the RUN76 (i.e. the RUNF) library).

These functions have been tested at BKY with the following results on the 6600.

FUNCTION on 6600	Compiler							
	MNF		RUN(OLD)		RUN76 (and RUNF)		FTN (not T) (and FTNX)	
	Time	Error	Time	Error	Time	Error	Time	Error
SIN	21	10	25	60	30	8	26	60
COS	22	10	25	35	29	8	25	35
EXP	18	10	27	5	29	5	15	25
ATAN	22	8	29	10	31	10	29	10
TANH	21	8	45	500	49	70	31	50
SQRT	15	4	22	4	16	10	22	4
ALOG	21	5	31	5	24	5	26	6
ALOG10	26	5	31	5	26	5	27	8
ASIN	26	10	58	6000	58	6000	54	6000
ACOS	26	8	60	2000	61	2000	57	2000
ATAN2	25	10	44	12	44	--	29	--

Times are in micro seconds per call on a 6600. Errors are in units of 10**(-15) - an optimum error being about 4 units. All figures are approximate.

These functions and their type complex equivalents have also been tested on the 7600. The errors were nearly the same as on the 6600, and the times are given below. The MNF functions are optimised for the 6600 at present, but the 7600 results are still good.

Function on 7600	Compiler					
	MNF		RUN76		FTNX (not T)	
	Simple	Complex (6600)	Single	Complex (6600)	Single	Complex (6600)
SIN	5.3	19(94)	6.1	19(122)	6.3	16(76)
COS	5.8	19(93)	5.8	18(123)	6.0	16(78)
EXP	5.0	17(84)	4.9	17(113)	4.2	13(64)
ATAN	5.5		7.0		7.2	
TANH	5.1		8.2		7.1	
SQRT	4.8	12(73)	4.4	13(74)	4.4	12(69)
ALOG	5.4	21(93)	6.2	21(120)	6.4	22(108)
ALOG10	7.0		6.6		6.6	
ASIN	6.8		10.3		10.3	
ACOS	7.1		11.0		11.1	
ATAN2	-	-	-	-	-	-

Note that FTN does no checking of argument values - this is only done if FTN(T) is used which slows down the FTN results by at least 20%. RUN and MNF both check arguments properly. A new, faster, version of MNF EXP will shortly be available.

It is clear that the MNF functions are superior to the various RUN and FTN functions. We intend to replace the RUN functions with the MNF ones. The functions assume a RUN style calling sequence and MNF(c) causes the standard FTN functions to be called. However the MNF functions could be modified to accept an FTN calling sequence and so replace the FTN library also. This would confer the benefits of compatibility in addition to greater accuracy and speed.

The MNF coded I/O library requires (at the moment) a reasonably SCOPE-compatible version of SYSTEM (i.e. Q8NTRY, etc.) so the 7600 RUN 1.0 routines RUNSYS and SYSTEM are used by MNF and MNF(c) on both 6600 and 7600.

One nuiscance is that, when MNF(c) is used both RUNSYS (and SYSTEM) and SYSTEM$ are called, and SYSTEM$ also calls SIO$. The compiled code calls SYSTEM$ (END., etc.), and the I/O routines call RUNSYS and SYSTEM. I have added SYSTEM$ from 7600 FTNX to the MNF library so that MNF(c) jobs will run on the 6600. What with RUNSYS, SYSTEM, SYSTEM$, and SIO$, MNF(c) jobs will use more execution core than on FTNX.

Summary of Core used by null program

6600			7600			
RUN76 (BKY I/O)	MNF	MNF(C)	FTN	MNF(C)	MNF	RUN76 (CDC I/O)
Q8NTRY(47) END(30) QTRACEB(10) SYSTEM(126) QFILEND(12) REMARK(13) /QJOBSFL/(2) /QASAFLG/(1) /QLINLMT/(1) /QFILEAD/(1)	RUNSYS (450) SYSTEM (316)	RUNSYS (450) SYSTEM (316) SYSTEM$ (1122) SIO$ (1255) REMARK (13)	SYSTEM$ (1011) S10$ (1276) REMARK$ (26)	RUNSYS (450) SYSTEM (316) SYSTEM$ (1122) S10$ (1276) REMARK$ (26)	RUNSYS (450) SYSTEM (316)	RUNSYS (303) SYSTEM (316)
267B	766B	3400B	2335B	3434B	766B	621B

Summary of core used by READ, PRINT routines.

The MNF library contains 5 routines used for coded I/O. These are listed below

RUN76(6600) (BKY I/O)	RUN76(7600)	MNF(6600, 7600)	FTNX (7600)
OUTPTC 70 KODER 1241	OUTPTC 50 KODER 1235	OUTPUT$ 1120	OUTPTC$ 74 KODER$ 1376
INPUTC 116 KRAKER 734	INPUTC 56 KRAKER 1026	INPUT$ 1222	INPUTC$ 116 KRAKER$ 1471
QREADST 46 QWRITES 42 QFNDFET 45 SYS$ 21	RUNIOP 323 CPUSYS 53 CPUCIO 506	INOUT$ 202 CPUSYS 53 CPUCIO 506 (CPUSYS is 34 on 6600) (CPUCIO is 273 on 6600)	
/QBCDBUF/231 /$IOSC/ 30		/CIOBUF/ 425 /FREEFOR/ 3 GETBA 17	GETBA 17
Total Length = 3062B	Total length = 3515B	Total length = 3774B	Total length = 3320B (4616B with S10$)

Summary of core used by READ 1, X $ PRINT 1, X $ END

	6600			7600			
	RUN76 (BKY I/O)	MNF.	MNF(c)	RUN76 (CDC I/O)	MNF.	MNF(c)	FTNX.
INPUT and OUTPUT default buffers	4044	4040	4040	1044	1044	1044	1044
SYSTEM Q8NTRY etc.	267	766	3400	621	766	3434	2335 (+S10$)
CODED I/O routines	3062	3542	3542	3515	3774	3773	3320 (-S10$)
TOTAL	7415B	10570B	13202B	5402B	6026B	10474B	6721B

APPENDIX 4

"NOTE: In this Appendix the character ' is the BKY character ≠ and 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 will no longer use the CDC routines INPUTC, OUTPUTC, KODER and KRAKER. This change will be 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 <u>not</u> with

RUN, FUN or FTN. These statements will (probably) not work on other machines, or compilers.

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 mangetic tape is to be processed.

```
JOBCARD                                  **NB**
REQUEST(FRED) *TAPE REQUEST*               See Appendix 2 for present
MNF(D,T)                                   control cards
RETURN(FRED)                             **NB**
7/8/9

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:

```
JOBCARD
REQUEST(TAPE5)*TAPE REQUEST*
MNF(D,T)
RETURN(TAPE5)
7/8/9

PROGRAM TEST(TAPE5,OUTPUT,TAPE6=OUTPUT)
 .
 .
 .
REWIND 5
READ (5,1000)A
 .
 .
 .
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.

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 or asterisks, 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  3.0000000000
```

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 the MNF 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 ' (Note: The character ' is the same as ≠)
r = signed or unsigned integer constant.
j = column position: unsigned integer constant.
k = Hollerith character count: unsigned integer constant.

Legal format specifications

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

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 accomodate 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 5.

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          the IF(EOF,n) statement
      NUMBER=NUMBER+1                     also works
      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:

```
      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 (*ØNUM=*,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 37773777377737773777B. 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:

```
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:

```
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.

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)=10HTHIS 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:

Integer	I20
Real	G20.13
Double Precision	G40.26
Complex	2G20.13
Logical	L10
Alphanumeric	A10

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=                    25R=                    .2ØØØØØØØØØØØØE+15
CX=     47.6ØØØØØØØØØØ          67.4ØØØØØØØØØØ
DX=             .55Ø12345678901234567890 1D-1Ø
```

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 active:

```
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 136. The upper limit is 150 characters and the lower limit is that just large enough to accomodate 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 <u>characters</u> are being output if W >10, A10 format will be used and the rest of the field filled with blanks.

The formats of section 6.2 may be given now more generally as:

Integer	I(2W)
Real	G(2W).(D)
Double	D(4W).(2D)
Complex	2G(2W).(D)
Logical	L(W)
Alphanumeric	A(W)

It can be seen that the default values of W and D are W=10 and D=13.

7. Example Program

```
JOBCARD
MNF(D,T)
7/8/9
            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      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      PRINT MESSAGE FOR CARD IN ERROR
  400       PRINT,/,'*** CARD IS IN ERROR ***'
            GOTO 200
 C      EXIT TO HERE WHEN END OF FILE SENSED ON INPUT
  600       PRINT ,/,/,*END OF OUTPUT*
            STOP
            END
7/8/9
*KITCHEN* 12,8.6
*LIVING ROOM*,15.75,12.3
  'BATHROOM',9.4A,7.9
  'BATHROOM',9.43,7.9
6/7/8/9
```

Output starting on a new page:

```
THIS IS A TITLE ON A NEW PAGE

AREA OF KITCHEN IS              103.200000000      SQ.FT.

AREA OF LIVING ROOM          IS          193.725000000    SQ.FT.

*** CARD IS IN ERROR ***

AREA OF BATHROOM IS             74.4970000000      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 MNF Reference Manual, and in the RUN and FTN Reference Manuals.

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.

Error Number	Message and Cause
64	ERROR DATA INPUT *** UNDEFINED FILE NAME: A READ was attempted on a file not defined on the PROGRAM card.
65	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.
66	DECODE ERROR ***** CHARACTER COUNT EXCEEDS 150: Bad decode statement.
68	ILLEGAL FUNCTIONAL LETTER: An illegal conversion letter occurred in an output format.
69	IMPROPER PARENTHESES NESTING: Parentheses do not match in an output format or are nested deeper than two levels i.e. deeper than FORMAT((( ))).
70	RECORD SPECIFICATION EXCEEDS 150 CHARACTERS: Tried to output more than 150 characters in one line.
71	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.
73	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.
or	RECURSIVE CALL TO INIT80: output list item is in TRACE statement.
or	ERROR FREE FORMAT OUTPUT *** COMMON BLOCK/FREEFOR/ PARAMETERS ZERO OR NEGATIVE: Common block parameters are illegal - see section 6.
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.
74	ILLEGAL FUNCTIONAL LETTER: A illegal conversion letter occurred in an input format.
75	IMPROPER PARENTHESES NESTING: Parentheses do not match in an input format or are nested deeper than two levels i.e. deeper than FORMAT ((( ))).

Error Number		Message and Cause
76		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.
77		RECORD SPECIFICATION EXCEEDS 150 CHARACTERS: Tried to input a record more than 150 characters long.
78		ERROR DATA INPUT *** ILLEGAL DATA IN FIELD: Input data characters do not match format. Usually caused by mispunched characters.
79		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.
80		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:
	or	RECURSIVE CALL TO INIT81: input list item is in TRACE statement.
	or	ERROR FREE FORMAT ALPHANUMERIC INPUT *** ILLEGAL OR MISSING DELIMITER: Alphanumeric input string too long or terminated illegally - see section 5.
	or	ERROR FREE FORMAT ALPHANUMERIC INPUT *** ARRAY OR VARIABLE NOT TYPE INTEGER: Alphanumeric data may only be read into integer variables.
83		ERROR DATA OUTPUT *** UNDEFINED FILE NAME: A WRITE was attempted on a file not defined on the PROGRAM card.
84		ERROR DATA OUTPUT *** OUTPUT FILE LINE LIMIT EXCEEDED: tried to print too many lines of output.
85		ENCODE ERROR *** CHARACTER COUNT EXCEEDS 150: bad encode statement.
88		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.

APPx 5

Debugging and List Control Statements 10

The statements described in this chapter are all non-standard and their usage 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, to control the listing of the source cards, and to give additional information about the source program.

It is recommended that the D option of the MNF control card and C$ in columns 1 and 2 be used with the statements described in this chapter 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.

10.1 TRACE AND NO TRACE 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 10.1.9. Several of the tracing forms are automatically turned on by the T option of the MNF control card (Appendix A) and begin tracing when the first statement is encountered.

There are two kinds of TRACE statements, frequency and debugging. The frequency statements give a usage count after the program has finished execution 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 (see Appendix J). 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 by 10 to 50% 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 errors occur which abort the program (except time limit, page limit, node exit, PP-detected errors, or operator drop).

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, up to a 300% increase in extreme cases, and will usually increase printed output sometimes to the point of page limit if many errors are detected. For example:

```
      TRACE SUBSCRIPTS
      COMMON A(10),B(10),C(10000)
      DO 2 I=1,10000
      A(I)=0.0
    2 B(I)=0.0
```

could produce 19980 error messages, the number of times the subscript I was greater than 10. 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.

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 statement number labels on them, but if they do the labels will be ignored and they will not be entered into the list of statement numbers.

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 (see Appendix H) 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 10.1.8 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. Certain misspellings in TRACE statements are corrected automatically by the compiler. Except for SUBPROGRAM CALLS and SUBPROGRAM ENTRY, if more than 7 alphanumeric characters appear after TRACE or NO TRACE and the first 5 agree with one of the forms below, any errors in the remaining characters are ignored and the correct statement is assumed.

The frequency TRACE and NO TRACE statements forms are:

(1) TRACE STATEMENT NUMBERS
NO TRACE STATEMENT NUMBERS

(2) TRACE SUBPROGRAM CALLS
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
NO TRACE DOLOOPING

(6) TRACE FORMATIO
NO TRACE FORMATIO

(7) TRACE SUBSCRIPTS
NO TRACE SUBSCRIPTS

(8) TRACE TRANSFERS
NO TRACE TRANSFERS

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. 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.

(9) TRACE list
NO TRACE list

The following statements complement the T option on the MNF control card (Appendix A). 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.

(10) TRACE

NO TRACE

10.1.1 TRACE STATEMENT NUMBERS

NO TRACE STATEMENT NUMBERS

When TRACE STATEMENT NUMBERS is used, a count is kept of the number of times each statement number was executed and the number of times each format number was used in each program and subprogram. The count is set up when the statement 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.)

10.1.2 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.

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.

10.1.3 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 SUBPROGRAMS 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 INF-compiled subprograms. Another example would be a system 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.

10.1.4 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 mode exits 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 mode exit 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 messag 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.

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

10.1.5 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 number of the statement in error), after which control is returned to the DO or FOR loop and the loop is executed as is. 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 131072 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 DOLOOPING 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 execution there will not be any TRACE messages for indexing errors involving only constants.

10.1.6 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 (since only the first 10 characters of such a two-word variable can be read or written). 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.

10.1.7 TRACE SUBSCRIPTS
NO TRACE SUBSCRIPTS

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. 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 or 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
```

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 10.1.9).

10.1.8 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. 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 values 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.

10.1.9 TRACE list

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

Simple variable name. Whenever the value of the named variable is 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.

Array names. 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 subscript is checked using the initial and final value of the DO or 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 X(10)
  TRACE X
  DO 1 I=1,11,3
1 X(I)=1.5+FLOAT(I)
```

Subroutine names or subroutine entry names. After each return from the named subroutines of subroutine entries, a message is printed on the OUTPUT file:

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

Function names or function entry names. Whenever the named 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 value and octalB is the octal value of the function or function entry after the return from the function is made.

Simple variable names each 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.

Array names each followed by (Lower Bound, Upper Bound). This is the same as the previous type except that every element of the array is checked and the subscript is also checked.

Function names or function entry names each followed by (Lower Bound, Upper Bound). This is the same as for simple variable names except that the check is made each time the named function or function 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*

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 that the statement number was on.

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 10.1.6). 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, 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. 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 (10.1.1).

10.1.1 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.

10.1.11 TRACE
NO TRACE

TRACE turns on the same tracing that the T option of the MNF card turns on (Appendix A), namely TRACE DOLOOPING, TRACE FORMATIO, TRACE STATEMENT NUMBERS, TRACE SUBPROGRAM CALLS, TRACE SUBSCRIPTS and TRACE TRANSFERS. These remain in effect until the end of the program and subprogram unless turned off individually or in toto or by a NO TRACE statement.

10.1.12
EXAMPLE OF
TRACING The output from the TRACE statements is organized so that a scan of two columns locates the desired information. The first column which is aligned vertically is the essential name, number value or octal address. The other column is the sequence number to locate where the problem is.

A sample program using TRACE facilities is completely reprinted below.

```
          UNIVERSITY OF MINNESOTA 6600 FORTRAN COMPILER
          MNF(U)
                      C  THIS PROGRAM ILLUSTRATES ALL FEATURES OF TRACING.
                      C
                      C  NEXT STATEMENT COMBINES TRACE DO LOOPING, TRACE FORMATIO,
                      C   TRACE STATEMENT NUMBERS, TRACE SUBPROGRAM CALLS, TRACE SUBSCRIPTS,
                      C   AND TRACE TRANSFERS
                      C
  000100B           1* CS     TRACE
SEQ       1* NOTE - TRACE OR NO TRACE IS NON STANDARD        ****************************************
SEQ       1* NOTE - CS STATEMENT IS NON STANDARD   ****************************************
  000100B           2* CS     TRACE ARITHMETIC
                      C
                      C  DECLARATIVE STATEMENTS MUST APPEAR BEFORE INDIVIDUAL TRACING
                      C   STATEMENTS WHICH HAVE BOUNDS TRACING
                      C
SEQ       2* NOTE - TRACE OR NO TRACE IS NON STANDARD        ****************************************
  000100B           3*        COMMON Z(10),ARRAY(10)
  002145B           4*        COMPLEX C,CC
  002145B           5*        DOUBLE PRECISION D,DD
  002145B           6*        LOGICAL L
                      C
                      C  INDIVIDUAL TRACE OF FOLLOWING VARIABLES, SUBPROGRAMS, STATEMENT
                      C   NUMBERS, AND FORMAT NUMBERS.
                      C
SEQ       6* NOTE - CS STATEMENT IS NON STANDARD   ****************************************
  002145B           7* CS     TRACE II(-3,10), RR(6.0,9.0), DD(1.0D0,2.0D0), CC(-10.0,10.0),
                      CS   I(-3,0:13,0)), I, R, D, C, L, 10, 20, 30, 40, JINX, 1000, ARRAY
                      CS   Z(1.5,2.5), 100
                      C
SEQ       7* NOTE - TRACE OR NO TRACE IS NON STANDARD        ****************************************
SEQ       7* CAUTION - ARRAY REPEATED IN TRACE LIST           ****************************************
SEQ       7* NOTE - CS STATEMENT IS NON STANDARD   ****************************************
  002145B           8*        DATA ZERO/0.0/,ZRO/0.0/,ONE/1.0/
                      C  PRINT CHANGE IN VALUE
  002145B           9*        R=10.0
  002172B          10*        L=.TRUE.
  002175B          11*        C=(1.7,6.2)
  002201B          12*        D=3.1415926535897900D0
  002205B          13*        I=13
  002210B          14*        ASSIGN 10 TO I00
                      C  ILLEGAL SUBSCRIPT
  002211B          15*        Z(I)=R**3.5
                      C  EXCEED BOUNDS
  002221B          16*        DD=R
SEQ      16* CAUTION - IS DOUBLE = REAL EXPRESSION CORRECT   ****************************************
  002231B          17*        RR=I
  002240B          18*        II=I-1
  002245B          19*        CC=(-1.0,15.0)
  002260B          20*        ARRAY(3)=RR**2
                      C  FUNCTION REFERENCE (SUBPROGRAM CALL AND CHANGE OF VALUE)
  002267B          21*        II=JINX(I)
```

```
                    C  THANSFER CODE
0023068      22*        5 GO TO (60, (10,20,30)
0023158      23*       10 ASSIGN 20 TO IGO
                    C  FORMAT SPECIFICATION MISMATCH WITH LIST
0023178      24*          WRITE (6,1000) II
0023278      25*     1000 FORMAT (/95H THE LIST OF SUBPROGRAMS FOLLOWED BY EACH STATEMENT NU
                         1MBER AND THE NUMBER OF TIMES EACH IS USED/86H HAS SOME ZERO COUNTS
                         2 WHEN TRACE TRANSFERS IS USED BUT TRACE STATEMENT NUMBERS IS NOT./
                         3 1X,E20.10)
0023278      26*          GO TO 5
0023308      27*       20 ASSIGN 30 TO IGO
                    C  STORE BAD VALUES
0023328      28*          X=ONE/ZERO
0023408      29*          Y=ZRO/ZERO
0023448      30*          J=1534567890206721
SEQ     30* CAUTION - INTEGER CONSTANT > 2**48-1 IN MAGNITUDE      ****************************************
0023508      31*          O=1.0D-285
0023548      32*          GO TO 5
0023568      33*       30 ASSIGN 40 TO IGO
                    C  STOP TRACING CALLS
SEQ     33* NOTE - CS STATEMENT IS NON STANDARD   ****************************************************************
0023608      34* CS       NO TRACE SUBPROGRAM CALLS
SEQ     34* NOTE - TRACE OR NO TRACE IS NON STANDARD      ****************************************************
0023608      35*          II=JINT(I)
                    C  BAD DO OR FOR LOOPS
0023778      36*          K=1
0024028      37*          M=11
0024068      38*          N=-1
0024128      39*          DO 50 JJ=J,K
0024248      40*          DO 50 I=J,2,J
0024408      41*          DO 50 II=M,K
0024528      42*          FOR 50 II=M,M,N
SEQ     42* NOTE - NON STANDARD DO LOOP          ****************************************************************
0024738      43*           GO TO 100
0024748      44*       50 CONTINUE
0025108      45*  100    LL=2
0025158      46*          M=131070
0025218      47*          N=-M
0025258      48*          DO 110 I=1,N,2
0025378      49*          MM=131072
0025438      50*          DO 110 J=M,MM
0025558      51*          FOR 110 KK=1,N,LL
SEQ     51* NOTE - NON STANDARD DO LOOP          ****************************************************************
0025708      52*          GO TO 5
0025728      53*   110     CONTINUE
0026038      54*       40 K=5
                    C  ILLEGAL COMPUTED GO TO (TERMINATES PROGRAM).
0026128      55*          GO TO (10,20,30,40),K
SEQ     55* NOTE - CS STATEMENT IS NON STANDARD   ****************************************************************
0026238      56* CS       NO TRACE ARITHMETIC
SEQ     56* NOTE - TRACE OR NO TRACE IS NON STANDARD      ****************************************************
0026238      57*          END
SEQ     57* COMMENT - 4 WORDS USED IN SETUP OF SUBPROGRAM   ****************************************************
SEQ     57* CAUTION - 40 IN ASSIGN BUT NEVER APPEARED IN ASSIGN GO TO LIST      ****************************
```

```
NUMBER AND NAME CROSS REFERENCE MAP
NUMBER TYPE ADDRESS  SN=STATEMENT NUMBER, FN=FORMAT NUMBER, J=JUMP, R=READ, W=WRITE, L=LOCATED, D=DO END, A= ASSIGN, T=TRACE
    5 SN   0023068 52J 32J 26J 22L
   10 SN   0023158 55J 23L 22A 14A 7T
   20 SN   0023308 55J 27L 23A 22A 7T
   30 SN   0023568 55J 33L 27A 22A 7T
   40 SN   0026048 55J 54L 33A 7T
   50 SN   0024748 44L 42D 41D 40D 39D
  100 SN   0025108 45L 43J
  110 SN   0025728 53L 51D 50D 48D
 1000 FN    LN= 95 25L 24W 7T
 NAME TYPE MAIN USE ADDRESS  BLOCK U=USED, S=STORED, D=DECLARED, I=DO INDEX, A=ASSIGN, P=DO PARAMETER
ARRAY   R ARRAY       0000128 //        20S 3D
BNDTRC, R SYSTEM      0000509           42S 35S 21S 20U 19U 18S 17U 16U
C       C VARIABLE    0021578           12S 7D 4D
CALTRC, R SYSTEM      0000709           35U 21U
CC      C VARIABLE    0021658           19S 4D
D       D VARIABLE    0021618           22S 13S 7D 5D
DU      D VARIABLE    0021518           16S 5D
DEFTRC, R SYSTEM      0000703           51U 51U 50U 50U 48U 42U 42U 42U 41U 41U 40U 40U 39U 39U
DOTRAC, R SYSTEM      0000008           50U 48U 41U 40U 39U
FORTRC, R SYSTEM      0000109           51U 42U
I       I VARIABLE    0021478           53U 48S 48I 44U 40S 40I 35U 35U 35U 21U 21U 21U 18U 17U 16U 16U 15U 14S 7D
IGO     I ASSIGN      0021538           35S 33A 28S 27A 24S 23A 22U 22U 15S 14A 7D
II      I VARIABLE    0021658           44U 42S 42I 35S 24U 21S 18S
INITBO, I SYSTEM      0001408           24U
IOFLAG, I SYSTEM      0001308           24U
II      I VARIABLE    0027258           44U 42S 41I
J       I VARIABLE    0027338           53U 51S 50I 44U 40U 40P 40U 40U 40P 39U 39U 39P 31S
JINX    I EXTERNAL    0000608           35U 21U 7D
JJ      I VARIABLE    0027268           44U 40S 39I
K       I VARIABLE    0027348           55U 55S 44U 44U 42U 42U 42P 41U 41U 41P 39U 39U 39P 37S
KK      I VARIABLE    0027308           53U 52S 51I
L       L VARIABLE    0021508           11S 7D 6D
LL      I VARIABLE    0027368           53U 51U 51U 51P 46S
M       I VARIABLE    0027358           53U 50U 50U 50P 48U 48U 48P 47S 47U 44U 42U 42U 42P 41U 41U 41P 38S
MM      I VARIABLE    0027278           53U 50U 50U 50P 50S
N       I VARIABLE    0027248           53U 51U 51U 51P 48S 44U 42U 42U 42P 39S
NUMTRC, I SYSTEM      0001208           54U 33U 27U 23U
ONE     R VARIABLE    0021468           28U 8U
ONTRY   R SYSTEM      0002308           57U
P       P VARIABLE    0021648           16U 15U 15U 10S 7D
RR      R VARIABLE    0021638           20U 17S
RTOR,   R SYSTEM      0000409           15U
SBTRAC, R SYSTEM      0000308           15U
TAPE6=    FILE        0000006           24U
TERMBO, R SYSTEM      0001608           24U
TRACKT, R SYSTEM      0001008           21U
TRASH,  R SYSTEM      0001108           22U
TRCST,  R SYSTEM      0002208           55U 55U
VARTRC, R SYSTEM      0000208           54U 49U 48U 47U 46U 45U 40U 38U 37U 36U 35U 35U 31U 30U 29U 28U 21U 21U 14U 13U 12U
                                        11U 10U 9U
```

```
WRITE1, R SYSTEM      0001[illegible]        24U
X       R VARIABLE    0027318        29S
Y       R VARIABLE    0027328        30S
Z       R ARRAY       0000008 //     16S 3D
ZERO    R VARIABLE    0021558        29U 28U 8U
ZRO     R VARIABLE    0021568        29U 8U
BLOCK NAMES AND LENGTHS
          //          20
VARIABLE AND ARRAY NAMES SORTED BY OCTAL ADDRESS, TRAILING B=ABSOLUTE, C=RELATIVE TO BLANK COMMON
  0021458 CC          0021478 I          0021508 L          0021518 C[illegible]     0021538 IGO        0021548 ZERO
  0021548 ZRO         0021578 C          0021618 D          0021638 R[illegible]     0021648 A          0021658 II
  0021668 ONE         0027268 N          0027258 II         0027268 JJ         0027278 MM         0027308 KK
  0027318 X           0027328 Y          0027338 J          0027348 K          0027358 M          0027368 LL
  000000C Z           000012C ARRAY

  0027768       58*       FUNCTION JINX(I)
  0027769       59*       JINX=I*10*(I-1)
  0030018       60*       RETURN
  0030028       61*       END
SEU    61* COMMENT - 3 WORDS USED IN SETUP OF SUBPROGRAM   ********************************************

NUMBER AND NAME CROSS REFERENCE MAP
 NAME TYPE MAIN USE ADDRESS  BLOCK U=USED, S=STORED, D=DECLARED, I=DO INDEX, A=ASSIGN, P=DO PARAMETER
I       I PARAMETER   0000008          59U 59U 58D
JINX    I VARIABLE    0027768          60S 58D
VARIABLE AND ARRAY NAMES SORTED BY OCTAL ADDRESS, TRAILING B=ABSOLUTE, C=RELATIVE TO BLANK COMMON
  0027768 JINX

ST.       0026248   0001578
JINX      0030038   0000338
GENVAR.   0030368   0000018
OUTPUTS   0030378   0004248
FREEFOR   0034638   0012068
INPUTS    0046718   0013318
INOUTS    0062228   0002078
CPUCIO    0064318   0002728
CPUSYS    0067238   0000308
TRACES    0067538   0006148
ARGSTS    0075578   0000058
GETAA     0075748   0000178
SYSTEM    0076138   0011578
RTOBS     0107728   0001068
SIOS      0111008   0013208
ALNLOG    0124208   0001078
EXP       0125278   0000548
//        0126038   0000248
 CTIME =      226MS
*TRACE* STORE INTO REAL     VARIABLE A      AT SEQUENCE NUMBER   9*  VALUE = 9.9999999999999          1[illegible]
*TRACE* STORE INTO LOGICAL  VARIABLE L      AT SEQUENCE NUMBER  10*  VALUE =                  T   7777777777777777748
*TRACE* STORE INTO COMPLEX  VARIABLE C      AT SEQUENCE NUMBER  11*  VALUE = 1.700000000000    +[illegible]  4.[illegible]
*TRACE* STORE INTO DOUBLE   VARIABLE D      AT SEQUENCE NUMBER  12*  VALUE = .3141592653549700000000000000D-01
*TRACE* STORE INTO INTEGER  VARIABLE I      AT SEQUENCE NUMBER  13*  VALUE =                 13   00000000000000000158
*TRACE* SUBSCRIPT  FOR  ARRAY       Z       AT SEQUENCE NUMBER  15*  VALUE =     13 OUTSIDE RANGE 1 TO  . 10..
```

The message "CAUTION - ARRAY REPEATED IN TRACE LIST" occurs near 8* because TRACE SUBSCRIPTS causes all later dimensioned names to be traced and thus bounds tracing of an individual array name looks to the compiler as another TRACE of that name and results in a spurious error message.

The statement "NO TRACE ARITHMETIC" at 56* was inserted to stop tracing inside the function JINX.

The execution output message "*TRACE* NAME AND TOTALS FOR SUBPROGRAM TRAEXT." refers to the counts of the number of times subprograms have been called or referenced. Note: Because the maximum count which can be printed is 262143, higher counts will affect the last characters of the subprogram name as printed and the printed count will be incorrect.

```
*TRACE*  BOUNDS ON DOUBLE  VARIABLE DD      AT SEQUENCE NUMBER     15*  VALUE =  .999999999999999715782905570.0.01
*TRACE*  BOUNDS ON REAL    VARIABLE RR      AT SEQUENCE NUMBER     17*  VALUE =  13.0000000[illegible]0          1723540000000000000 33
*TRACE*  BOUNDS ON INTEGER VARIABLE II      AT SEQUENCE NUMBER     18*  VALUE =                    12     0000000000000000000149
*TRACE*  BOUNDS ON COMPLEX VARIABLE CC      AT SEQUENCE NUMBER     19*  VALUE = -1.000000000[illegible]6     .1*  15.000000000000
*TRACE*  BOUNDS ON REAL    VARIABLE ARRAY   AT SEQUENCE NUMBER     20*  VALUE =  149.0000000[illegible]0          1727522000000000000008
*TRACE* CALL OF SUBPROGRAM          JINX    FROM SEQUENCE NO.      20*
*TRACE* STORE INTO INTEGER VARIABLE JINX    AT SEQUENCE NUMBER     21*  VALUE =                   133     0000000000000000002053
*TRACE* STORE INTO INTEGER VARIABLE I       AT SEQUENCE NUMBER     21*  VALUE =                    13     0000000000000000000153
*TRACE*  BOUNDS ON INTEGER VARIABLE II      AT SEQUENCE NUMBER     21*  VALUE =                   133     0000000000000000002053
*TRACE* TRANSFER TO STATEMENT NUMBER    10  FROM SEQUENCE NO.      22*

THE LIST OF SUBPROGRAMS FOLLOWED BY EACH STATEMENT NUMBER AND THE NUMBER OF TIMES EACH IS USED
HAS SOME ZERO COUNTS WHEN TRACE TRANSFERS IS USED BUT TRACE STATEMENT NUMBERS IS NOT.
*TRACE* I/O LIST ELEMENT, ADDRESS = 0021658 AT SEQUENCE NUMBER     24*   INTEGER LIST ELEMENT USES E FORMAT SPEC
          0
*TRACE* TRANSFER TO STATEMENT NUMBER    20  FROM SEQUENCE NO.      22*
*TRACE* STORE INTO REAL    VARIABLE X       AT SEQUENCE NUMBER     29*  VALUE =                     8     3777000000000000000008
*TRACE* STORE INTO REAL    VARIABLE Y       AT SEQUENCE NUMBER     29*  VALUE =                     1     1777000000000000000008
*TRACE* STORE INTO INTEGER VARIABLE J       AT SEQUENCE NUMBER     30*  VALUE =                     8     0006305524744300[illegible]0018
*TRACE* STORE INTO DOUBLE  VARIABLE D       AT SEQUENCE NUMBER     31*  VALUE = .99999999999999939390144558[illegible]-245
*TRACE* TRANSFER TO STATEMENT NUMBER    30  FROM SEQUENCE NO.      22*
*TRACE* CALL OF SUBPROGRAM          JINX    FROM SEQUENCE NO.      22*
*TRACE* STORE INTO INTEGER VARIABLE JINX    AT SEQUENCE NUMBER     35*  VALUE =                   133     0000000000000000002053
*TRACE* STORE INTO INTEGER VARIABLE I       AT SEQUENCE NUMBER     35*  VALUE =                    13     0000000000000000000153
*TRACE*  BOUNDS ON INTEGER VARIABLE II      AT SEQUENCE NUMBER     35*  VALUE =                   133     0000000000000000002053
*TRACE* DO/FOR INITIAL   PARAMETER          AT SEQUENCE NUMBER     39*  ABS VALUE .GT. 131071
*TRACE* DO/FOR INCREMENT PARAMETER          AT SEQUENCE NUMBER     40*  ABS VALUE .GT. 131071
*TRACE* STORE INTO INTEGER VARIABLE I       AT SEQUENCE NUMBER     40*  VALUE =                     1     0000000000000000000018
*TRACE* DO LOOP INITIAL VALUE =      11     AT SEQUENCE NUMBER     41*  GREATER THAN TERMINAL VALUE =      1
*TRACE* DO/FOR INCREMENT VALUE=     -1.LE.0 AT SEQUENCE NUMBER     42*
*TRACE* FOR LOOP INITIAL VALUE=      1      AT SEQUENCE NUMBER     42*  LESS THAN TERMINAL VALUE =      11
*TRACE* DO LOOP TERMINAL-INCREMENT          AT SEQUENCE NUMBER     48*  GREATER THAN 131071
*TRACE* STORE INTO INTEGER VARIABLE I       AT SEQUENCE NUMBER     48*  VALUE =                     1     0000000000000000000018
*TRACE* DO/FOR TERMINAL  PARAMETER          AT SEQUENCE NUMBER     50*  ABS VALUE .GT. 131071
*TRACE* DO LOOP INITIAL VALUE = 131070      AT SEQUENCE NUMBER     50*  GREATER THAN TERMINAL VALUE = -131071
*TRACE* FOR LOOP TERMINAL-INCREMENT         AT SEQUENCE NUMBER     51*  LESS THAN -131071
*TRACE* ASSIGNED VARIABLE, VALUE =  002604A AT SEQUENCE NUMBER     22*  IS NOT EQUIVALENT TO ANY ITEM IN THE STATEMENT NUMBER LIST
*TRACE* TRANSFER TO STATEMENT NUMBER    40  FROM SEQUENCE NO.      22*
*TRACE* COMPUTED GOTO VALUE=             5  AT SEQUENCE NUMBER     55*  IS NOT INSIDE RANGE 1 TO NUMBER OF STATEMENTS (FATAL)
*TRACE* NAME AND TOTALS FOR SUBPROGRAM TRAEXT.
  JINX          1
*TRACE* NAME AND TOTALS FOR SUBPROGRAM ST.
        5       4
       10       1
       20       1
       30       1
       40       1
       50       0
      100       1
      110       0
     1000       1
```

10.2 SOURCE LISTING CONTROL STATEMENTS

LIST

NO LIST

PAGE

These three statements may be used anywhere and LIST will over-ride the L=O option on the MNF card (Appendix A). After a LIST statement, all further source statements are printed on the OUTPUT file. After a NO LIST statement no further source statements are printed except for the NO LIST statement. However error messages are always printed and if the listing is suppressed, the next ten line images are printed after any error messages that are given. When a PAGE statement is encountered, the statement is printed after which a new page is begun. MNF automatically begins a new page at each program or subprogram statement.

Deletion of a source listing only saves paper but not computer time.

10.3 CROSS REFERENCE LISTING CONTROL STATEMENTS

REFERENCES

NO REFERENCES

These statements in conjunction with the R option on the MNF control card govern the information inserted and the printing of the cross-reference map at the end of each subprogram. The map fully describes the statement numbers and names used in the program or subprogram, particularly where and how each is used (see Appendix I). NO REFERENCES stops the insertion of most of the information in the cross-reference map and deletes the printing of the map if NO REFERENCES is in effect at the END statement. REFERENCES begins the insertion of information and allows the printing of the map. It also overrides R=O on the MNF control card and resets R to the default option (see Appendix A).

Deletion of the cross-reference map saves paper and about 20% of compile time but eliminates a valuable aid for debugging and program information; thus the REFERENCES and NO REFERENCES statements should only be used with caution.

3.4
PSEUDO
COMPASS
LISTING
CONTROL
STATEMENTS

```
CODE
NO CODE
```

These two statements can be used anywhere in the source deck (program and subprograms) and will override the Ø option on the MNF control card; however the CODE statement is only effective when the FORTRAN listing is being put out. When the CODE statement appears, an object code pseudo-Compass listing for each Fortran statement is put out after the statement MNF cannot output true Compass assembly language since it generates instructions directly in machine language. The object code pseudo-Compass listing is only for easier reference by the user and cannot be assembled directly by the Compass assembler.

The NO CODE statement causes the object listing to be suppressed until the next CODE statement (if any) is reached.

APPx 6

COMPILATION AND EXECUTION A

A.1 MNF CONTROL CARD*

The control card to obtain the compilation and execution of a Fortran source and data deck has the characters MNF followed by a period if the implied parameters of the MNF compiler are to be used, or by an optional free-form parameter list.

MNF.
or
MNF(p_1,p_2,...,p_n)

Parameters are separated by commas if the a=b form is used. Otherwise, after a comma (or the open parenthesis) single letter forms may be combined without commas. Blank columns may be used in the optional parameter list for readability, but are ignored by the MNF compiler.

Examples:

```
MNF(BCØ)
MNF(BR=0,I=TAPE1,L=TAPE2,AP=999,E=2)
```

The standard control card MNF. is equivalent to the following control card:

```
MNF(I=INPUT,L=OUTPUT,E=0,P=64,R=3)
```

i.e. 1) Fortran source from the card deck on the INPUT file
2) Source listing on the OUTPUT file to the printer
3) All levels of error messages are printed
4) Page limit on execution of 64 pages
5) The complete cross reference listing (except for unused variables) is printed

If any errors are found on the MNF control card, the message:

ILLEGAL CONTROL PARAMETER x

(where x is the illegal character on the control card) will appear on the users's output following the printed control card after which the character is ignored and the rest of the control

---

* Students who use the MNF Batch system should first read section A.5.

card is scanned. The legal parameter characters explained below are A,B,C,D,E,F,I,L,Ø,P,R,T and U.

PARAMETER LIST

CHARACTER	EXPLANATION AND/OR ACTION
A	ANSI/CDC I/O INTERACTION If A is omitted (default condition) the I/O interaction conforms to that of the American National Standards Institute standard Fortran (See chapter 8). If A is used, the Control Data Corporation (CDC) I/O interaction is followed (see 8.9).
B	BINARY FILE OF OBJECT ROUTINES TO BE APPENDED If B is omitted, the Fortran source deck is compiled to binary and all external subprogram requests are obtained (if possible) from the Fortran subroutine library. If B is used, the Fortran source deck is compiled to binary, the file of binary routines on the file LGO is loaded after the compiled binary object code, and then all external subprograms are obtained from the Fortran subroutine library (if possible). This allows Compass and other Fortran compilers to supply subprograms to the MNF compiler. MNF opens and rewinds the LGØ file before loading it. If B=filnam, the action is identical to B except that filnam rather than LGØ is the name of a file of binary routines which are appended to the MNF compiler generated binary object routines. However, B=INPUT is not a valid request so users having binary decks on the INPUT file should copy them to filnam before the MNF control card is processed and use B=filnam. MNF opens and rewinds the named file before loading it.

CHARACTER	EXPLANATION AND/OR ACTION
	The following two control card sequences are examples which require the B parameter. They will compile FUN or FTN Fortran routines to binary on the LGØ file. The Fortran source deck is then compiled to binary under MNF and the file of binary routines on the LGØ file is appended. Note that the C parameter in MNF is necessary when using the FTN compiler with MNF (see below).

```
FUN(S)     FTN.
MNF(B)     MNF(BC)
```

C	CALLING SEQUIENCE FOR EXTERNAL ROUTINES
	If C is omitted, the RUN/FUN compiler calling sequence is generated for subprograms except for those routines that are unique to MNF. See Appendix M.
	If C is used, the FTN or Fortran Extended calling sequence is generated for subprograms except for those routines that are unique to MNF. See Appendix M.
D	DEBUGGING MODE
	If D is omitted, Fortran comment source cards with C$ in the first two columns are considered normal comment statements and if any FATAL-TO-EXECUTION errors are detected, the compiler does not go into the execution phase.
	If D is used, comment source cards with C$ in the first two columns are treated as normal Fortran statements and the first two columns are ignored (see 1.1). This gives two options to the Fortran user:

1) Standard Fortran statements that are necessary for debugging a Fortran source deck may be left in the deck for possible future use. The only limitation is that the C$ in the first two columns leaves only three columns for statement numbers on these debugging statements.

CHARACTER	EXPLANATION AND/OR ACTION
	2) Non-standard Fortran tracing and listing statements of the MNF compiler will be invisible when the source deck is used on other Fortran compilers if these statements have C$ in the first two columns.
	In addition to the C$ option, the use of D means that the compiler will enter the execution phase even though FATAL-TO-EXECUTION errors were detected (unless DEADLY errors were detected). The object code will then run until normal termination.
E	ERROR LEVEL OF DIAGNOSTICS
	If E is omitted, all six levels of error messages (NON-STANDARD, COMMENT, CAUTION, WARNING, FATAL-TO-EXECUTION, and DEADLY-TO-COMPILATION) will go to the file OUTPUT (see OUTPUT, below) after the source statement causing such a message.
	If E=digit, all error messages of that level and lower are not output. Each level of message is assigned a digit value. The levels are: 1) NOTE (NON-STANDARD, i.e. NON-ANSI) 2) COMMENT 3) CAUTION 4) WARNING 5) FATAL-TO-EXECUTION
	The level DEADLY-TO-COMPILATION is not assigned a value since only two messages may occur at this level: POSSIBLE MACHINE ERROR and NOT ENOUGH STORAGE FOR COMPILATION. When the latter message occurs, all current generated binary code is cleared and the compiler continues compiling the remaining Fortran statements and checks for compilation errors. See Appendix G for a further explanation of the error types.
	Those programmers who use octal constants, multiple statements per card, multiple replacement, and expressions in the DO parameter and output lists will

CHARACTER	EXPLANATION AND/OR ACTION
	quickly tire of the many NON-STANDARD messages and use E=1 as a compiler parameter. Since the MNF compiler tries to identify all errors or possible errors, it will sometimes give out COMMENT and CAUTION messages for correct code. In this case, E=3 used in later runs is useful in avoiding those messages. If E=5 is used, the program will be executed even if FATAL-TO-EXECUTION errors are detected since no error messages are printed.
F	FORCED STORES

If the forced store parameter F is omitted, MNF assumes that each computer word is known by a unique name within the program and each subprogram. Under certain circumstances, it is possible for a single computer word to have more than one variable name associated with it. If so, there are a few cases in which the wrong answer will be generated when the MNF compiler tries to optimize by using operands already in the machine operating registers.

If F is used, stores are forced before loads and equivalenced common parameter operands are not remembered across statements. Thus MNF will not optimize code but will also not generate incorrect results in the rare cases described below.

The following four examples illustrate a computer word having more than one name and also show rare cases where MNF optimization can cause errors to occur.

```
I)   C     I AND J ARE THE SAME WORD
           EQUIVALENCE (I,J)
           J=1+I                    (2)
           K=1+I                    (3)
```

```
II)        CALL SAM(I,I)
           END
           SUBROUTINE SAM(I,J)
      C    I AND J ARE THE SAME WORD
           J=1+I                        (2)
           K=1+I                        (3)

III)       COMMON J
           CALL GEO(J)
           END
           SUBROUTINE GEO(I)
           COMMON J
      C    I AND J ARE THE SAME WORD
           J=1+I                        (2)
           K=1+I                        (3)
```

In each of the above examples, if the F parameter is not used, the value of I in statements (2) and (3) will be the same (incorrectly) because I will be loaded for statement (3) prior to the storage of the new, equivalenced value of I from statement (2). If F is used, the result of statement (2) will be stored prior to the loading of the value I for statement (3).

If the programmer knows of this possible condition, he can also avoid invalid results by using the name that was most recently to the left of the equal (thus (3) becomes K=1+J) or by putting a statement number on the statement.

```
IV)        EQUIVALENCE (I,J)
      C    I AND J ARE THE SAME WORD
           DO 10 I=1,10                 (2)
           K=1+J                        (3)
```

A similar situation exists in the above example where I is not actually stored before J is used if the F parameter is omitted. Again, this may be corrected by using the F option, by using I in statement (3) or by putting a statement number on it.

Most programs will run correctly without the F option, but if it is used and the output results differ from the normal mode then the program has a computer word known by two names.

CHARACTER	EXPLANATION AND/OR ACTION
I	INPUT FILE FOR FORTRAN SOURCE

If the source input file parameter is omitted (default condition) the Fortran source statements will be taken from the file INPUT (i.e. the same file which contains the control cards). If the source statements are on another file, a parameter of the following form must be provided: I=filnam (default I=INPUT) where filnam is the name of the file containing the source statements. Parameters of the forms I=INPUT and I are equivalent to omitting the parameter.

L LISTING OF THE FORTRAN SOURCE STATEMENTS

If the source listing output parameter is omitted (default condition) the Fortran source listing is written on the file OUTPUT (however, see OUTPUT below). If the listing is to be written on another file, a parameter of the following form must be provided: L=filnam (default L=OUTPUT) where filnam is the name of the file to be written, and it replaces the OUTPUT file as the source and object listing file (see Ø and OUTPUT below). Parameters of the form L=OUTPUT and L are equivalent to omitting the parameter.

If the parameter L=0 is used, the Fortran source listing is not written out, except that (subject to the E parameter) any errors that are detected cause the resulting error message to be written out and the next ten line images are also written out to help the user find the statement in error.

The Fortran source listing may be further manipulated by the Fortran statements PAGE, LIST and NOLIST which may appear between Fortran subprograms or between individual Fortran statements and thus give control over the spacing of source statements on the output and control which source statements may appear on the output.

CHARACTER	EXPLANATION AND/OR ACTION
	Note that a NOLIST statement will also suppress the Ø and R output options until the next LIST statement. See also 10.3.
Ø	OBJECT CODE LISTING If the object code listing parameter is omitted (default option) the mnemonic pseudo-Compass object listing is not written on the OUTPUT file (see OUTPUT below). If the object code listing for each Fortran statement is desired, a parameter of the form Ø must be used which causes the object listing to be written on the file OUTPUT if the L or LIST options are currently set. If Ø=filnam is used, the mnemonic pseudo-Compass listing is written out on the file filnam and it replaces the OUTPUT file as the source and object listing file (see OUTPUT below). If the Fortran source is being written out, the Fortran mnemonic pseudo-Compass listing may be further manipulated by the Fortran statements CODE and NOCODE which may appear between Fortran subprograms or between individual Fortran statements. See also 10.4. Thus, these statements give control over which sections of the Fortran source will be followed by the object code listing. Note that NOLIST will also cause the object code listing to be suppressed.
OUTPUT	All four distinct output line images (Fortran source, pseudo-Compass object, errors and cross-reference) go to the file OUTPUT or to that file which was last equated to one of the letters L or Ø. Thus, !ØNF(L=TAPE1,Ø=TAPE2) will be equivalent to !ØNF(LØ=TAPE2) since only one buffer area is supplied for these line images.

CHARACTER	EXPLANATION AND/OR ACTION
P	PAGE COUNT LIMIT

If P is omitted, the execution page count limit for the standard OUTPUT file is set to 64 pages of up to 66 lines each. (There is no limit to the number of pages used during compilation, or during execution if the file is not OUTPUT.)

If P is used, the execution page count is set to zero. Thus the first output on the standard OUTPUT file will terminate the job.

If P=decimal number, the execution page count for the standard OUTPUT file is set to that decimal number.

R CROSS-REFERENCE LISTING

If R=0 is used, no cross reference listing is written on the OUTPUT file.

If R=1 is used, cross references are written on the OUTPUT file for Fortran statement numbers and names sorted in numerical and alphabetic order except for unused variables having no active references in the current subprogram.

If R=2 is used, a cross references map for variables and arrays sorted by octal address is written on the OUTPUT file. This is helpful in interpreting the octal dump of an abnormal termination.

If R=4 is used, null cross reference names are written on the OUTPUT file. This applies only if the R=1 option is set so that R=4 is not actually allowed by itself. However, these values may also be summed. Thus, R=5 gives the cross reference listing including nulls but does not give a sorted map. Valid numbers are 0,1,2,3,5,7. See Appendix I for a complete description of the cross reference map.

CHARACTER	EXPLANATION AND/OR ACTION
	If the cross reference listing parameter is omitted (default value) the value R=3 is assumed and the cross reference map is written on the OUTPUT file. The parameter R is equivalent to omitting the parameter. The cross reference listing may be further manipulated by the Fortran statements REFERENCES and NO REFERENCES which may appear between Fortran subprograms and between individual Fortran statements. Thus, these statements give control over which sections of a Fortran source deck will provide names and numbers to the cross reference listing. Note that a NOLIST statement will also suppress the cross reference listing. See also 10.2 and 10.3
T	TRACE POSSIBLE ERRORS If T is omitted, normal object execution code is produced for Fortran statements. If T is used, most of the tracing facilities of MNF are turned on at the first statement of the program. They only cause printed output if an error occurs or give total usage counts at the end of execution. They are: TRACE DOLOOPING - checks for initial greater than terminal in DO and implied DO loops (terminal greater than initial in FOR loops), negative or zero DO/FOR incrementation, terminal + increment greater than 131071 in DO loops (or terminal - increment less than -131071 in FOR loops) or the absolute value of an indexing parameter is greater than 131071 TRACE FORMATIO - checks that the type of the format specification implies only certain types of list variables. TRACE SUBSCRIPTS - checks subscript to be between 1 and total dimension product.

CHARACTER	EXPLANATION AND/OR ACTION
	TRACE TRANSFERS - checks the assign variable against the assigned GO TO number list and checks the computed GO TO variable (or expression) against the total list of GO TO numbers.
These counts are put on the OUTPUT file after the program has completed execution.	TRACE STATEMENT NUMBERS - counts the number of times a statement with a statement number has been executed or the number of times the FORMAT has been referenced in an I/O statement. TRACE SUBPROGRAM CALLS - counts the number of times a subprogram has been referenced or called.
	See 10.1 for a more complete explanation of these TRACE statements.
U	USER LIBRARY
	If U is omitted, no user library is searched by MNF.
	If a parameter of the form U=filnam is used, when MNF has finished loading any binary file specified by the B parameter, external subprograms are obtained from user library specified by filnam after which remaining external subprograms are obtained from the Fortran subroutine library. Those who wish to make up user libraries should see a UCC consultant for further information.

A.2
DECK
STRUCTURE*

Program and Subprograms Structure

MNF program or subprogram or block data source decks are divided into five sections as follows; they must conform to the order shown.

Section	Content
A	PROGRAM, SUBROUTINE, or BLOCK DATA statement

*Students who use the MNF Batch system should read section A.5

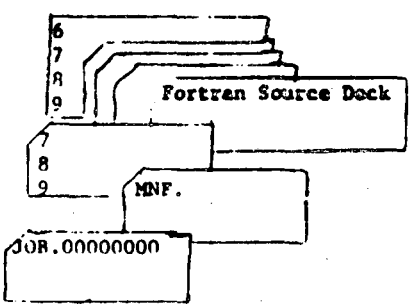
Section	Content
B	Specification statements (DIMENSION, TYPE, DATA, etc.)
C	Statement Function definitions
D	Executable statements (X=Y, GO TO 14, etc.)
E	END statement

Sections A and E must appear in every subprogram (except that a PROGRAM statement need not be used in the main program). Sections B, C, and D may include FORMAT statements and NAMELIST statements. All sections may contain comment lines, global TRACE statements, and the statements described in sections 10.2, 10.3, and 10.4 (PAGE, LIST, etc.). If section A is a BLOCK DATA statement, sections C and D must not be included in the subprogram.

Source decks for MNF must only contain Fortran statements. Compass decks must be separately assembled by the COMPASS assembler. However, Compass subprograms and subprograms compiled with RUN, FUN or FTN may be combined in an MNF program using the B parameter of the MNF control card (see A.1).

Sample Deck Structures

1. Compilation and execution, no data. Fortran only.



APPX-7

FORTRAN STATEMENT LIST E

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)

* non-standard Fortran statement

	PROGRAM UNIT STATEMENTS	SYMBOL	PAGE
MAIN	PROGRAM name ($file_1,\ldots,file_n$)	*P	7-3
PROGRAM	PROGRAM name	*P	7-3
SUBPROGRAM	FUNCTION name ($p_1,\ldots,p_n$)	P	7-11
	type FUNCTION name ($p_1,\ldots,p_n$)	P	7-11
	SUBROUTINE name ($p_1,\ldots,p_n$)	P	7-6
	SUBROUTINE name	P	7-6
SPECIFICATION	BLOCK DATA	P	5-16
SUBPROGRAM	BLOCK DATA name	*P	5-16
ENTRY POINT	ENTRY name	*E	7-9
	EXTERNAL $name_1,\ldots,name_n$	S	7-7
INTERSUBPROGRAM	CALL name	E	7-6
TRANSFER	CALL name ($p_1,\ldots,p_n$)	E	7-6
STATEMENTS	RETURN	E	6-16
STATEMENT FUNCTION	name ($p_1,\ldots,p_n$) = expression	A	7-13

	DATA DECLARATION AND STORAGE ALLOCATION	SYMBOL	PAGE
TYPE DECLARATION	INTEGER $name_1,\ldots,name_n$	D	5-1
	TYPE INTEGER $name_1,\ldots,name_n$	*D	5-1
	REAL $name_1,\ldots,name_n$	D	5-1
	TYPE REAL $name_1,\ldots,name_n$	*D	5-1
	COMPLEX $name_1,\ldots,name_n$	D	5-1
	TYPE COMPLEX $name_1,\ldots,name_n$	*D	5-1
	DOUBLE PRECISION $name_1,\ldots,name_n$	D	5-1
	DOUBLE $name_1,\ldots,name_n$	*D	5-1
	TYPE DOUBLE PRECISION $name_1,\ldots,name_n$	*D	5-1
	TYPE DOUBLE $name_1,\ldots,name_n$	*D	5-1
	LOGICAL $name_1,\ldots,name_n$	D	5-1
	TYPE LOGICAL $name_1,\ldots,name_n$	*D	5-1
	ECS $name_1,\ldots,name_n$	*D	5-1
	TYPE ECS $name_1,\ldots,name_n$	*D	5-1
STORAGE ALLOCATION	DIMENSION $name_1(d_1),\ldots,name_n(d_n)$	D	5-2
	COMMON $name_1,\ldots,name_n$	D	5-4
	COMMON / /$name_1,\ldots,name_n$	D	5-4
	COMMON /$blkname_1$/$name_1,\ldots,name_n$.../$blkname_n$/ $name_1,\ldots,name_n$	D	5-4
	EQUIVALENCE $(name_1,\ldots,name_n),\ldots,(name_1,\ldots,name_n)$	D	5-9
DATA DECLARATION	DATA $vlist_1/dlist_1/,\ldots,vlist_n/dlist_n/$	D	5-11
	DATA $(vlist_1 = dlist_1),\ldots,(vlist_n = dlist_n)$	*D	5-11
	ASSIGNMENT AND CONTROL		
ASSIGNMENT	v = arithmetic expression	E	4-1
	logical v = logical or relational expression	E	4-3
	v = masking expression	*E	4-4
MULTIPLE ASSIGNMENT	$v_1 = \ldots = v_n$ = expression	*E	4-4

		SYMBOL	PAGE
CONTROL	GO TO sn	E	6-1
	GO TO (sn_1,...,sn_n), iv	E	6-3
	GO TO (sn_1,...,sn_n) iv	*E	6-3
	GO TO (sn_1,...,sn_n), expression	*E	6-3
	GO TO (sn_1,...,sn_n) expression	*E	6-3
	GO TO iv, (sn_1,...,sn_n)	E	6-2
	GO TO iv (sn_1,...,sn_n)	*E	6-2
	GO TO iv	*E	6-2
	ASSIGN sn TO iv	E	6-2
DECISION	IF (arithmetic expression) sn_N,sn_Z,sn_P	E	6-5
	IF (masking expression) sn_N,sn_Z,sn_P	*E	6-5
	IF (logical expression) sn_T,sn_F	*E	6-6
	IF (logical expression) statement	E	6-5
LOOP CONTROL	DO sn iv = m_1,m_2,m_3	E	6-7
	DO sn iv = m_1,m_2	E	6-7
	DO sn v = $expression_1$,$expression_2$,$expression_3$	*E	6-7
	DO sn v = $expression_1$,$expression_2$	*E	6-7
	FOR sn iv = m_1,m_2,m_3	*E	6-7
	FOR sn iv = m_1,m_2	*E	6-7
	FOR sn v = $expression_1$,$expression_2$,$expression_3$	*E	6-7
	FOR sn v = $expression_1$,$expression_2$	*E	6-7
	sn CONTINUE	E	6-14
STOP, PAUSE, END	STOP	E	6-15
	STOP n	E	6-15
	PAUSE	E	6-15
	PAUSE n	E	6-15
	END	N	6-16

INPUT/OUTPUT

FORMATTED	PRINT fn,iolist	*E	9-1
	PRINT fn	*E	9-1
	PUNCH fn,iolist	*E	9-2
	PUNCH fn	*E	9-2
	WRITE (u,fn) iolist	E	9-3
	WRITE (u,fn)	E	9-3

		SYMBOL	PAGE
	READ fn,iolist	*E	9-6
	READ fn	*E	9-6
	READ (u,fn) iolist	E	9-6
	READ (.END. = sn_1,.ERR. = sn_2,u,fn) iolist	*E	9-8
	READ (u,fn)	E	9-6
	READ (.END. = sn_1,.ERR. = sn_2,u,fn)	*E	9-8
BINARY	WRITE (u) iolist	E	9-4
	WRITE (u)	E	9-4
	READ (u) iolist	E	9-6
	READ (.END. = sn_1,.ERR. = sn_2,u) iolist	*E	9-8
	READ (u)	E	9-6
	READ (.END. = sn_1,.ERR. = sn_2,u)	*E	9-8
FORMAT-FREE	PRINT,iolist	*E	9-13
	PUNCH,iolist	*E	9-13
	WRITE (u,) iolist	*E	9-13
	READ, iolist	*E	9-9
	READ (u,) iolist	*E	9-9
	READ (.END. = sn_1,.ERR. = sn_2,u,) iolist	*E	9-10
BUFFERED	BUFFER OUT (u,j) (a,b)	*E	9-25
	BUFFER IN (u,j) (a,b)	*E	9-25
INTERNAL DATA CONVERSION AND TRANSFER	ENCODE (cc,fn,v) iolist	*E	9-27
	DECODE (cc,fn,b) iolist	*E	9-27
	DECODE (.ERR. = sn_1,cc,fn,v) iolist	*E	9-27
NAMELIST	NAMELIST /$lstname_1$/$name_1$,...,$name_n$.../$lstname_n$/ $name_1$,...,$name_n$/	*S	9-17
	WRITE (u,lstname)	*E	9-17
	READ (u,lstname)	*E	9-17
	PRINT lstname	*E	9-17
	PUNCH lstname	*E	9-17
	READ lstname	*E	9-17
FILE MANIPULATION	REWIND u	E	9-20
	REWIND ulist	*E	9-20
	BACKSPACE u	E	9-21
	BACKSPACE ulist	*E	9-21
	ENDFILE u	E	9-21
	ENDFILE ulist	*E	9-21
	UNLOAD u	*E	9-21
	UNLOAD ulist	*E	9-21

	INPUT/OUTPUT LISTS AND FORMATS	SYMBOL	PAGE
IMPLIED DO LOOP	$(\ldots((iolist, i_1=m_1, m_2, m_3), i_2=n_1, n_2, n_3), \ldots, i_n=k_1, k_2, k_3)$		8-2
FORMAT	sn FORMAT $(fs_1, \ldots, fs_n)$	S	8-4
DATA CONVERSION	rEw.d		8-8
	rFw.d		8-12
	rGw.d		8-14
	rDw.d		8-15
	rIw		8-15
	rLw		8-20
	rAw		8-18
	rRw	*	8-19
	rOw	*	8-17
	sP		8-21
RECORD AND CHARACTER MANIPULATION	nX		8-24
	$nHh_1, \ldots, h_n$		8-25
	...	*	8-26
	¢...¢	*	8-26
	nT	*	8-27
	/		8-29
	r( )		8-30

	DEBUGGING AND OUTPUT LISTING CONTROL		
GENERAL TRACING	TRACE	*L	10-12
	NO TRACE	*L	10-12
SPECIFIC TYPE OF TRACING	TRACE ARITHMETIC	*L	10-5
	NO TRACE ARITHMETIC	*L	10-5
	TRACE DOLOOPING	*L	10-6
	NO TRACE DOLOOPING	*L	10-6
	TRACE FORMATIO	*L	10-7
	NO TRACE FORMATIO	*L	10-7
	TRACE STATEMENT NUMBERS	*L	10-4
	NO TRACE STATEMENT NUMBERS	*L	10-4
	TRACE SUBPROGRAM ENTRY	*L	10-4
	NO TRACE SUBPROGRAM ENTRY	*L	10-4
	TRACE SUBPROGRAM CALLS	*L	10-4
	NO TRACE SUBPROGRAM CALLS	*L	10-4

		SYMBOL	PAGE
	TRACE SUBSCRIPTS	*L	10-8
	NO TRACE SUBSCRIPTS	*L	10-8
	TRACE TRANSFERS	*L	10-9
	NO TRACE TRANSFERS	*L	10-9
SPECIFIC	TRACE list	*L	10-9
ELEMENT TRACING	NO TRACE list	*L	10-12
SOURCE	LIST	*L	10-18
LISTING	NO LIST	*L	10-18
CONTROL	PAGE	*L	10-18
PSEUDO-COMPASS	CODE	*L	10-19
LISTING CONTROL	NO CODE	*L	10-19
CROSS	REFERENCES	*L	10-18
REFERENCE MAP	NO REFERENCES	*L	10-18
	OVERLAY		
	OVERLAY (filname,i,j)	*L	A-14

Fortran Library Functions and Subroutines* F

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_1 or a to denote formal parameters. Type complex parameters are also denoted this way or as x + i * y.

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 COMPASS 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.

---

*The lists in this appendix do not include the large number of scientific subroutines available at the UCC. See the UCC consultant for further information about them.

Fortran Library Functions

	Symbolic Names	Definition and Limitations
1	ABS	Absolute value $\lvert a \rvert$, i.e. a if $a \geq 0$, -a if $a < 0$
2	IABS	
3	DABS	
4	CABS	Modulus of a, i.e. $\sqrt{REAL^2(a) + AIMAG^2(a)}$
5	FLOAT	Conversion from integer to real
6	IFIX	Conversion from real to integer with truncation (same as INT)
7	AINT	Truncation of a : [a] = Sign of a times largest integer $\leq \lvert a \rvert$,
8	INT	i.e. sign of a times the integer i such that $i \leq \lvert a \rvert < i + 1$
9	IDINT	$\lvert a \rvert < 2^{59}-1$
10	AMOD	Remaindering. Result is $a_1 - [a_1/a_2] * a_2$, where [x]
11	MOD	is defined as the integer resulting from the truncation of a_1/a_2.
12	DMOD	If a_2 is zero, these functions are undefined. For DMOD,
		$\lvert a_1/a_2 \rvert$ < 2 ** 96. For MOD, $\lvert a_1 \rvert$ < 2 ** 48, $\lvert a_2 \rvert$ < 2 ** 48.
13	AMAX0	Determine maximum argument. Maximum $(a_1, a_2, \ldots)$
14	AMAX1	For AMAX0 and MAX0, $\lvert$(maximum of previous a_j) $-a_i \rvert < 2^{59}-1$
15	MAX0	
16	MAX1	
17	DMAX1	
18	AMIN0	Determine minimum argument. Minimum $(a_1, a_2, \ldots)$
19	AMIN1	For AMIN0 and MIN0, $\lvert$(minimum of previous a_j) $-a_i \rvert < 2^{59}-1$
20	MIN0	
21	MIN1	
22	DMIN1	
23	SIGN	Transfer of sign. Sign of a_2 times $\lvert a_1 \rvert$
24	ISIGN	If $a_2 = 0$, use $\lvert a_1 \rvert$. If $a_2 = -0$, Use $-\lvert a_1 \rvert$.
25	DSIGN	
26	DIM	Positive difference: a_1 - minimum (a_1, a_2)
27	IDIM	For IDIM, $\lvert a_1 - a_2 \rvert < 2^{59}-1$
28	SNGL	Truncate to obtain most significant part of double precision argumer
29	DBLE	Express single-precision argument in double precision form

Standard or Non-standard FORTRAN*	Number of Arguments	Type of: Argument(s)	Function Result	Compiler Type**	Example
S	1	REAL	REAL	I	Y=ABS(R)
S	1	INTEGER	INTEGER	I	J=IABS(I)
S	1	DOUBLE	DOUBLE	I	DOUBLE A,D $ D=DABS(A)
S	1	COMPLEX	REAL	L	COMPLEX C $ R=CABS(C)
S	1	INTEGER	REAL	I	R=FLOAT(I)
S	1	REAL	INTEGER	I	I=IFIX(R)
S	1	REAL	REAL	I	Y=AINT(R)
S	1	REAL	INTEGER	I	I=INT(R)
S	1	DOUBLE	INTEGER	I	DOUBLE D $ I=IDINT(D)
S	2	REAL	REAL	I	R=AMOD(A1,A2)
S	2	INTEGER	INTEGER	I	J=MOD(I1,I2)
S	2	DOUBLE	DOUBLE	L	DOUBLE D,D1,D2 $ D=DMOD(D1,D2)
S	≥ 2	INTEGER	REAL	I	R=AMAX0(I,J,K)
S	≥ 2	REAL	REAL	I	R=AMAX1(X,Y,Z)
S	≥ 2	INTEGER	INTEGER	I	I=MAX0(J,K,L,M)
S	≥ 2	REAL	INTEGER	I	I=MAX1(A,B)
S	≥ 2	DOUBLE	DOUBLE	I	DOUBLE D,X,Y,Z $ D=DMAX1(X,Y,Z)
S	≥ 2	INTEGER	REAL	I	R=AMIN0(I,J)
S	≥ 2	REAL	REAL	I	R=AMIN1(R,S)
S	≥ 2	INTEGER	INTEGER	I	L=MIN0(I,J,K)
S	≥ 2	REAL	INTEGER	I	J=MIN1(1.5,S)
S	≥ 2	DOUBLE	DOUBLE	I	DOUBLE D,C,B $ B=DMIN1(D,C)
S	2	REAL	REAL	I	R=SIGN(X,Y)
S	2	INTEGER	INTEGER	I	I=ISIGN(J1,J2)
S	2	DOUBLE	DOUBLE	I	DOUBLE D,A,B $ A=DSIGN(A,B)
S	2	REAL	REAL	I	R=DIM(Z,-1.0)
S	2	INTEGER	INTEGER	I	I=IDIM(K1,K2)
S	1	DOUBLE	REAL	I	DOUBLE D $ R=SNGL(D)
S	1	REAL	DOUBLE	I	DOUBLE D $ D=DBLE(R)

* 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 $a_1 + a_2 * \sqrt{-1}$
33	CONJG	Obtain conjugate of a complex argument
34	AND	Boolean AND $a_1 \wedge a_2 \wedge \ldots \wedge a_n$
35	OR	Boolean OR $a_1 \vee a_2 \vee \ldots \vee a_n$
36	EOR	Boolean Exclusive OR
37	XOR	
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	LRSHFT	
41	ICOUNT	Count the number of bits in a word that are 1
42	EXP	Exponential, e^a. For complex: $e^x(\cos(y) + i * \sin(y))$
43	DEXP	For real and double precision, $a < 740.3$.
44	CEXP	For complex, $x < 740.3$, $\lvert y \rvert < 1.4E14$. If a (or x) < -675.82, result is zero.
45	ALOG	Natural logarithm, $\log_e(a)$. For complex: $0.5*\log_e(x^2+y^2)+i*\arctan(y/$
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	
50	SQRT	Square root, $(a)^{1/2}$. For complex $(x^2+y^2)^{1/2} *$
51	DSQRT	$(\cos(\arctan(y/x))+i*\sin(\arctan(y/x)))$ For real and
52	CSQRT	double precision: $a \geq 0.0$ For complex, $x \neq 0.0$ or $y \neq 0.0$
53	SIN	Trignometric sine, sin(a), a in radians. For real and double
54	DSIN	precision, $\lvert a \rvert < 1.4E14$. For complex, $\lvert x \rvert < 1.4E14$, $\lvert y \rvert < 740.3$
55	CSIN	For complex, $\sin(x)*\cosh(y)+i*\cos(x)*\sinh(y)$
56	COS	Trigonometric cosine, cos(a), a in radians. For real and double
57	DCOS	precision, $\lvert a \rvert < 1.4E14$. For complex, $\lvert x \rvert < 1.4E14$, $\lvert y \rvert < 740.3$
58	CCOS	For complex, $\cos(x)*\cosh(y)-i*\sin(x)*\sinh(y)$

$$\sinh(a) = \frac{e^a - e^{-a}}{2} \qquad \cosh(a) = \frac{e^a + e^{-a}}{2}$$

Standard or Non-standard FORTRAN*	Number or Arguments	Type of: Argument(s)	Function Result	Compiler Type**	Example
S	1	COMPLEX	REAL	I	COMPLEX C $ R=REAL(C)
S	1	COMPLEX	REAL	I	COMPLEX C $ R=AIMAG(C)
S	2	REAL	COMPLEX	I	COMPLEX C $ C=CMPLX(A,B)
S	1	COMPLEX	COMPLEX	I	COMPLEX C,D $ C=CONJG(D)
N	≥ 2	Single words	Octal	I	R=AND(A1,A2)
N	≥ 2	Single words	Octal	I	C=OR(R1,R2)
N	≥ 2	Single words	Octal	I	D=EOR(C1,C2)
N	≥ 2	Single words	Octal	I	E=XOR(D1,D2)
N	1	Single word	Octal	I	B=COMPL(A)
N	2	a_1 single word, a_2 integer	Octal	I	B=SHIFT(A,1)
N	2	a_1 single word, a_2 integer	Octal	I	R=LRSHIFT(X,-3)
N	1	Single word	INTEGER	I	J=ICOUNT(B)
S	1	REAL	REAL	L	R=EXP(Y)
S	1	DOUBLE	DOUBLE	L	DOUBLE X,Y $ X=DEXP(Y)
S	1	COMPLEX	COMPLEX	L	COMPLEX Z,W $ W=CEXP(Z)
S	1	REAL	REAL	L	Z=ALOG(Y)
S	1	DOUBLE	DOUBLE	L	DOUBLE U,V $ U=DLOG(V)
S	1	COMPLEX	COMPLEX	L	COMPLEX I,J $ J=CLOG(I)
S	1	REAL	REAL	L	B=ALOG10(A)
S	1	DOUBLE	DOUBLE	L	DOUBLE T,U $ T=DLOG10(U)
S	1	REAL	REAL	L	Y=SQRT(X)
S	1	DOUBLE	DOUBLE	L	DOUBLE Z,W $ Z=DSQRT(W)
S	1	COMPLEX	COMPLEX	L	COMPLEX A,B $ A=CSQRT(B)
S	1	REAL	REAL	L	X=SIN(Y)
S	1	DOUBLE	DOUBLE	L	DOUBLE E,F $ E=SIN(F)
S	1	COMPLEX	COMPLEX	L	COMPLEX C,D $ C=CSIN(D)
S	1	REAL	REAL	L	Z=COS(Y)
S	1	DOUBLE	DOUBLE	L	DOUBLE G,H $ G=DCOS(H)
S	1	COMPLEX	COMPLEX	L	COMPLEX I,J $ I=CCOS(J)

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

	Symbolic Name	Definition and Limitations
59	TAN	Trigonometric tangent, tan(a), a in radians. $\lvert a \rvert < 8.8E14$
60	ASIN	Arcsin (a), $\lvert a \rvert \leq 1.0$, $-\pi/2 \leq \arcsin(a) \leq \pi/2$ (radians)
61	ACOS	Arccos (a), $\lvert a \rvert \leq 1.0$, $0.0 \leq \arccos(a) \leq \pi$ (radians)
62	ATAN	Arctan (a) $\quad -\pi/2 \leq \arctan(a) \leq \pi/2$ (radians)
63	DATAN	
64	ATAN2	Arctan (a_1/a_2), not both $a_1 = 0.0$ and $a_2 = 0.0$
65	DATAN2	$-\pi \leq \arctan(a_1/a_2) \leq \pi$ (radians)
66	TANH	Hyperbolic tangent: $\tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$, $\lvert a \rvert < 740.3$
67	EOF	I/O status on unit a: = 0 if no EOF on previous read
68	IOCHEC	Parity status on non-buffer unit a: = 0 if no parity error on previous read
69	UNIT	I/O status on buffer unit a; = negative if unit ready, no error = 0 if unit ready, EOF on last operation = positive non-zero if unit ready, parity error on last operation
70	LENGTH	Number of central memory words read on previous BUFFER IN or READMS from unit a
71	LOCF	Address of argument a
72	LEGVAR	Returns -1 if a is indefinite, +1 if out of range (infinite) and 0 if normal
73	CPTIME	Returns floating-point CP seconds since beginning of job
74	ICPTIME	Returns integer CP milliseconds since beginning of job
75	RANF	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.

Standard or Non-standard FORTRAN*	Number of Arguments	Type of: Argument	Function Result	Compiler Type**	Example
N	1	REAL	REAL	L	A=TAN(B)
N	1	REAL	REAL	L	C=ASIN(D)
N	1	REAL	REAL	L	E=ACOS(F)
S	1	REAL	REAL	L	G=ATAN(H)
S	1	DOUBLE	DOUBLE	L	DOUBLE I,J $ I=DATAN(J)
S	2	REAL	REAL	L	P=ATAN2(Q,R)
S	2	DOUBLE	DOUBLE	L	DOUBLE K,L,M $ K=DATAN2(L,M)
S	1	REAL	REAL	L	S=TANH(T)
N	1	INTEGER	REAL	I	IF(EOF(I))1,2,1
N	1	INTEGER	INTEGER	I	IPAR=IOCHEC(J)
N	1	INTEGER	REAL	I	IF(UNIT(K))3,4,5
N	1	INTEGER	INTEGER	L	LEN=LENGTH(L)
N	1	Any type	INTEGER	I	COMPLEX C $ I=LOCF(C)
N	1	REAL	INTEGER	I	I=LEGVAR(P)
N	1	Any non-zero value	REAL	L	T=CPTIME(1.0)
N	1	Zero value	INTEGER	L	I=ICPTIME(0)
N	1	Any type	REAL	I	X=RANF(0.0)

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

Fortran Library Subroutines

	Symbolic Name	Definition and Limitations
1	EXIT	Terminate program execution and return control to operating system.
2	SLITE	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.
3	SLITET	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.
4	SSWTCH	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.
5	REMARK*	Place a message of up to 80 display code characters in the dayfile. The message must be terminated with 12 binary zeros (0000_8) in the lowest bits of the last word of the message even if an entire word is necessary.
6	DISPLA*	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 <u>not</u> 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)
7	TIME**	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 6 binary zero bits (00_8) even if an entire word is necessary.
8	SECOND	After calling SECOND, the argument <u>a</u> contains the central processor time used by the job in floating-point seconds (accurate to the nearest millisecond).
9	DATE	After calling DATE, the argument <u>a</u> 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.

* 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

Standard or Non-standard FORTRAN*	Number of Arguments	Type of Argument(s)	Compiler Type**	Example
N	0	(none)	I	CALL EXIT
N	1	INTEGER	I	CALL SLITE(3)
N	2	a_1 INTEGER a_2 INTEGER variable	I	CALL SLITET (2,J)
N	2	a_1 INTEGER a_2 INTEGER variable	I	CALL SSWTCH(1,K)
N	1	Hollerith	L	CALL REMARK (5HBEGIN)
N	2	a_1 Hollerith, a_2 REAL or INTEGER	L	CALL DISPLA (2HX=,20.2)
N	1	Hollerith	L	CALL TIME (9HCALCULATE)
N	1	REAL	L	CALL SECOND (TIMER)
N	1	Hollerith	L	CALL DATE (IDAT)

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

	Symbolic Name	Definition and Limitations
10	JDATE	After calling JDATE, the argument a contains the year and day of the year in an A10 format: bbbyydddbb 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).
11	CLOCK	After calling CLOCK, the argument a contains the time of day in an A10 format: bhh.mm.ssb where b is a blank character, hh is hours, mm is minutes, and ss is seconds.
12	RANGET	After calling RANGET, the argument a contains the current generative value of RANF.
13	RANSET	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.
14	OPENMS	Open mass storage random access file (see 9.9)
15	STINDX	Change index array of mass storage random access file (see 9.9)
16	READMS	Read mass storage random access file (see 9.9)
17	WRITMS	Write mass storage random access file (see 9.9)
18	READEC	Read extended core (see 9.8)
19	WRITEC	Write extended core (see 9.8)
20	FTNBIN	Blocked binary flag (see Appendix O)

Standard or Non-standard FORTRAN*	Number of Arguments	Type of Argument(s)	Compiler Type**	Example
N	1	Hollerith	L	CALL JDATE (IDAYS)
N	1	Hollerith	L	CALL CLOCK (IMLOK)
N	1	REAL	L	CALL RANGET (X)
N	1	REAL	L	CALL RANSET (X)
N	4	see 9.9	L	CALL OPENMS (99,INDX,101,0)
N	3	see 9.9	L	CALL STINDX (98,JNDX,51)
N	4	see 9.9	L	CALL READMS (97,A,100,3HA1A)
N	4	see 9.9	L	CALL WRITEMS (96,B,50,15)
N	3	see 9.8	L	CALL READEC (A,0,1)
N	3	see 9.8	L	CALL WRITEC (B,0,1)
N	3	see Appendix O	L	CALL FTNBIN (0,0,Dummy)

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

Alphabetic List and Index

I OUTPUT LISTING AND CROSS REFERENCE MAP

OUTPUT LISTING

The first line of an output listing gives the date this MNF compiler was assembled, the computer installation, the version of the operating system, and the PSR level of the compiler. The date and time the user's program was compiled appears after this on the first line. Unless the student BATCH mode is used (see below), the second line reprints the MNF control card that initiated this compilation.

The remaining lines of the source listing can each be divided into four fields. The first is the octal address (e.g. 000100B) which is the absolute octal location within the user's field length where the code for the current statement is being compiled (if there is any). See Appendix K for an explanation of the order of storage of variables, arrays, and machine code. The second field is the MNF sequence number of the Fortran statement followed by a star (e.g. 1*). Comments, blank lines, and continuation lines are not assigned a sequence number. The next field is the first 72 columns of the Fortran statement. The last field is columns 73-80 of the Fortran statement which are separated from the first 72 by ten blank columns. This aids debugging in that Fortran statements that are erroneously continued beyond column 72 have these characters separated from the actual Fortran statement. If there are any errors or informative diagnostic messages listed, they begin with the sequence number to which they refer (e.g. SEQ 9*), and are followed by the relevant message. If the error is fatal to execution, following the message is listed the error number, a consecutive count of the number of fatal errors encountered in the program (e.g. ERROR NO. 00001).

Student BATCH output begins with the reprinting of the double-period control card. The last 40 columns of the card are reprinted as the second line of output. Then the 5 character identification immediately following the double-period on the control card is reprinted vertically in large block letters on the remainder of the page. Except as noted above, all output following this is the same as for regular MNF. At the end of the job, a line is printed: END OF JOB, COST = $ n.m where n.m is the estimated cost of the job in dollars.

CROSS-REFERENCE MAP

The cross reference map is a dictionary of all FORMAT numbers, statement numbers, and names appearing in the current program, subprogram, or block data subprogram with their usage listed according to MNF sequence number. It appears only when the listing option L is on and is controlled by the R option of the MNF control card (see also Appendix A).

R = 0 No cross reference map is listed

1 Cross references are listed for FORMAT numbers and statement numbers sorted in ascending numeric order and Fortran names sorted alphabetically except for unused variables (nulls) having no execution references in the current program or subprogram. The common block names and numbers with corresponding block 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.

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

7 Combination of R = 2 and R = 5.

If the L option is on, the default value is R = 3; otherwise R = 0 if L = 0. The cross reference map may be further manipulated using the Fortran statements REFERENCES and NO REFERENCES. See 10.2 and 10.3.

Deletion of the cross reference map saves paper and about 20 % of compile time but eliminates a valuable aid for debugging and program information. Some compilation space is saved if the map is deleted since not all of the map information is generated by the compiler if it is not listed.

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 of programming considerations. In particular, a given Fortran statement may have more references

than actually appeared since the generated machine code may need to 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. The last inaccuracy is also due to optimization. In order to achieve maximum speed, storage of results into memory is delayed as long as possible. Thus, the variable or array element appearing on the left-hand side of an equal sign will often not be in the cross reference map until the next statement.

Lines in the 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 a statement number or FN for a FORMAT number. The third is either the absolute octal address of the statement number (e.g. 000503B) or the decimal length of the longest record within the FORMAT (e.g. LN=11); thus for FORMAT (3F10.5/4X,I20), LN = 30. The length should be 80 or less for a FORMAT using punch cards or 137 or less for line printer output. The last field is a list of sequence numbers denoting where the statement or FORMAT number was referenced. Each is followed by an identifying character (e.g. 24L) and possibly followed by / and a decimal integer (e.g. 19J/3). The integer is the number of times the reference appeared (19J appeared 3 times).
The characters are:

J	if the statement number was in a GO TO or IF statement
R or W	if the FORMAT was used in a READ/DECODE or WRITE/ENCODE/PUNCH I/O statement respectively
L	to denote the location in the program or subprogram of the statement or FORMAT number
D	if the statement number was in a DO or FOR statement
A	if the statement was in an ASSIGN or assigned GO TO statement
T	if the statement or FORMAT number was in the list of a TRACE statement

Lines in the name cross reference map each have six fields. First the name appears sorted alphabetically. Then a type letter 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. However, subroutine names will have Fortran implicit type. The next field is the main use of the name:

VARIABLE	for a simple variable name (including that on the FUNCTION statement if within a function subprogram) which is not in the formal parameter list or in an EQUIVALENCE statement
ARRAY	for an array name not in the formal parameter list or in an EQUIVALENCE statement
PARAMETER	for a simple variable name in the formal parameter list
PAR ARRAY	for an array formal parameter name
EQUIVLNCE	for a simple variable name appearing in an EQUIVALENCE statement
EQ ARRAY	for an array name appearing in an EQUIVALENCE statement
NAMELIST	for a namelist name
ST FUNCTN	for an arithmetic or logical statement function name
ENTRY	for an entry name
EXTERNAL	for the name of the current program unit (if it is a program or subroutine) or subroutines or function names which are not known to be standard
STANDARD	for known standard functions or subroutines
SYSTEM	for system subroutines (i.e. those names indirectly needed by the Fortran program)
FILE	for input/output file names (each followed by ≡)

A star is appended before the previous field if the name had no execution references (i.e. was a null and the cross-reference parameter R = 5 or 7). The fourth field is the absolute octal location of a variable or array name, or the parameter number in octal of a name appearing in the formal parameter list. The next field is the block name: a name or number for a labelled common block or / / for the blank common block. The last field is a list of sequence

numbers denoting where the name was referenced. Each sequence number is followed by an identifying character (e.g. 28D) and possibly followed by / and a decimal integer (e.g. 28U/5). The integer is the number of times the reference appeared (28U appeared 5 times). The characters are:

- U if the name was used in the statement
- S if the name was on the left hand sign of an equal sign or was in an input list or in a DATA statement
- D if the name was in a declarative statement
- I if the variable was a DO or FOR index variable
- P if the variable was a DO or FOR index parameter
- A if the variable was an assign variable in an ASSIGN or assigned GO TO statement

The block names for labelled, numbered and blank common and their decimal lengths are listed next. The blank common block is denoted as / /.

The last portion of the map is the list of variable names and array names sorted by octal address. First the address is printed followed by either B denoting an absolute address or C denoting an address relative to the beginning of blank common, and then the name is printed. The addresses appear sorted horizontally across the page.

The final part of the MNF output listing is the loader map. It contains three columns. The first is the list of program, subprogram, or common block names (/ / is the blank common block). The second is the octal address of the first word for common blocks, or the entry point address for programs and subprograms. The third column is the octal "length" of each item, i.e. the difference between the consecutive addresses in the second column, with the last value being the length of blank common if any, or the length of the last subprogram. Following the loader map is a line which gives the compile time in milliseconds (CTIME), the version number of MNF and the maximum of core needed to compile and execute.

An example is printed below which has a full cross reference map and includes every different kind of map element.

```
0001008        1*        SUBROUTINE AA0 (I,J,K,X,Y,Z)
0001008        2*        DIMENSION X(1),W(10,20),T(2)
0001008        3*        COMPLEX S
0001008        4*        COMMON /CMMN/ V,U
0001008        5*        COMMON /123/ B
SEQ     5* NOTE - NUMBER COMMON IS NON STANDARD  ****************************************************
0001008        6*        COMPLEX V
0001008        7*        DOUBLE PRECISION U
0001008        8*        LOGICAL M
0001008        9*        EQUIVALENCE (T(1),V)
0001008       10*        NAMELIST /Q/ S
SEQ    10* NOTE - NAMELIST IS NON STANDARD       ****************************************************
0001008       11*        DATA M/.TRUE./
0001008       12*        B(P)=P**2*S
0001028       13* C$     TRACE 10
SEQ    13* NOTE - TRACE OR NO TRACE IS NON STANDARD       ****************************************
SEQ    13* NOTE - C$ STATEMENT IS NON STANDARD   ****************************************************
0001228       14*        IF(I.EQ.J) GO TO 10
0001258       15*        ASSIGN 10 TO IGO
0001258       16*        DO 20 L=1,K
0001278       17* 20     W(L,L)=X(L)**2*SQRT(X(L))
0001418       18* 50     READ 40, W(J,2)
SEQ    18* NOTE - NON STANDARD INPUT OUTPUT STATEMENT     ****************************************
0001508       19*        PRINT 40,W(J,2)
SEQ    19* NOTE - NON STANDARD INPUT OUTPUT STATEMENT     ****************************************
0001558       20*        READ(5,40) W(I,3)
0001648       21*        WRITE (6,40) W(I,3)
0001728       22* 40     FORMAT (1X,E10.1)
0001728       23*        GO TO IGO, (10,30)
0001738       24* 30     W(J,2)=Y(K,Z)
0001768       25*        CALL EXIT
0001008       26* 10     ASSIGN 30 TO IGO
0001028       27*        DO 60 L=1,10
0001048       28* 60     W(L,L)=(X(L)-Z)**2*ABS(T(1))
0001138       29*        GO TO 50
0001138       30*        ENTRY AA0
SEQ    30* NOTE - ENTRY IS NON STANDARD          ****************************************************
0001218       31*        DO 70 L=1,K
0001238       32* 70     W(L,L)=DSIN(U)
0001338       33*        READ (5,Q)
SEQ    33* NOTE - NAMELIST IS NON STANDARD       ****************************************************
0001378       34*        WRITE (6,Q)
SEQ    34* NOTE - NAMELIST IS NON STANDARD       ****************************************************
0001428       35*        PUNCH 40,K
SEQ    35* NOTE - NON STANDARD INPUT OUTPUT STATEMENT      ****************************************
0001478       36*        RETURN
0001508       37*        END
SEQ    37* COMMENT - ARGUMENT - J SHOULD BE SET TO LOCAL VARIABLE FOR FASTER SUBPROGRAM SETUP      **********
SEQ    37* COMMENT - ARGUMENT - K SHOULD BE SET TO LOCAL VARIABLE FOR FASTER SUBPROGRAM SETUP      **********
SEQ    37* COMMENT - 32 WORDS USED IN SETUP OF SUBPROGRAM  ****************************************
```

```
NUMBER AND NAME CROSS REFERENCE MAP,  /N=REPEATED N TIMES
NUMBER TYPE ADDRESS  SN=STATEMENT NUMBER, FN=FORMAT NUMBER, J=JUMP, R=READ, W=WRITE, L=LOCATED, D=DO END, A=ASSIGN, T=TRACE
   10 SN    0005[illegible] 26L 23A 15A 14J 13T
   20 SN    0004[illegible] 17L 16D
   30 SN    0004[illegible] 26A 24L 23A
   40 FN     L[illegible] 11 35W 22L 21W 20R 19W 18R
   50 SN    0004[illegible]2B 29J 18L
   60 SN    0005[illegible]4B 28L 27D
   70 SN    0005238 32L 31D
 NAME TYPE MAIN USE ADDRESS  BLOCK U=USED, S=STORED, D=DECLARED, I=DO INDEX, A=ASSIGN, P=DO PARAMETER, *=NULL
AAO       *ENTRY                                30D
ABS      R STANDARD                             28U
AC       I*EXTERNAL                             1D
B        R*VARIABLE     000104B /123/           5D
DSIN     D STANDARD                             32U
EXIT       SYSTEM                               25U/2
I        I PARAMETER    000001B                 21U/2 20U/2 14U 2D 1D
IGO      I ASSIGN       000633B                 27S 26A 23U/2 16S 15A
INITBI.    SYSTEM                               20U 18U
INITBO.    SYSTEM                               35U 21U 19U
INPUTN     SYSTEM                               33U
INPUT=     FILE                                 18U
J        I PARAMETER    000002B                 25U/2 19U/2 18U/2 14U 1D
K        I PARAMETER    000003B                 35U 32U 31U 31P 24U 17U 16U 16P 1D
L        I VARIABLE     000634B                 32U/4 32S 31I 28U/5 27I 17U/7 17S 17U 16I
M        L VARIABLE     000105B                 11S 8D
NUMTRC.    SYSTEM                               26U
OUTPTN     SYSTEM                               34U
OUTPUT=    FILE                                 19U
PUNCH      FILE                                 35U
Q          NAMELIST       000550B                 34U 33U 10D
R        R*ST FUNCTN    000416B                 12D
READ1.     SYSTEM                               20U 18U
S        R VARIABLE     000000B //              12U 3D
SQRT     R STANDARD                             17U
T        R EQ. ARRAY    000100B /COMN/          28U 9D 2D
TAPE5=     FILE                                 33U 20U
TAPE6=     FILE                                 34U 21U
TERMBI.    SYSTEM                               20U 18U
TERMBO.    SYSTEM                               35U 21U 19U
U        D VARIABLE     000102B /COMN/          32U 7D 4D
V        C*EQUIVLNCE    000100B /COMN/          9D 6D 4D
W        R ARRAY        000106B                 32S 28S 25S 21U 20S 19U 18S 17S 2D
WRITE1.    SYSTEM                               35U 21U 19U
X        R PAR ARRAY    000004B                 28U 17U/2 2D 1D
Y        R EXTERNAL                             24U 1D
Z        R PARAMETER    000006B                 28U 24U 1D
BLOCK NAMES AND LENGTHS
        //              1     /COMN/          4     /123/          1
VARIABLE AND ARRAY NAMES SORTED BY OCTAL ADDRESS, TRAILING B=ABSOLUTE, C=RELATIVE TO BLANK COMMON
  000100B T            000100B V            000102B U            000104B B            000105B M            000106B W
  000633B IGO          000634B L            000000C S
```

APPX 10

J 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

---

*See also Appendix K for information about storage space optimization and Appendix N for information concerning optimization of Fortran Input/Output operations.

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 <u>not</u> 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,

```
      DO 10 J = 1, N
      DO 10 I = 1, N
   10 V(I) = VJ(I) + 0.5 * H * VK(J)
```

is better written as

```
      DO 10 J = 1, N
      TEMP = 0.5 * H * VK(J)
      DO 10 I = 1, N
   10 V(I) = VJ(I) + TEMP
```

```
      SUMJ = 9.78
      DO 50 J = 1, 70
   50 SUMJ = SUMJ + XT(J) * P
```

is better written as

```
      SUMJ = 0.0
      DO 50 J = 1, 70
   50 SUMJ = SUMJ + XT(J)
      SUMJ = 9.78 + SUMJ * P
```

and

```
    5 F = SIN(Y) * XN ** 2 + 2. * COS(XN) + 2. * SIN(Y)
      FP = 2. * SIN(Y) * XN - 2. * SIN(XN)
      XNP1 = XN - F / FP
      T = XNP1 - XN
      XN = XNP1
      IF (ABS(T / XN) .GT. 1.E-6) GO TO 5
```

is better written as

```
      Z = SIN(Y)
      TWOZ = 2. * Z
    5 F = Z * XN ** 2 + 2. * COS(XN) + TWOZ
      FP = TWOZ * XN - 2. * SIN(XN)
      XNP1 = XN - F / FP
      T = XNP1 - XN
      XN = XNP1
      IF (ABS(T / XN) .GT. 1.E-6) GO TO 5
```

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 SWTCH
      DO 20 I = 1, N
      IF (SWTCH) 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 /P1/ 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 (see 5.2.1). 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,

```
      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 I = 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.

```
      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 I) 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 <u>not</u> be stored into local arrays since except for very small arrays, this storing will take longer than the corresponding prestoration 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 such as in the equation in 2.6.1, the compiler may be better able to do them (see Section G, below).

However, the compiler cannot optimize subscript calculations when a loop contains complicated decisions, SUBROUTINE calls, FUNCTION references, I/O statements, extended DO ranges (6.3.4) or non-standard subscripts. In this case, the user can optimize himself by using single-dimension arrays and calculating the subscript explicitly from the equation in 2.6.1. 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) (see 2.6.1). Note that the Fortran compiler used must allow fewer subscripts than were declared in the DIMENSION, COMMON or type statement (MNF 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:

```
      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

```
      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 $P(X) = ((\ldots (a_n x + a_{n-1}) x + \ldots) x + a_0$. 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 = 6.28318530717958 * ANGLE
```

Appendix D contains many of the common constants used in computing just for this purpose. 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,

```
      IF (SQRT(X) .GE. Y) GO TO 10
```

is better written as

```
      IF (X .GE. Y ** 2) GO TO 10
```

(assuming Y ** 2 does not overflow) and

```
      D = 0.0
      DO 100 I = 1, N
  100 D = AMAX1 (D, SQRT (X(I) ** 2 + Y(I) ** 2))
```

can be better written as

```
      D = 0.0
      DO 100 I = 1, N
  100 D = AMAX1 (D, X(I) ** 2 + Y(I) ** 2)
      D = SQRT(D)
```

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
   20 JX = JCHAR(I)
```

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,

```
      S = 0.0
      DO 10 I = 1, N
   10 S = S + A(I)
```

is better written as

```
      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)
```

or

```
      T = 0.0
      DO 30 I = 1, N
   30 T = T + A(I) * B(I)
```

is better written as

```
      T = 0.0
      IF (N .LE. 1) GO TO 50
      DO 30 I = 1, N, 2
   30 T = T + A(I) * B(I) + A(I + 1) * E
   50 IF (N .NE. 2 * (N / 2))
     1   T = T + A(N) * B(N)
```

However, see section A below about further cautions.

15. Minimize the number of multiplications by use of the distributive law: A * C + B * C = (A + B) * C and A * C - B * C = (A - B) * 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,

```
      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) * T
```

should be re-written by reversing the order of the I and J loops as

```
      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	standard Fortran requires
`REAL A(10,20),B(10,20),`	
`1  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)`	`A(I, J) = FLOAT(X(I, J)) + B(I, J)`
`10 D(I, J) = X(I, J) * E(I, J) + F(I, J)`	`10 D(I, J) = FLOAT(X(I, J)) * E(I, J)`
	`1  + F(I, J)`

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

```
      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 * Z - (I - 1) / T + 6 * I
```

can be better written as

```
V = I
X = V * (Z + 6.0) - (V - 1.0) / T
```

However, for MNE 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 * IS - X) ** (2 * 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
      S = 4.0 - X                          completion)
    1 OS2 = 1.0 / (S * S)     (division done here instead of inside loop;
      PROD = 1.0                  see section J, below)
      DO 2 IR = 1, N
      PROD = PROD * OS2       (exponentiation replaced by multiplication)
    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 <u>not</u> 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
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,
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(1)
      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 <u>not</u> be re-written as

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

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
  100 ...
```

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

```
      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 <u>true</u> should be placed first in the statement (e.g. A .GT. B). Similarly,

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

is compiled as if it had been written as

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

so that whichever of the original relations is most likely to be <u>false</u> 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

```
      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

```
      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

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

in order to minimize the number of unneeded evaluations of FCN(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 <u>not</u> 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 MNF 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 <u>least</u> 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, AMINO, AMIN1, DMINO, 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 subscript only once in a DO or FOR loop when the subscript or similar subscripts 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 <u>not</u> 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)
      NM1 = N - 1
      DO 200 I = 2, N
      DO 200 J = I, NM1
      B(I - 1, J + 1) = D(J - 1, I)     Permanent subscripts generated for
      DO 100 K = I, J                   B and D.
      C(I, J, K) = CC                   Permanent subscript generated for C.
      C(I + 1, J + 1, K - 1) = C(I - 1, J + 1, K + 1)     Permanent subscript
      TEMP = E(K + 1, L, 7)             used for C.  Permanent subscript generated
      IF (TEMP .EQ. 0.0) GO TO 100      for E; original subscript occurs before
      E(K + 1, L, 7) = E(K - 1, L, 7)   transfer.  Permanent subscript used for E.
      E(K - 1, L, 7) = TEMP             Permanent subscript used for E.
  100 CONTINUE
  200 D(J, I) = B(I, J) * C(I + 1, J + 1, MM - 1)     Permanent subscripts
      DO 300 J = 1, N                   used for B and D.
      DO 300 I = 1, J                   Permanent subscript generated for B
      C(I + 3, J, MM) = B(I, J)         (but not C); MM not a variable dimension
      C(I, J + 1, MM - 2) = 1.0 + B(J, I + 1)     or loop control.  Permanent
      BB = B(I, I)                      subscript for B not used because not
      IF (BB .GT. CC) Z = D(I, J + 1)   similar.  Permanent subscript not generated
      X = D(I, J) * D(I, J + 3)         for D in dependent statement of logical IF.
      IF (BB .GT. CC) GO TO 300
      CC = CC + E(6, L, 6)              Permanent subscript not generated for
      X = X + E(6, L - 3, 6)            E after transfer statement.
  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:

```
DIMENSION D(6, 5)
Y = D(K + 3, J + 3)
```

The subscript is calculated as (D + 14) + 6 * 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

```
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

```
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

```
X = Y
Z = Y
```

use

```
X = Y
Z = X
```

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

```
A = 0
B = 0
```

is better written as

```
A = 0.0
B = 0.0
```

Also, group variables of the same type together. Thus

```
C = 1.0
N = 1
D = 1.0
M = 1
```

is better written as

```
C = 1.0
D = 1.0
N = 1
M = 1
```

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 = 1.0 / SQRT(X) * SIN(1.0 / SQRT(X))
```

must be re-written as

```
Q = (1.0 / SQRT(X)) * SIN((1.0 / SQRT(X)))
```

to take advantage of remembering the identical subexpression 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

```
IF (G .GT. (B + C) * D) GO TO 10
D = (B + C) * D
```

should be re-written as

```
TEMP = (B + C) * D
IF (G .GT. TEMP) GO TO 10
B = TEMP
```

and

```
XP = X * COS(PHI) + Y * SIN(PHI)
YP = X * SIN(PHI) - Y * COS(PHI)
```

should be re-written as

```
CT = COS(PHI)
ST = SIN(PHI)
XP = X * CT + Y * ST
YP = X * ST - Y * CT
```

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 (see section 5.5) 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 (see 3.1.2) is done as an addition by the compiler, e.g. 2.0 * 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, C1, C2 are COMPLEX operands):
C1 ± C2, C1 * C2, C1 / C2, C ** I, C / R, R * C, C * R, C ± R, R ± C
(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.

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 (Appendix A) 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 machinese 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

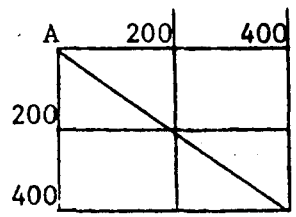
An example such as the usual way of programming matrix multiplication in Fortran illustrates the problem.

```
      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.




```
      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(1, I), A(1, I), 200)
   10 CALL READEC (F(1, 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, 1), B(1, L), 20000)
      DO 80 I = 1, 50
      DO 80 M = 1, 400, 50
      CALL READEC (G(1, 1), 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 6000/7000 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

	Operation / Machine:	6200	6700CP1	6400 6500	6600 6700CPO	7600
	INTEGER LOAD	16	15	13	8	8
	INTEGER STORE	14	13	11	10	1
	INTEGER + or -	9	8	6	3	4
	INTEGER *	60	59	57	10	5
	INTEGER /	112	106	94	44	29
SPC	INTEGER * 2^P	18	16	12	7	4
SPC	INTEGER / 2^P	18	16	12	7	4
	REAL LOAD	16	15	13	8	8
	REAL STORE	14	13	11	10	1
	REAL + or -	24	22	18	8	6
	REAL *	60	59	57	10	5
	REAL /	60	59	57	29	20
SPC	INTEGER → REAL	21	19	15	9	5
	DOUBLE LOAD	34	32	28	10	10
	DOUBLE STORE	30	28	24	13	3
	DOUBLE + or -	137	127	107	41	23
	DOUBLE *	300	292	276	34	22
	DOUBLE /	468	451	417	118	81
SPC	DOUBLE ± REAL	123	114	96	36	22
SPC	2 * D or D + D	58	54	46	19	10
SPC	DOUBLE * 2^P or DOUBLE / 2^P	137	134	128	18	15
SPC	D ** 2 or D * D	240	233	219	28	18

	COMPLEX LOAD	34	32	28	10	10
	COMPLEX STORE	30	28	24	13	9
	COMPLEX + or -	50	46	38	12	8
	COMPLEX *	292	284	268	32	19
	COMPLEX /	506	488	451	121	80
	COMPLEX ± REAL	34	31	25	7	7
	2 * C or C + C	29	27	23	9	5
SPC	COMPLEX * 2^p or COMPLEX / 2^p	137	134	128	18	15
	C ** 2 or C * C	221	215	203	27	15
	COMPLEX / REAL	121	119	115	59	38

Some Functions	Machine:	6400 6500	6600 6700CPO
ACOS		1399	264
ALOG		1041	206
ASIN		1383	259
ATAN		927	216
ATAN2		1017	248
COS		1248	221
EXP		898	180
SIN		1230	208
SQRT		856	153
TANH		974	196

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. (J-2)

2. Remove unnecessary statement numbers that are not being used. (J-3)

3. Place variables and arrays needed by subprograms into COMMON where practical. (J-3)

4. Use local variables in subprograms when a simple variable formal parameter would be needed many times within the subprogram. (J-5)

5. Use the DATA statement to prestore constants for once-only initialization. (J-6)

6. Use single-dimension instead of two- or three-dimension arrays when feasible. (J-7)

7. Consider a multi-dimension array to be single-dimension for setting all elements to a constant. (J-8)

8. Incorporate one-use subprograms into their calling or referencing program units. (J-9)

9. Use Horner's rule to evaluate polynomials. (J-10)

10. Evaluate constant subexpressions elsewhere (e.g. on paper) and replace the subexpression with that result. (J-10)

11. A ** (REAL integer constant) should be A ** (INTEGER constant). (J-11)

12. A constant k * 10 ** n should be written as kEn. (J-11)

13. Rewrite (-1) ** N to take advantage of the even/odd result. (J-11)

14. Avoid unneeded decisions in loops. (J-12)

15. Use the distributive law to minimize the number of multiplications. (J-14)

16. When possible, use library routines provided by the computer installation rather than writing your own. (J-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. (J-14)

18. Be careful when using non-standard mixed-mode arithmetic. (J-15)

19. Attempt to replace exponentiations by multiplications, multiplications by additions and loop tests by equivalent tests. (J-16)

Machine or Compiler Oriented Optimizations

A. Use one-statement DO or FOR loops whenever possible. (J-18)

B. Choose the fastest branch statements for the situation and put multiple decisions in decreasing likelihood of occurrence. (J-20)

C. Avoid decisions whenever possible especially by using ABS, MIN, MAX, and SIGN functions. (J-24)

D. Attempt to take advantage of MNF permanent subscripting. (J-24)

E. Write subscripts with the constant portion last so that the compiler can lump the constant portion with the array address. (J-27)

F. Put subscripts in standard form in a single statement (if possible) to help the compiler evaluate the subscripts only once. (J-27)

G. Help the compiler remember result values previously stored which remain in active registers. (J-27)

H. In single statements, help the compiler to remember to evaluate identical subexpressions only once; in multiple statements, do this yourself. (J-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. (J-29)

J. Multiply by the reciprocal whenever the possible slight inaccuracy can be tolerated. (J-29)

K. Avoid the unary operator minus preceding a variable. (J-30)

L. Use power-of-2 constants in multiplication or division because the compiler makes a special attempt to recognize them. (J-30)

M. Use 2 * variable rather than variable + variable. (J-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). (J-31)

O. Use REAL rather than DOUBLE PRECISION if it does not affect the accuracy of the program. (J-31)

P. After debugging is finished eliminate the listing and cross-reference map or generate and use an overlay. (J-32)

Q. Program large Fortran array problems to use adjacent data. (J-32)

K LAYOUT OF EXECUTION TIME MACHINE CODE

APPX 11

Each program unit (PROGRAM, SUBROUTINE, or FUNCTION) has the following memory structure:

	Input/output buffers defined implicitly or by PROGRAM statement (if this program unit is the main program)	Pseudo-Compass key names
This storage is assigned for variables and arrays appearing in declarative statements when the first executable statement is encountered	Labelled and numbered common blocks which were not previously seen in other programs or subprograms	
	Arrays and variables appearing in DIMENSION, type, and DATA statements	
This binary code is generated during processing of executable statements	Executable code equivalent to source statements (all addresses are relative to the final execution location)	
This is NAMELIST storage	List names needed for NAMELIST statements	
This formal parameter storage is done when the END statement is encountered	Formal parameter address storage (if the RUN/FUN calling sequence is used and this subprogram has more than 6 formal parameters)	
This is the entry point	Traceback word	(TRA. i.e. traceback)
	Program unit entry point	(XIT. i.e. entry/exit)
This binary code is generated when the END statement is encountered	Execution code to generate formal parameter address prestoration and variable dimension products (if this program unit is a subprogram) or a call of Q8NTRY (if this program unit is the main program) and setup of FETs (if the FTN calling sequence is used)	(PRS. i.e. preset)
This storage is assigned and/or preset during the processing of the END statement	Remote parameter lists for arithmetic statement functions (and for called subprograms if the FTN calling sequence is used)	(APTAG. i.e. actual parameter tag)
	Permanent subscripts	(IND. i.e. indices)
	Temporaries for subscripts and expressions	(TMP. i.e. temporaries)
	Temporaries for DO-parameter expressions during the DO/FOR range	(DOTMP. i.e. DO temporaries)
	Constants	(CON. i.e. constants)
	Formats	(FMT. i.e. formats)
	Variable-dimension variable values and products	(VARDIM. i.e. variable dimension)
	Simple variables which were not seen before the first executable statement	(VAR. i.e. variables)
	Names and statement numbers required by TRACE statements	(GV. i.e. generated variables)

The first program unit is stored beginning at location 101B. The binary code from externally compiled programs (B parameter on MNF control card) is placed following the last user-defined program unit. After this comes binary code from user libraries (U parameter) and then binary code from the system library. The last area is for blank common.

Since MNF compiles a complete program directly into absolute binary rather than putting out relocatable binary for each program unit, it requires that the storage needed by compiler and loader tables and the total needed by the execution code be available during compilation. There are two exceptions to this that the programmer can use to minimize needed field length. If the main program is the first Fortran source program unit, the storage space for I/O buffers is not allocated until execution begins. Also, because MNF does not allow the presetting of blank common by the DATA statement, the area assigned for blank common can be used for compiler and loader tables. Thus the programmer should place the main program first in his source deck and put arrays in blank common in order to keep field length as small as possible and in order to allow extremely large programs to be compiled within computer memory.

APPENDIX 12 MNF Performance

7600

JOB	CPU TOTAL			XEQ FL (100B)			I/O REQ			RMS CALLS		
	MNF	FTN	RUN	MNF	FTN	RUN	MNF	FTN	RUN	MNF	FTN	RUN
APROX	1.63	2.06	1.77	101	111	101	90	197	96	8	28	6
CTIM	1.22	1.12	1.20	543	551	542	23	49	16	6	10	4
EVANN	12.76	11.77	13.02	163	171	161	141	319	139	11	55	7
JEEZ	47.40	44.29	45.34	137	144	141	178	394	177	11	51	7
LEEDN	13.62	11.25	14.01	256	256	256	82	222	39	8	32	5
MAIN	1.04	2.14	1.69	424	430	426	243	348	242	12	47	10
MATRIX	0.72	2.41	1.07	217	230	224	64	319	53	8	62	5
PSTPKN	2.76	4.68	3.64	267	264	2066	109	451	81	10	70	7
RADM	0.33	1.23	0.55	322	326	320	57	322	54	7	63	5
SMATH	3.09	3.12	4.18	145	145	147	100	276	87	10	44	6
STIM	1.01	1.28	1.24	315	321	313	23	76	20	6	13	4
TAFEL	7.22	7.75	8.32	226	233	226	75	255	62	9	54	6
TOPSY	3.46	4.70	4.05	137	141	137	309	538	315	15	89	12
WIRE	0.42	0.64	0.64	121	133	120	100	68	95	7	8	6
TOTAL	96.68	98.44	100.72	4310	4401	4312	1594	3834	1476	128	626	90
Ratios (MNF=1)	1.00	1.02	1.04	1.00	1.03	1.00	1.00	2.40	0.93	1.00	4.88	0.70

Control cards:
```
RUN76(S) (6600)
MNF(B=BIN,R=3,E=2) (6600)
FTNX(OPT=2,R=3) (7600)
RUN76(S) (7600)
MNF(B=BIN,R=3,E=2) (7600)
```

APPENDIX 12 MNF Performance

6600

	CP TIME		XEQ f1(100B)		REQS.		SOURCE STATEMENTS
	MNF	RUN	MNF	RUN	MNF	RUN	
APROX	8.66	10.44	130	122	66	76	198
CTIM	6.54	7.78	561	551	83	73	88
EVANN	67.41	70.78	210	201	90	95	396
JEEZ	237.34	249.46	153	146	84	91	464
LEEDN	66.87	78.46	303	276	87	59	400
MAIN	6.35	10.61	454	446	100	96	367
MATRIX	4.61	7.35	247	245	80	86	585
PSTPKN							756
RADM	2.52	3.87	346	340	73	85	461
SMATH	16.70	22.80	163	156	83	75	331
STIM	5.48	7.47	331	321	117	111	181
TAFEL	36.45	47.83	266	261	72	77	360
TOPSY	17.55	24.05	165	156	133	122	507
WIRE	3.01	3.89	151	140	58	57	46
TOTAL	481.49	544.79	4436	4317	1126	1103	5140
Ratios (MNF=1)	1.00	1.13	1.00	0.97	1.00	0.98	

TECHNICAL INFORMATION DIVISION
LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720